## ABSTRACT

HABIB, MURSALIN Provisioning Algorithms for Service Differentiation in Middleware Appliance Clusters. (Under the direction of Dr. Yannis Viniotis).

Service oriented architectures (SOA) and XML-based Web Services have become the technology of choice in enterprise networks. These networks support multiple services and are typically architected in multiple computing tiers, with a main service tier for the business logic and a separate, "offload" tier, for, say, the CPU-intensive XML processing. The offload tier is typically populated by clusters of middleware appliances, usually hardware-assisted devices that are optimized for their tasks. Service differentiation refers to the generic problem of managing the enterprise network resources in order to achieve desired performance objectives on a per service basis. In our research, we defined SAA/SDA (**S**ervice **A**ctivation **A**lgorithm/**S**ervice **D**eactivation **A**lgorithm) and its variations that manage the CPU allocation in the appliance tier, in order to provide service differentiation. The main design objective of SAA/SDA is to overcome the disadvantages of the present known, static solutions. We analyze the performance of SAA/SDA via simulations.

Provisioning Algorithms for Service Differentiation in Middleware Appliance Clusters

by
Mursalin Habib

A thesis submitted to the Graduate Faculty of
North Carolina State University
in partial fullfillment of the
requirements for the Degree of
Master of Science

Computer Engineering

Raleigh, North Carolina

2009

APPROVED BY:

| | |
|---|---|
| Dr. Robert Callaway | Dr. Michael Devetsikiotis |

Dr. Yannis Viniotis
Chair of Advisory Committee

# DEDICATION

To My Grandmother, Shamsunnahar Begum

I still somehow childishly convince myself that if I go to your garden surrounded home in Ukil Para, Naogaon(Bangladesh) I will still be able to hug you in the Balcony, just as I did thousand times when I was a kid, a teen and a grownup. That is exactly the reason I will never go to Naogaon.

# BIOGRAPHY

Mursalin Habib was born in Bangladesh, in 1982. After attending seven schools and two colleges, he eventually graduated from Bangladesh University of Engineering and Technology with a Bachelors degree in Electrical and Electronics Engineering in June 2005. His undergraduate thesis was titled "Combating Coexistence issue by developing Smart Antenna in IEEE 802.11g WLAN". He takes great pride, being one of the founding member of "BUET EEE Undergraduate Research Group (BERG)". After undergraduate studies, he worked with Grameenphone Ltd. (a telenor company), Dhaka, for two years as a system engineer. In August 2007, he joined North Carolina State University as a graduate student. While working towards the Masters degree, he worked on his thesis under the guidance of Dr. Yannis Viniotis. He loves to take photos, bike, cook new dishes, eat out with a strict policy of "never order the same thing twice in life", think "green" and always "Question".

# ACKNOWLEDGMENTS

I would list my acknowledgements to my adviser Dr. Viniotis. Firstly, for giving me the opportunity to work in this project. Secondly, for walking me through it. And thirdly and most important, being a very flexible adviser. The flexibility and support he provided were indispensable for me to have a nice research experience. I would like to thank Dr. Devetsikiotis to serve as my committee member and of course, his course in Spring 2009 first exposed me to the obscure fun part lying inside network performance research. I also extend my thanks to Dr. Wang and Dr. Callaway, who agreed to serve as my committee member.

Dr. Khan, my undergraduate adviser, I still remember your EEE 301 class in my junior year. After that I have taken about 25 courses, EEE301 is still one of my favorite. And Dr. Alam, my undergraduate thesis supervisor, I acknowledge you for inspiring me for higher education, which I realize now, was a very rewarding and intellectually stimulating experience for me.

Thanks to Ma, Abbuji and my little brother Mehrab for the support and ten thousand minutes of phone calls every month. Staying away from you is my biggest sacrifice to get this degree. Thanks to Sonia, for supporting, accompanying and encouraging me. And of course for being the biggest distraction to finish this thesis. But, it was fun trying out all those exotic food with you around Raleigh. Thanks to my roommates: Ayon, Mahmud (Bhai), Tony and Anh. Thanks to Dipak (Da) for introducing me to the $n^{th}$ degree of freedom (it handles very well and saves me time). I will never forget, Merl and Amy Mangum for something that is called 'Southern Hospitality".

It was amazing two years in Raleigh, from spring to summer and fall. From a buzzing Dhaka with 13 million people to a serene calm Raleigh with one tenth the population (and one tenth the buzz), sometimes I felt bored and alone, yet, I enjoyed mostly. Yes my Bangladeshi friends of NCSU ruined my schedule time to time by having way too much fun than one person sanely should have, I am thankful, life was not as boring. And Ms. Elaine Hardin, if you do not know, I tell you, you made life easy for me and for many other graduate students.

# TABLE OF CONTENTS

# LIST OF TABLES

# LIST OF FIGURES

# Chapter 1

# Introduction

## 1.1 History and Evolution

Information Technology has become the basic building block of any enterprize today. Because of widespread deployment , ever changing technology, lack of homogeneity IT infrastructures has become very rigid when it comes to make change in the existing applications or developing a new application. Despite of development of more open standards, one of the biggest challenge maintaining an IT infrastructure is to deal with ever changing hardware and softwares, at the same time mainlining legacy systems to provide backward compatibility. This is becoming one of the biggest hindrance to provide new or enhances services to cater the business need.

The history of IT as the service providing architecture evolves, very interestingly defines the emergence of a system that can support multiple services, ensure high availability, facilitate new application deployment.

In the 50s, when computer was first used to facilitate business, the very common architecture was a centralized mainframe. These mainframes were the source of all the processing power and connected with dumb-clients across the network providing users with services. These dumb-clients didn't have the capability to process any request. They were used as Input/Ouput device mainly. The processing power contained in the mainframe shared among all the users.

As predicted by Moore's, computer became cheaper, smaller and powerful. Which instigated the introduction of personal computer in late 70s/early 80s. Provided with lim-

ited processing power, these PCs used to process simpler tasks while depended on mainframe/server for processor intensive task. Thus we saw the development of a communication model widely known as Client/Server model, which later stirred up the development of networking. Due to the advancement of networking technology, the world saw the first concept of distributed computer.

As computers became even cheaper, smaller and more powerful and network became faster, cheaper and widely available as internet, drove development of many advanced network oriented application started. As the applications were becoming more complex and dynamic in nature, multi-tier architecture was developed. Multitier architecture provided the industry with following functionalities,

**Offloading** Offloading services, such as Authentication, XML Processing, Separation of GUI and Back end processing

**Integration** Through Protocol Mediation and XML

**Intelligent Routing** Content Based Routing, Priority Routing, Service Partitioning

A very popular multi-tier architecture is three-tier architecture. The first tier usually is the user agent, communicates with the second tier, which usually hosts the application. The second-tier communicates with third-tier depending on the application requirement from data repository and eventually provides result to the user. One easy example would be user authentication. First tier provides the HTML page to the user to send user name and password for authentication. The second tier contains the application logic (one very popular example is PHP) that can create HTML pages and fetch required information from data repository stored in data base. This architecture provides an amazing flexibility to the developers by providing logic abstraction. The developer for second tier only need to know it's interfaces with connected tier. This also enables us to develop services, based on existing modules residing in multiple system. Some of the popular development platforms are,

**Sun Microsystem:** Java Remote Method Invocation (Java RMI)

**Microsoft:** Remote Procedure Call (RPC) and Distributed Component Object Model (DCOM)

**Object Management Group:** Common Object Request Broker Architecture (CORBA)

## 1.2 Service Oriented Architecture

Service Oriented Architecture (SOA) provides the facility of development and integration where different applications require to exchange data with each other. SOA targets an architecture that provides user the agility to combine functionalities of existing services and reuse them to produce new applications by aiming a very loose coupling of service with OS and development environment. Here each of the services provide interfaces that can be used for data exchange and communication. However these interfaces and services of course involves resource overhead. Because of the facility of reusing, the cost and time for developing a new application is quite low for SOA (thus it is very promising) as all necessary requirements for the new applications are already in the system to serve existing application.

Considering flexibility, reuse of existing software and addressing the complexity to be satisfied the ever changing business goals in this era, Service oriented architectures (SOA) is becoming the automatic choice of technology [1], [2].

A very successful implementation of SOA is Web services that exposes functionality and solve the point-point integration problems. As web services are gaining popularity, many enterprizes are now exploring way to utilize the capability and flexibility of westerlies from enterprize to enterprize exploiting loose coupling, abstraction and reusability of SOA. Web services also leverage the ubiquitous nature of XML as a universal message format. However, XML processing involves high CPU overhead mostly because of parsing. There comes the requirement for very specialized, hardware appliances for XML processing. There facilitator appliances are called called middleware appliances or SOA appliances. System administrators all over the world, as industry standard, very often position these middleware appliances on the edge of the enterpriser network, as a very distinct tier before the service tier. For higher processing power and availability, clusters of appliances are commonplace today.

## 1.3 Prior Work and Motivations

Multiple classes of service requests are supported by an enterprize network, defined frequently as service domains [3], [4]. For our research, by *service domain* we mean an

application deployed in the network that provides very specific service. The very generic problem here is to manage the enterprize network resource to achieve performance target in per domain resolution. This generic problem is very often referred as *Service differentiation.* Resource may include the CPU processing power at any tier, performance could be in terms of waiting time or throughput.

There are different ways to address this allocation mechanism to achieve target resource share or performance. One very good widely discussed example of allocation mechanism is "priority-based" CPU scheduling for service domains as discussed in [5]. It requires per domain buffering and typically used in the server tier. A very common alternative approach, specially for inexpensive setups with no built-in intelligence (e.g., FCFS buffering) and no CPU scheduling, is *"activation/deactivation"* of service domains at the gateway. If more CPU resource is requires to achieve the goal, service domain gets activated in the same or other appliances in the cluster. Deactivation can be done from a subset of the appliances, as deemed necessary to achieve the goal. So, activation/deactivation of service domains actually attempts to change the rate at which requests gets into the system through the gateway. Thus we control the allocation of resources or performance class indirectly, as resource utilization or performance depends on arrival rate of that service domains.

According to our knowledge, the two possible known solutions approaches for providing differentiated services via activation/deactivation actions, as we describe in this section. Both known approaches result in following disadvantageous situation,

- Inefficient use of appliance resources
- Inability to provide service differentiation. We address both issues in this paper

Here the discussed mechanisms that solves the Unqualified Service Differentiation problem via activation/deactivation actions only. The motivation of our research focus on this type of mechanism comes from following reasons,

- Existing, commercially available appliances utilize this method
- In systems with large numbers of service domains, multiplexing multiple domains onto the same buffer forces FIFO scheduling

To the best of our knowledge, the two known solutions, both assumes a gateway (typically an HTTP router/IP sprayer, HRIS) to distribute service load to the appliances

arranged in cluster. This HRIS simply routes for example an URL, and uses IP spraying to send traffic to appliances without ay deep-content inspection.

For the first solution, the administrator group all the appliances in a single group and enables them all to process service requests for any given service [6]. The gateway forwards a service request to each of the appliances as it routes the request to the appropriate service port for service-specific processing. It has several major disadvantages. Firstly, enabling all service domains in every appliance is much more difficult to control the differentiated services across service domains competing for the same resources from appliances. Though an IP sprayer (i.e. gateway) actually can spread the load based on policy amongst different appliances in the cluster, it cannot however, meter the effect of a specific service request on resource. This means providing differentiated service amongst competing service domains is impossible. For example, if the system administrator wants to enforce a policy that allocates up to 50% of total CPU to a chosen service domain, the whole system only can hope that it will be evenly spread across the appliances, which for a system with three service domains, results in a situation where each service domain receive 1/3 (33%) of the total resource under overload condition. The other problem, very much detested by the system administrator, is the improbability of controlling any spatial locality [6],[7].

The second solution, the legacy approach where the system administrator statically allocate a portion of the appliance resource to each of the service domains as deemed necessary by the administrator. Here, each appliance is assigned to a specific service domain(s) which it will serve. Here, service requests for a specific service domain are served only on specific appliance and we achieve spatial locality. Also, the administrator can allocate appliances for service domains keeping the target goal in mind for each individual services and achieve in a rude sense, some level of differentiated service. It has also some shortcomings. The difficulty of leveraging the white space of the appliance service on service for incoming request to satisfy the goal wastes valuable system resource and overall system thus may run under-utilized. Apart from this major shortcoming, this method cannot adapt to the ever changing requests and prevailing conditions. So, unless we can predict future traffic perfectly, which is impossible, this method will lead the system to inefficient resource partitioning and this violating intended differentiated service goals [8],[9],[10],[11],[12],[7],[13]. Our research here is to eradicate these disadvantages by describing how an algorithm can be designed to affect *dynamic* provisioning of service domains amongst a cluster of appliances.

Unlike the known solutions, service domains are not statically bound to a particular (subset of) appliances.

The main contribution of this research is proposing such an algorithm in contract to presently available solutions, has following advantages,

- Capable of providing arbitrary allocation of CPU resources to service domains, thus achieving true service differentiation
- Utilizes appliance resources in an efficient manner, and thus it leverages processing white-space across all appliances
- Enhances service locality
- Doesn't require manual configurations

## 1.4 Outline of the Thesis

This is how the thesis is organized,

**Chapter 2: Problem Description and Proposed Algorithm** In this chapter, we define the system architecture from SOA perspective and discuss the requirements of an algorithm for service differentiation. Based on the requirement we propose SDA/SAA algorithm with examples.

**Chapter 3: Simulation and Result Analysis** We start the chapter mentioning the research questions we are going to answer. Then we provide description of the simulator used. Followed by experiment design to address these questions with simulation results. Suggested modification to the algorithm follows to solve some of the shortcoming discovered in the experiment. The chapter ends with a summary of the results.

**Chapter 4: Conclusion and Future Work** We summarize our contribution. Then we provide direction to future research direction on SDA/SAA algorithm and service differentiation with brief description.

# Chapter 2

# Problem Description and Proposed Algorithm

The architecture of the system under consideration is given in Figure 2.1. Service Requests to served in the system arrives from clients, over a transport network. At the extreme periphery, the first entry point to the system, there sits a Gateway which has the responsibility of distributing Service Requests to different appliances. Different types of Service Requests are grouped into Service Domains. The servers organized in "service tier" process service requests of different service domains. Appliances, allocated in "offload" tier, as its name implies offload the work for "service tier" by preprocessing incoming service requests. These appliances (Middleware) accommodate bursty traffic by buffering. For this research, we only considered FIFO service discipline thus without any preemption. [1].

In our research, we focus on developing algorithms by managing the "island" of middleware appliances (discussed in section 2.1) to achieve the goals described in Section 2.2.

## 2.1 System Architecture

As depicted in Figure 2.1, the system under consideration has three tiers. A more detailed hierarchical picture can be seen in Figure 2.2.

[1]Even with the presence of a CPU scheduling algorithm inside the appliance, when the number of service domains exceeds the number of buffering classes, service requests within a buffering class are still processed in a FIFO manner.

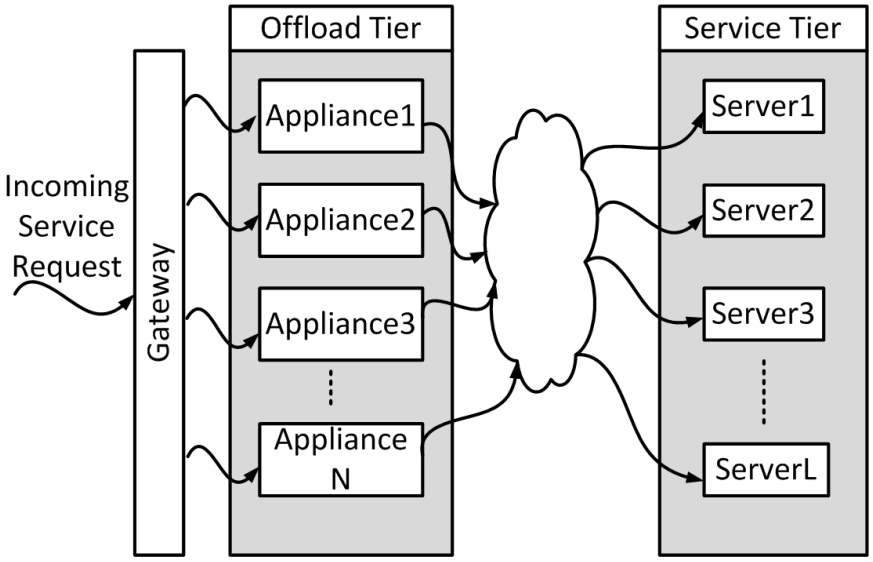


Figure 2.1: Abstract Architecture of a Two-tier Enterprise System.

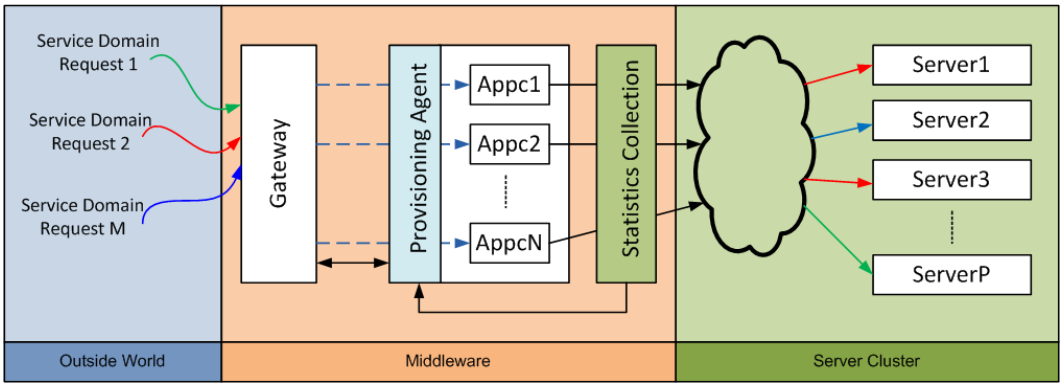


Figure 2.2: System Architecture

### Gateway

Gateway is the first entry point of the system, where service requests from customers arrives via transport network. Gateway maintains communication with provisioning agent to enforce the algorithm by forwarding incoming requests to proper server based on activated instances. So, gateway has to know which appliance can process what service domains. Gateway lacks buffer, thus it cannot store any service requests.

**Provisioning Agent**

Provisioning agent is responsible for collecting statistics from the appliances and calculating activations and deactivations based on performance and adjusting activated instances in appliances. Provisioning agent also signals the gateway that gateway can route service requests in proper manner. It is very easy to confuse provisioning agent as a separate hardware or physical entity. But it is rather a logical entity, may exist in one or more appliances as a stand-alone or distributed application.

**Appliances**

Appliances are the "Middleware" for the system. Appliance cluster preprocesses incoming service requests (e.g. security authentication and authorization, content based routing, mediation, XML/SOAP message processing, protocol translation) and reduce some workload from the server cluster, thus the name "Offload tier". Each of the appliance has a buffer to store unprocessed service requests and these unprocessed service requests get processed in FIFO manner.

**Servers**

Servers process the bulk of the service request. Appliances, after finishing the preprocessing routes the rest of partial processed requests to servers for final servicing. This is why, this tier is known as "server tier". Before the advent of Service Orient Architecture in service industry, "server tier" used to be the only tier beyond Gateway or load balancer.

## 2.2 Goals and Requirement

Being an dynamic reactive algorithm, the algorithm has the potential of meeting many different QoS/service contract parameter. We describe the widely accepted requirement in subsection2.2.1 and attainable goals in subsection 2.2.2 for such algorithm.

### 2.2.1 Requirements

As proposed in [14], requirements for service differentiation algorithm to maintain QoS and service contract can be classified as follows,

**Response Time Guarantee** The algorithm should be able to control concurrent services with a specific response time limitation for each services. The SLA enforcement scope here is individualistic for each service while the control is global.

**Service Differentiation** The algorithm should provide service differentiation as services are proved to the customers. This enables the administrators to segment customers into different categories and provide service of different class.

**Grouping of Interaction** The algorithm should group different interaction into session to reduce overhead.

**Compatibility** The algorithm that facilitates service differentiation must not require modification in the existing software or application logic. Rather, it should possess the capacity of running as an independent entity.

**Simplicity** The algorithm should be easy to configure and should reflect service contract or business goal in a simple manner.

**Scalability** The algorithm should be scalable enough that it can be deployed in a single service environment as well as in a cluster.

**Overload Protection** The algorithm should provide protection against overload.

**Maximum Utilization of Resources** The algorithm should utilize all the white space so that resource doesn't get wasted.

### 2.2.2 Goals

Service contract may include service differentiation to prioritize different classes of services to achieve different goals as per business need. Some of the goals that can be used to formulate the problem are as follows,

#### Provide "unqualified" Service Differentiation

Provide Service domain $m$ with upto a certain percentage, $P_m$, of CPU cycles in the appliance cluster. The target percentage for each service domain is not dependent on overload conditions, hence we call the percentage "static". No other constrains are enforced in this goal.

**Provide "overload" Service Differentiation**

Provide Service domain $m$ with upto a certain percentage, $P_m$, of CPU cycles in the appliance cluster when system is not overloaded. Once the system is overloaded, provide the Service Domain $m$ another percentage of CPU cycles, $P'_m$. In this goal, the target percentage for one service domains depends on loading condition. Like the first goal, no other constrains are to be implied.

**Guaranteed Throughput**

Provided service differentiation such that service domain $m$, gets $C_m$ requests serviced per unit time, when the system is under-loaded. If the system gets overloaded, the service domain $m$ gets $C'_m$ requests processed per unit time.

**Average Response Time Guarantee**

Guarantee that service domain $m$ have its request processed within $T_m$ unit time on average when the system is under-loaded. While the system gets overloaded, the guarantee is updated to $T'_m$

In reality, service contract or SLAs may require more than one goals to achieve at the same time. In that case the problem becomes a optimization problem, which is for this research we render out of scope. In this research, we considered and proposed an algorithm for to achieve unqualified service differentiation as described in section 2.4.

## 2.3 Approach to the Problem

The algorithm proposed in the thesis is a closed loop algorithm. It restricts the incoming traffic based on the collected periodic performance measurement from the appliances as feedback, to meet the differentiation goal.

The intuition supporting the solution is very simple. If the last measurement indicates the service domain didn't achieve the goal, the algorithm allows more request to come in until the next measurement than it did for previous measurement period. While if the last measurement signals that the service domain achieved beyond the goal, then the algorithm restricts the arrival until the next measurement. As the algorithm controls the

incoming rate, that particular domain is expected to see a increase/decrease in terms of CPU resource utilization as share of the total resource. As the target/goal attainment will be judged by the achieved share of total CPU cycle, the only required statistics in the next measurement instance is the CPU utilization per service domain collected from the last instance until the next.

To describe the algorithm we need the definitions provided in subsection 2.3.1.

### 2.3.1 Definitions

**Provisioning Agent (PA)** is responsible for deciding on activation/deactivation of service domain instances in the appliance cluster. This agent can be implemented as a centralized or distributed application, residing on one or more appliances or a separate compute node.

**Decision Instant ($T_k$)** is the $k^{th}$ decision instant at which PA activates/deactivates service domain instances based on the algorithm outcome. As denoted in Fig. 2.3, at $T_k$, all the measurements collected in the time interval $(T_{k-1}, T_k)$ are evaluated; activation and deactivation of service domains are enforced. In our simulations, $T_k$ is assumed to form a periodic sequence, for simplicity.

**Target CPU % ($P_m$)** is the desired percentage of CPU resources to be allocated to the $m^{th}$ service domain.

**Achieved CPU % ($X_m(T_k)$)** is the percentage of the cluster CPU resources obtained by the $m^{th}$ service domain until time $T_k$.

**Down and Up Tolerances $DT_m$ and $UT_m$** : in order to avoid unnecessary oscillations and overhead, when the Achieved CPU % is "close enough" to the Target CPU %, i.e., when

$$P_m - DT_m < X_m(T_k) < P_m + UT_m \tag{2.1}$$

the service domain is excluded from activation/deactivation.

**Utilization Matrix ($U_{nm}$)** is the achieved resource utilization (e.g., total CPU time used) by the $m^{th}$ service domain in the $n^{th}$ appliance, in the time interval $(T_{k-1}, T_k)$.

**Instantiation Matrix ($B_{nm}$)** is the number of instances of the $m^{th}$ service domain that should be activated in the $n^{th}$ appliance during the time interval $(T_{k-1}, T_k)$. This is the main decision variable that the PA computes. The mechanism of signalling $HRIS$ about the values of $B_{nm}$ and how PA collects the measured statistics from the appliance cluster is out of the scope of this paper.

**Number of Appliances** $N$ is the total **Number of Appliances** in the cluster.

**Number of Service Domains** $M$ is the **Number of Service Domains** supported by the system.

**Groups** $A$ **and** $D$ denote the ranking of service domains. When service domain $m$ is not achieving its Target CPU % ($P_m$), the PA selects it to be activated in the next decision instant in one or more appliances and thus includes it in Group $A$. Similarly, when service domain $m$ is allocated more than its Target CPU % ($P_m$), the PA selects it to be deactivated in the next decision instant in one or more appliances and thus includes it in Group $D$.

### 2.3.2 Available Actions

As described in subsection 2.3.1, Provisioning Agent is responsible for take activation and deactivation decision. And these are the only two actions available for the algorithm considered in this research. Each activation and deactivation decision invoked overhead and of course time required to process those actions, which assumed to be zero in this research. Fig. 2.3 depicts the instances when decision are calculated and enforced.

#### Action Owner

As provisioning agent takes the decision as per algorithm for activation and deactivation, it is gateway's responsibility to enforce the decision in the manner that it distributes incoming service requests to different appliances.

As mentioned, actions are,

**Activation** Increase the number of instances for service domain $m$ in appliance $n$ by one, that service domain $m$ gets more CPU cycles

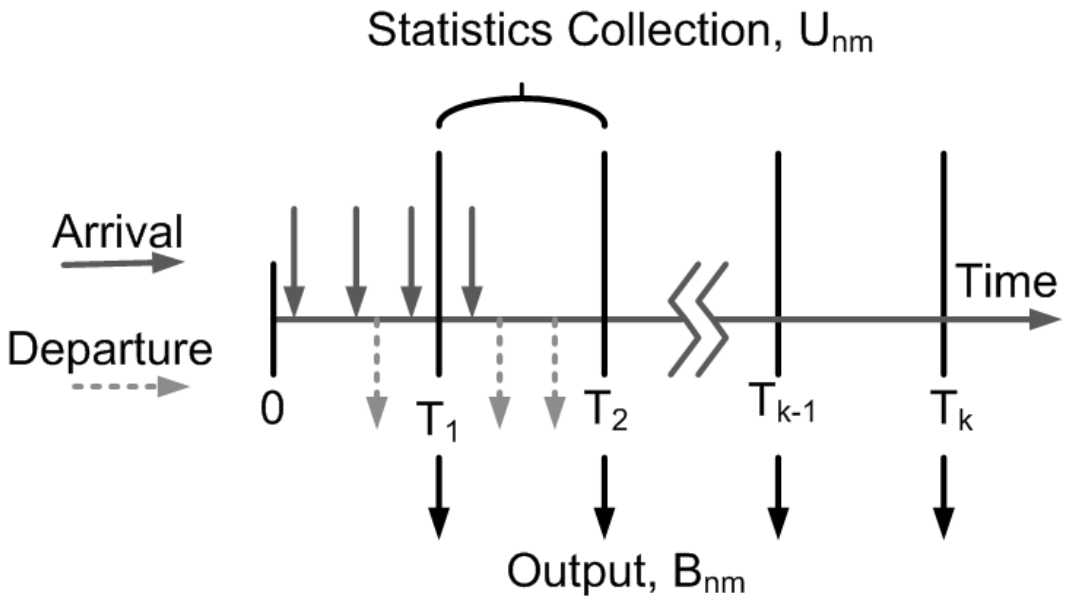


Figure 2.3: Decision Instances $T_k$.

**Deactivation** Decrease the number of instances for service domain $m$ in appliance $n$ by one, that service domain $m$ gets more CPU cycles

### 2.3.3 Performance Measurement

Depending on the goal, following performance matrix can be taken into consideration as measurement to enforce service differentiation and QoS.

**CPU Utilization** For each appliance, the utilization of CPU resource per service domain.

**Buffer Size** For each appliances, the number of service requests waiting to be serviced.

**Average Response Time** For each appliances, the average time a service request has to wait before getting serviced.

**Throughput** For each appliances, the number of service request served per service domain.

In this research we restricted the goal for the algorithm is to provide unqualified service domain, thus only collected matrix is CPU Utilization.

## 2.4 Proposed Algorithms

The algorithm we are proposing, is a closed-loop measurement based algorithm.

### 2.4.1 Algorithm Summary

At each decision instance, at time $T_k$, $k = 1, 2, \ldots$

1. **Collect measurements** ($U_{nm}$) from the $N$ appliances.

2. **Calculate the actual percentile of allocated resources** for the $M$ service domains using the iterative equation:

$$X_m\left(T_k\right) = \frac{1}{kN}\sum_{n=1}^{N} U_{nm} + \frac{k-1}{k} X_m\left(T_{k-1}\right)$$

   This equation is providing long-term time average of CPU utilization in a recursive way.

3. **Calculate Thresholding** operations according to Eqn. 2.1.

4. **Evaluate and Rank Performance** to check if the goal is met. Intuitively, the lower $|X_m(T_k) - P_m|$ is, the "better" the performance of that particular service domain. The service domain is placed in Group $A$ or $D$ as follows. When

$$X_m(T_k) - P_m \geq 0$$

   the service domain meets or exceeds its target and is thus included in Group $D$. When

$$X_m(T_k) - P_m < 0$$

   the domain misses its target and is thus included in Group $A$.

5. **Apply Deactivation Algorithm** to deactivate instances of all service domains in Group $D$ as per algorithm SDA (defined in subsection 2.4.2).

6. **Apply Activation Algorithm** to activate instances of all service domains in Group $A$ as per algorithm SAA (defined in subsection 2.4.2).

7. **Feedback** these decisions (expressed as values of the matrix $B_{nm}$) to the gateway.

Here, during the next interval $(T_k, T_{k+1})$, hunch is that the rate of arrival of service request for a domain will be changed favorably. By activation of more instances of a service domain will, conceivably increase the rate at which service requests enters the appliance tier. So, there will be an increment in the share of the cluster CPU resources from that particular service domain's point of view. It is very important to notice that, the increment in terms of CPU resource percentage may not be obtained during the following cycle, due the fact the system deploys FIFO service discipline in the buffer. In the same way, deactivating instances of a service domain will, hopefully, decrease the rate at which requests enter the appliance tier. Thus, the domain will eventually see a decrease in its share of the cluster CPU resources. [2]

### 2.4.2 SAA/SDA activation and deactivation algorithm

The activation and deactivation of service domains can be sanctioned in endless possible ways. Few selected possibilities, that is considered in our research is as follows,

### 2.4.3 Algorithm SAA1/SDA1

1. (SDA1) Deactivate one instance of every service domain in Group $D$ in appliances which run service domains in Group $A$ to free up CPU cycles utilized by domains in Group $A$.

2. (SAA1) Using the instantiation matrix $B_{nm}$, activate one instance of every service domain in Group $A$, in appliances which run service domains of Group $A$.

Note that both SDA and SAA will result in a change of the values stored in the matrix $B_{nm}$. As an example, suppose that we have 4 appliances and 5 service domains with target CPU percentages set at $\{35\%, 25\%, 15\%, 10\%, 5\%\}$ . Suppose that the initial value for the instantiation matrix is given by

$$B = \begin{bmatrix} 1 & 0 & 10 & 1 & 4 \\ 10 & 3 & 1 & 2 & 4 \\ 0 & 5 & 8 & 4 & 1 \\ 0 & 0 & 2 & 4 & 10 \end{bmatrix}$$

[2] We can reasonably assume the time required to finish these steps is much less than $T_{k+1} - T_k$, thus we can consider the algorithm execution duration to be zero to facilitate easy analysis

Suppose that the collected $U_{nm}$ values result in actual CPU percentages $X_m(T_k)$ equal to $\{11\%, 8\%, 19\%, 11\%, 19\%\}$. The tolerances for thresholding are set at 2%, so the algorithm calculates group $A = \{1, 2\}$ and group $B = \{3, 5\}$. Therefore, we must activate domains 1 & 2 and deactivate domains 3 & 5. Now based on the algorithm described (SDA), there is no instances of domains 1 and 2 activated in appliance 4, so there is no need to deactivate instances of domains 3 & 5 in that appliance. However, as there are instances of domains 1 and 2 running in appliances 1, 2 and 3, there will be deactivations of domains 3 and 5 in these appliances. Note that, because there is only one instance of domain 3 activated in appliance 2 and only one instance of domain 5 activated in appliance 3, these two entries will be kept unchanged. Because of the deactivation, as some of the CPU resource utilized by domain 3 and 5 is freed up, under-utilized domain 1 and 2 can take advantage of that and activate one more instance of domain 1 and 2 in appliance 2, domain 1 in appliance 1 (domain 2 cannot be activated in appliance 1 as it is not already activated there) and domain 2 in appliance 3. So, after SDA, we will get (changed values are in bold face),

$$B = \begin{bmatrix} 1 & 0 & \mathbf{9} & 0 & \mathbf{3} \\ 10 & 3 & \mathbf{0} & 1 & \mathbf{3} \\ 0 & 5 & \mathbf{8} & 4 & \mathbf{1} \\ 0 & 0 & 2 & 4 & 10 \end{bmatrix}$$

and after SAA, we will get instantiation matrix as follows (changed values are in bold face),

$$B = \begin{bmatrix} \mathbf{2} & 0 & 9 & 0 & 3 \\ \mathbf{11} & \mathbf{4} & 0 & 1 & 3 \\ 0 & \mathbf{6} & 8 & 4 & 1 \\ 0 & 0 & 2 & 4 & 10 \end{bmatrix}$$

As the algorithm activates and deactivates service domain, the anticipation and hope is that the algorithm will change the rate of arrival of service requests in favor of the goal, achieving CPU share as per SLA. However, as SAA1/SDA1 doesn't limit the number of activated instances in the appliances, it is vulnerable to thrashing [footnote]. SAA1/SDA1 also allows the number of instances to go down to zero, which may result in a failure to meet the goal. As the pattern of the incoming service requests is not deterministic the system will go through series of activations and deactivations to adjust itself to achieve the goal. Tighter tolerance for threshold expected to have higher oscillation than looser

tolerance. The measured statistics, based on which the algorithm takes decision is a time average value, the algorithm output converges as $T \to \infty$.

### 2.4.4 Algorithm SAA2/SDA2

1. (SDA2) Deactivate one instance of every service domain in Group $D$ in appliances which run service domains in Group $A$ to free up CPU cycles utilized by domains in Group $A$. Denote the number of freed up slots by $F$.

2. (SAA2) Order the service domain in Group $A$, from most hurting to least hurting. Then activate one instance of every service domain in Group $A$, in appliances which run service domains of Group $A$, until $F$ slots are used.

Here, like SAA1/SDA1, both SDA and SAA will change the values in matrix $B_{nm}$. Like the previous example, with 5 appliances and 5 service domains with target CPU percentages set at $\{35\%, 25\%, 15\%, 10\%, 5\%\}$ . Suppose that the initial value for the instantiation matrix is given by

$$B = \begin{bmatrix} 1 & 0 & 10 & 0 & 0 \\ 10 & 3 & 0 & 2 & 4 \\ 0 & 5 & 8 & 4 & 0 \\ 0 & 0 & 2 & 4 & 10 \end{bmatrix}$$

After collecting the statistics using the 2% tolerance for thresholding the algorithm calculates group $A = \{1, 2\}$ and group $B = \{3, 5\}$. Therefore, we must activate domains 1 & 2 and deactivate domains 3 & 5. As the deactivation process is exactly same for both the algorithms, after SDA1, we will get (changed values are in bold face),

$$B = \begin{bmatrix} 1 & 0 & \mathbf{9} & 0 & 0 \\ 10 & 3 & 0 & 1 & \mathbf{3} \\ 0 & 5 & \mathbf{8} & 4 & 0 \\ 0 & 0 & 2 & 4 & 10 \end{bmatrix}$$

Here, the only difference between SDA1 and SDA2 is SDA2 keeps track of the number of changed value which is 3 for this case. and after SAA, we will get instantiation matrix as

follows (changed values are in bold face),

$$B = \begin{bmatrix} \mathbf{2} & 0 & 9 & 1 & 3 \\ \mathbf{11} & \mathit{3} & 1 & 2 & 3 \\ 0 & \mathbf{6} & 7 & 4 & 1 \\ 0 & 0 & 2 & 4 & 10 \end{bmatrix}$$

As SAA activates instances in $B(1,1) = 1 \Rightarrow 2$, $B(2,1) = 10 \Rightarrow 11$ and $B(3,2) = 5 \Rightarrow 6$, it uses up all allowed activation as tracked in $F$. So, it cannot activate $B(2,2) = 3$. While if SAA1/SDA1 were used, it would have activated $B(2,2) = 3 \Rightarrow 4$

Here, as we stated before, we expect that the algorithm will qualify the rate of arrival of service requests by changing the number of activated instances in the appliances. In this variation of the SAA/SDA algorithm, we limit the number of activation. So, number of activation in each execution of algorithm is never more than the number of deactivation. Thus, total number of instances in the whole appliance cluster doesn't grow indefinitely. However, this variation doesn't protect the algorithm from the vulnerability that the number of instances in the appliances may go down to zero, which may restrict the algorithm to reach the goal. Due to stochastic nature of the incoming request, the algorithm will require several execution to achieve the target. Each execution will cause oscillation in $B_{nm}$, magnitude of the oscillation will depend on the tolerance. As we discussed for SDA1/SAA1, the algorithm will converge as $T \to \infty$.

### 2.4.5 Alternative Approaches to the Problem

In the proposed algorithm, the goal is met by throttling the arrival. Several process sharing and scheduling mechanisms are discussed in [Cite GPS Parekh Gallagar][Cite M. Shreedhar and G. Varghese][Cite Stiliadis, D. and Varma, A, a general model for analysis of traffic scheduling algorithms - TON][Cite Demers, A.; Keshav, S.; Shenker, S - Efficient fair queueing using deficit round robin, SIGCOM]. Many of these scheduling methods are widely implemented in network traffic differentiation, such as IETF standard DiffSERV and IntSERV. Many of these algorithms has the potential for the application of service differentiation.

### Dedicated Queue for each class of Services

An approach could be defining dedicated queues for each class of the services. As shown in Figure. 2.4, the weighted round robin schedulers accepts service requests to provide service. Scheduler can be, as per the goal follow different disciplines. To achieve the goal, the algorithm should successfully calculate the weights or priority based on the collected statistics.

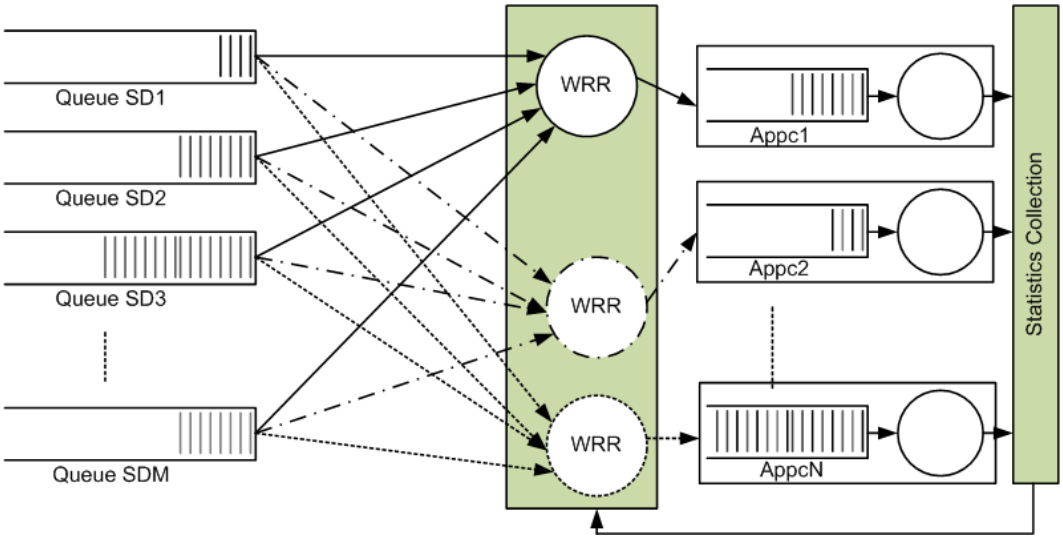


Figure 2.4: Weighted Round Robin with Dedicated Queue for each class of service

### Single Queue with Priority and Selective Preemption

Goals can also be achieved by applying priority sorting of service requests in the queue in single queue system. As the Figure 2.5, the server can drop selective requests as per algorithm decision and enqueue them again to service them later. This way, the service, that is to be serviced with prioritization gets services and thus the goal is achieved.

### Applying Bin Packing

Application of various Bin Packing Algorithm to arrange the service requests waiting to be serviced in the queue also has the potential to achieve the goal.

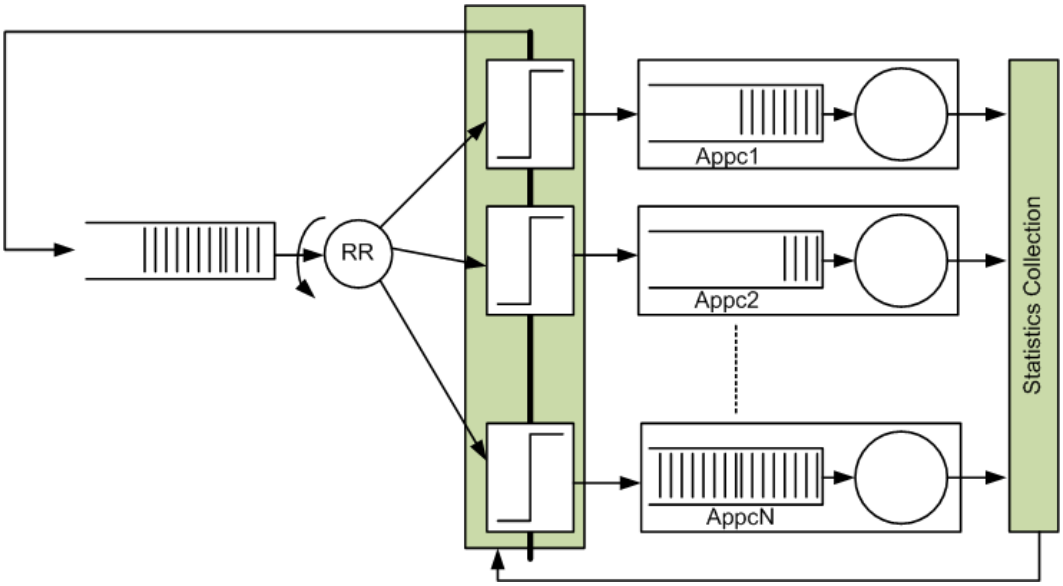


Figure 2.5: Single Queue with Priority and Selective Preemption

# Chapter 3

# Simulation and Result Analysis

Due to the nature of the problem, we don't have any deterministic proof that the proposed algorithm will be able to accommodate any arbitrary desired share of CPU. To verify the algorithm, we estimate the performance of the algorithm heuristically by developing a discrete event simulator in C. We will discuss in detail, the structure of the simulator in section 3.1.

In this research, we focused our analysis to answer the follows set of questions,

Q1. Does SDA/SAA "work" (i.e., can it meet the $P_m$ service differentiation goals)?

Q2. Is SDA/SAA indeed "better" than the other open-loop, static approaches (i.e., does it have the advantages described in section 1.3)?

Q3. How do algorithm parameters (i.e., $UT_m/DT_m$, $N$, $M$, $\{P_m\}$, initial $B_{nm}$ values) affect the behavior of SDA/SAA?

Q4. Is the Algorithm scalable (i.e., can meet the goal mentioned in Q1 immaterial of the number of appliances in the systems or number of eligible service domains)?

## 3.1 Discrete Event Simulator

As we mentioned earlier, there is no deterministic conclusion on the algorithm's behavior, we used simulator. The platform for development was "Microsoft Visual Studio Express Edition 2008" on Windows Vista Business. Matlab was not chosen as it is suited

more for discrete time simulation rather then event simulations. Matlab simulink provides nice interface for event simulations, however, is not as scalable and flexible as a custom built simulator. Network simulators (OMNET++, NS-2, OPNET) includes certain overhead because of their very type, simulation to the deepest detail of the network, which is irrelevant for the simulation. A tailor made simulator was found to be the most appropriate, as it gives most control, more flexibility while not loosing much of user friendliness. C was chosen based solely on author's choice.

The simulation model is depicted in Figure 3.1. The service requests arrive at the system in a random fashion. The gateway arrival process for service domain $m$ is modeled for simplicity as a Poisson process[1] with arrival rate $\lambda_m$. The CPU service time for requests from domain $m$ is a uniform random variable with average value $ES_m$. For simplicity, all appliances are considered homogeneous. They employ a single, infinite-capacity FIFO buffer for all domains activated in them; their CPU capacity is normalized to 1 unit. Therefore, the CPU utilization of (and thus the CPU allocation to) a service domain $m$ would be $\lambda_m \cdot ES_m$.

### 3.1.1 Event Generation

As mentioned before, the service request arrival at the gateway is modeled as a poisson process with arrival rate $\lambda_m$. The CPU service time for requests from domain $m$ is a uniform random variable with average value $ES_m$.

All the simulations can be categorized into two broad class of experiments, as we discuss the observations.

#### Infinite Supply of Service Requests

With infinite supply of service requests, the gateway can provide the appliance cluster, with the service requests it wants to process as per the algorithm. This configuration assumes that, there is always enough request waiting for each service domain that the algorithm meets the goal. For this, a poisson process is design with arrival rate of $\lambda \times m$, where the arrive request assigned with m with uniform probability that $\lambda_m = \frac{\lambda}{m}$. The service

[1]Since the system is controlled in a closed-loop fashion, the nature of the randomness in the arrival (and service time process) is not that critical.

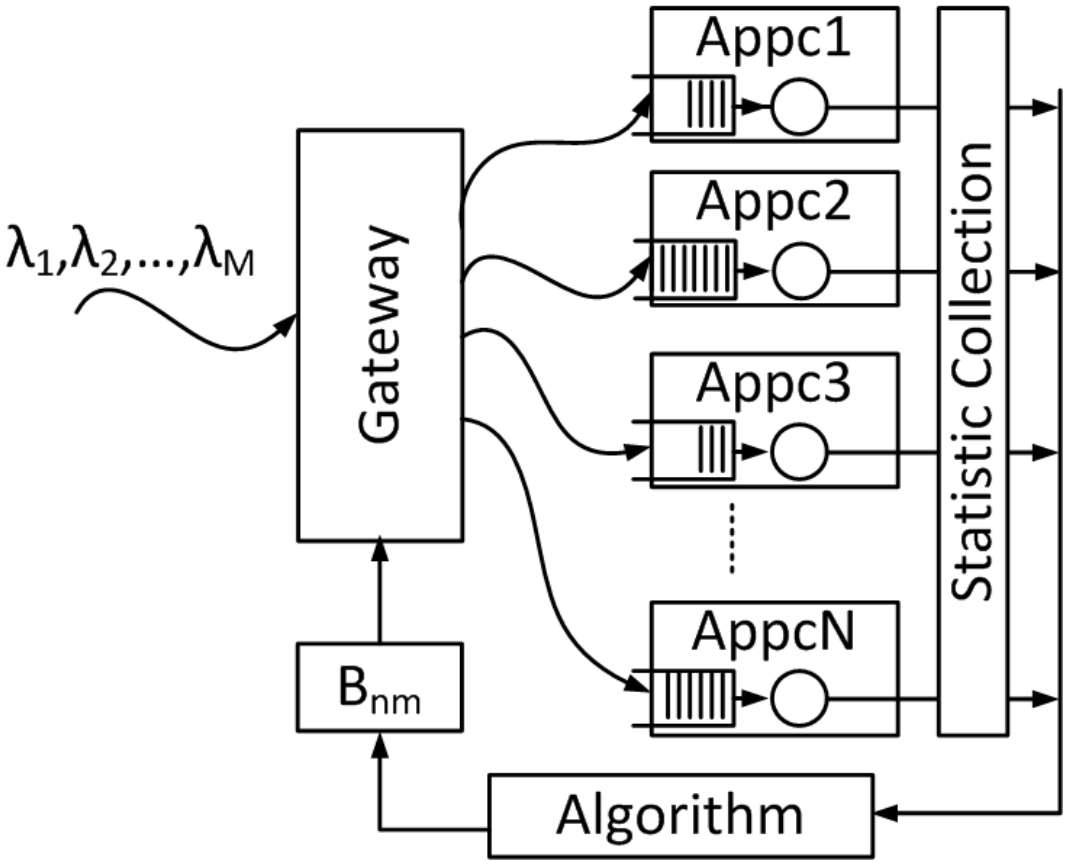


Figure 3.1: Simulator Design

request time is proportionate to the amount of time the appliance will take to process the XML part of the requests, follows uniform distribution with mean $ES_m$.

**Finite Supply of Service Requests**

This configuration is to simulate the scenario, when there is not enough service request arriving for any of the $M$ service domains. It will be very interesting to observe the behavior of the algorithm due to starving. For this, $M$ poisson processes are designed with arrival rate of $\lambda_m$, where $\lambda_m$ value for different $m$ can be chosen independently of $B_{nm}$ or any other dependency. The service request time is, as the previous scenario, proportionate to the amount of time the appliance will take to process the XML part of the requests, follows uniform distribution with mean $ES_m$. In this scenario, too simulate finite supply,

we have added dedicated buffer in the gateway for each service domain.

### 3.1.2 Event Simulation

The gateway distributes service requests to different appliances. This distribution is stochastic and follows a distribution in coherent with $B_{nm}$. If the queue in the appliance is empty, the service requests get it's CPU resource right away. All the appliances are considered identical, for the sake of simplicity of analysis. Each of the appliance deploys a single, infinite-capacity FIFO buffer for all domains activated in them; their CPU capacity is normalized to 1 unit. Therefore, the CPU utilization of (and thus the CPU allocation to) a service domain $m$ would be $\lambda_m \cdot ES_m$.

As a request gets served by the appliance, it leaves the system and update the statistics, which is used later to drive the algorithm.

Algorithm executes periodically and update $B_{nm}$ matrix (changing the number of activated instances for service domains in appliances). If service request is being processed during the algorithm execution, the simulator can take partial processed part of the request into account and keep the residual part for later execution.

## 3.2 Algorithm Analysis via Simulation Result

In our research, in order to answer the set of research questions asked in 3, we design following set of experiments.

**Observation 1,2 and 4** In order to answer question **Q1** (does the algorithm work?), we checked whether the algorithm works, with different number of services ($M$) running in different numbers of appliances ($N$) in cluster once it reaches steady state (roughly assumed to be 60,000 seconds, see page 29), by verifying the achieved $X_m$ for each service domain against $P_m$, whether they are within the tolerance $UT_m/DT_m$. This set of verification partially answers **Q1**.

**Observation 3** will answer the question **Q2**(is it better than open loop static allocation). We will see, how our algorithm behaves compare to manual static allocation described in section 1.3. In order to answer this question, we set up experiment with $M = 3$,

$N = 4$ and compared $P_m$ data against manual static allocation. Setting a tolerance to unreasonably higher value will replicate the scenario for manual static allocation.

**Observation 1,2,3,4,5** will address the question **Q3**(how does the algorithm parameter affect the system behavior). Observation 1 and 2 addresses, how the varying number of $N$ and $M$ affects the behaviors of the algorithm. Observation 3 will show the effect of threshold for tolerance $UT_m/DT_m$ in terms of oscillation. Observation 4 will demonstrate how extreme cases of goals ($\{P_m\}$) can destabilize the algorithm,. Observation 5, will clarify how the system administrators configuration ($B_{nm}$) based on erroneous anticipation of future traffic affects the algorithm behavior. Throughout experiment 3-5, simulation was done with $M = 3$, $N = 4$ and we observed $B_{nm}$ and $P_m$ to observe the effect.

**Observation 1,2** also address the question **Q4** (is it scalable?) As we vary the number of appliances $N$ from 1 to 1000 and observed whether the $P_m$ performance changes. Similar performance will lead us to a conclusion that the algorithm is scalable in a stand-alone cluster to a multiple appliance cluster. And by varying the number of service domains $M$, we can check how the algorithm behaves in terms of $P_m$ performance by supporting varying number of applications (i.e. service domains).

### 3.2.1 Observation 1:Varying the Number of Appliances, $N$

This observation suppose to answer **Q1**,**Q3** and **Q4** partially. **Q1** states the question as "does the algorithm work?". Which can be translated as "does the algorithm attain the target $P_m$ 'eventually'"? **Q3** enquires the behavior of algorithm (namely $B_{nm}$ and $P_m$) for variation in parameters. We demonstrate the effect of varying $N$ on the system in this observation. **Q4** interrogates the scalability of the algorithm. In this experiment we observe the system behavior with varying number of appliances and conclude whether it scales with different number of appliances.

Before we answer the question, let us discuss the simulation parameter setup. Here $M = 1$ will not allow any service differentiation which is the very goal of the algorithm. So, we set $M = 3$ ($M > 1$ suffices, for observation with different $M$, refer to section 3.2.2), the desired goals for each of the service domains are $\{P_m\} = \{43\%, 32\%, 21\%\}$ (effect of other choices of $P_m$ discussed in Section 3.2.4) with 1% threshold tolerances (effect of stricter

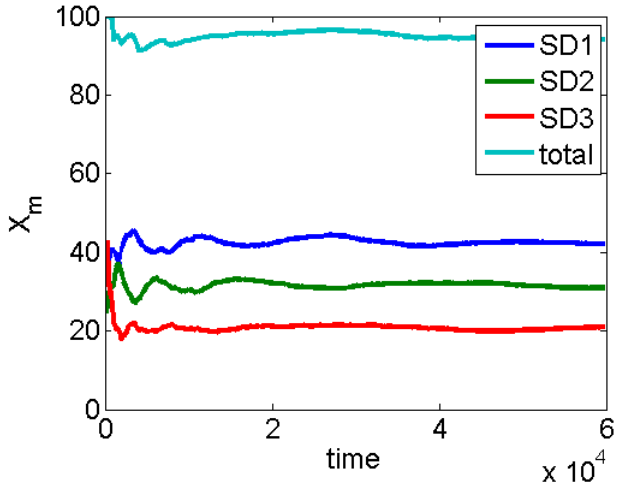


Figure 3.2: Observation 1: Variation of $X_m$ with N=1

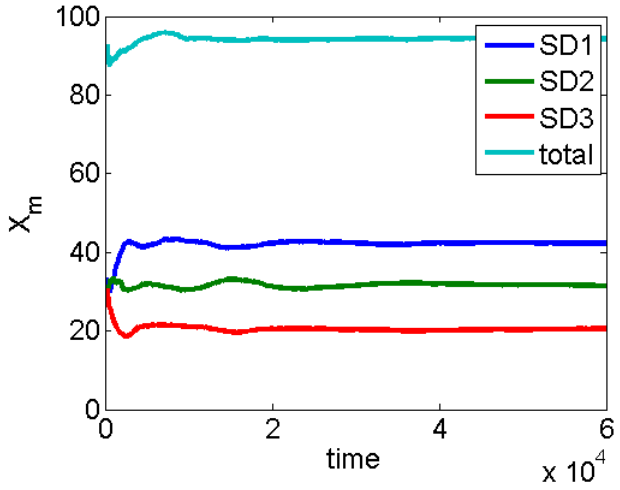


Figure 3.3: Observation 1: Variation of $X_m$ with N=4

or looser tolerance discussed in Section 3.2.3) up and down. All domains follow the same statistics for service time and arrival rate (uniform and poisson respectively). The initial instantiation matrix, the initial $B_{nm}$ initialized to same value for all the appliances, to create an initial "unfavorable" scenario (whatever the goal is, all service domain gets same preferential treatment in the begining) for the algorithm to overcome by throttling initial

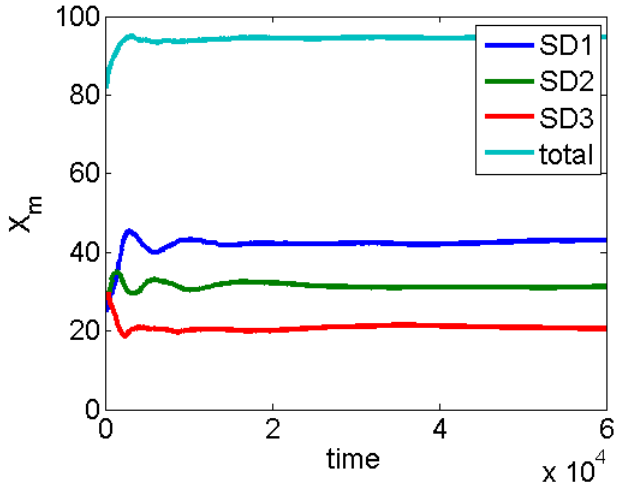


Figure 3.4: Observation 1: Variation of $X_m$ with N=10

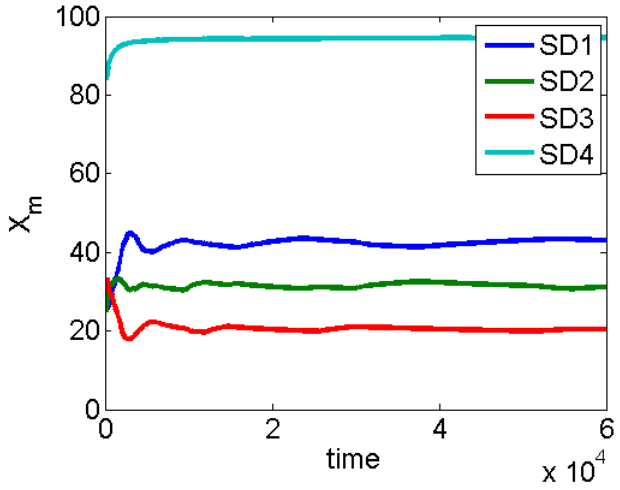


Figure 3.5: Observation 1: Variation of $X_m$ with N=100

arrivals to the system in oppose to the goal of 43:32:21 ( most favorable being $B_n m$ initialized that follows the ratio given in goal as [43:32:21] or roughly [4:3:2] for all n). In this set of simulation, the total arrival rate was chosen high enough that CPUs never run out of service requests waiting to be served, which in turn ensures the total utilization approaching 100%.

Fig. 3.2, 3.3, Fig. 3.4, Fig. 3.5 shows that SDA1/SAA1 algorithm meets the

desired goal as $X_m$ approaches $P_m$ within margin of tolerance ($X_1$ approaches $43 \pm 1\%$, $X_2$ approaches $32 \pm 1\%$ and $X_3$ approaches $21 \pm 1\%$ despite the initial unfavorable instantiation in the entire cluster (10 instances instantiated for all service domains in all appliances which instigates unfavorable allocation, $X_s$ of 33.33% for each service domains before the algorithm comes into play and changes $B_{nm}$ to qualify the incoming requests in order to meet the goal). This confirms, even if we do not have sufficient statistics to predict the traffic, the algorithm will try to modulate the incoming requests in order to achieve the goal, thus it partially answers **Q1**, yes the algorithm attains the target $P_m$ at time = 60000, with systems with different number of appliances $N$. However, we are yet to answer, whether time= 60,000 can be considered as 'eventually'.

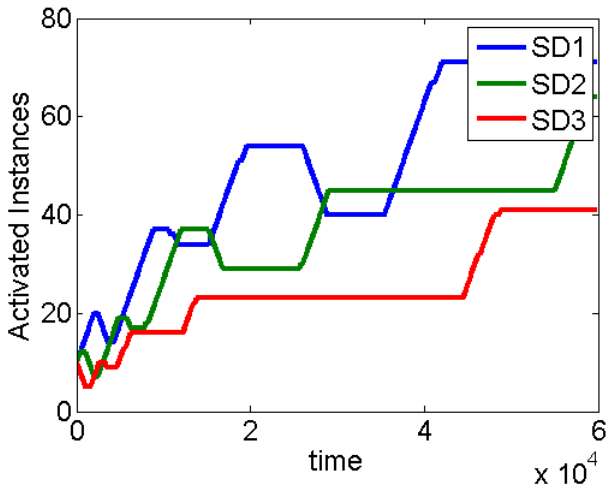


Figure 3.6: Observation 1: Variation of $B$ with N=1 (only appliance 1 shown)

In Fig. 3.6,3.7,3.8,3.9, we observe how the algorithm changes the number of instances (only for appliance 1, as we set the value of initial instantiation to be 10 for all, graphs for all other appliances are same). From the variation of $B_{nm}$ and $X_m$ as we see in Fig. 3.2, 3.3, 3.4, 3.5, 3.6,3.7,3.8,3.9, it can easily be seen that the algorithm causes oscillation in the beginning as for lower value of $k$, $X_m(k)$ changes abruptly which in turn causes oscillations in the values of $B_{mn}$. This oscillation is expected. Because, we take the time average of utilization to calculate $X_m(k)$. The bigger the value of $k$ is, the less affect $k^{th}$ measurement has on overall $X_m(k)$. For example, when time approaches 50,000,

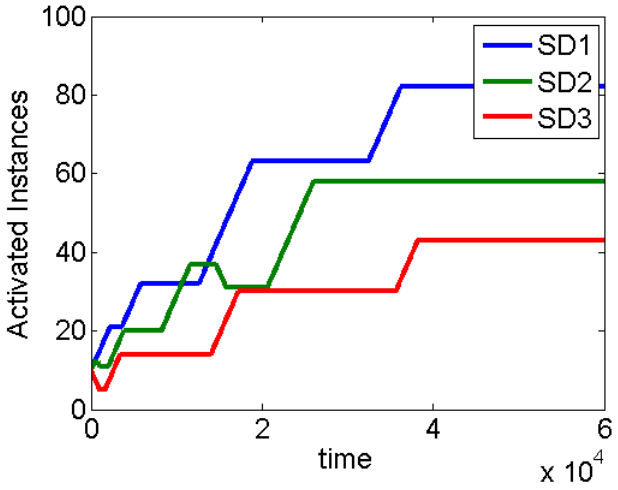


Figure 3.7: Observation 1: Variation of $B$ with N=4 (only appliance 1 shown)

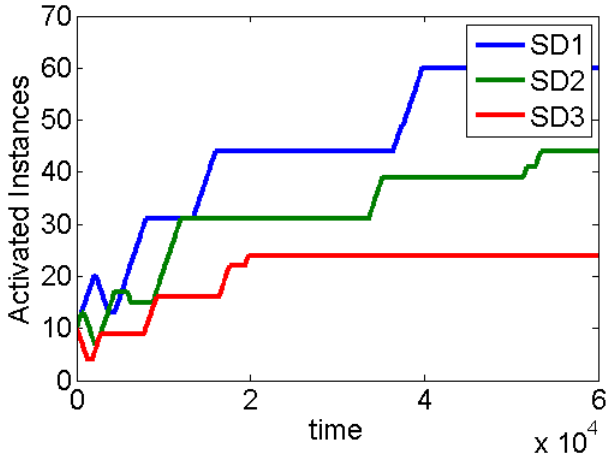


Figure 3.8: Observation 1: Variation of $B$ with N=10 (only appliance 1 shown)

the value of k is 500 (50000/100). So, this particular measurement would change $X_m(k)$ in extreme cases by 0.002% (when $X_m(k-1)$ is 100%) and this reduces to smaller values for every increment in K linearly thus having less and less affect on $X_m(k)$. In contrast, when the time was 200, and the value of k is 2, the $k^{th}$ measurement has the ability to change $X_m(k)$ by 50% in extreme cases ($X_m(k-1)$ is 100%). So, for k¿500, the new measurement will change $X_m(k)$ by less than 0.002%, thus if we accept 0.002% to be a negligible

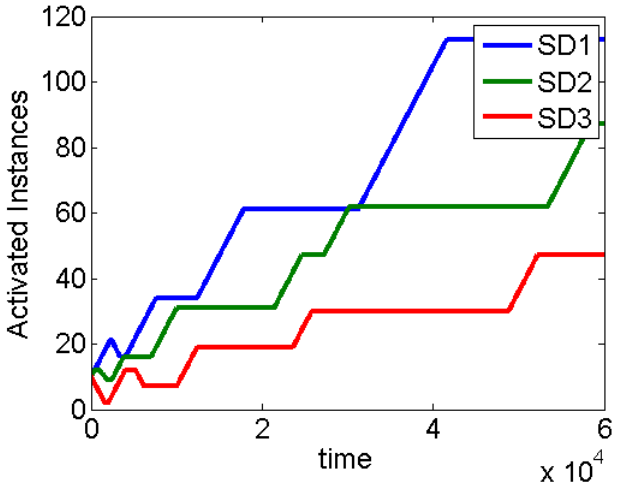


Figure 3.9: Observation 1: Variation of $B$ with N=100 (only appliance 1 shown)

small change, it can be assumed that at time¿50,000, the algorithm will have little effect on collected statistics and we can consider time¿50,000 (more accurately k¿500) where the algorithm converges, 'eventually'. So, yes the algorithm achieves the target $P_m$ with systems with different number of appliances $N$ 'eventually', thus we answer the part of **Q1** we wanted to answer in Observation 1.

From Fig. 3.2, 3.3, Fig. 3.4, Fig. 3.5 we see $X_m$ for all service domains demonstrates similar behavior and approaches $P_m$ within threshold for all service domain for $N = 1, 4, 10, 100$ which answers the question **Q3, Q4** partially. In answer to **Q3**, we assert that the number of appliances $N$ has little or no effect on algorithm as we see little or no variation in response of $X_m$ with varying number of $N$. As for above mentioned exemplarily cases we see the algorithm attains the target $P_m$, we can also answer **Q4**, the algorithm indeed scales with different numbers of appliances in the cluster. Increasing the number of appliances by two order of magnitude (from 1 to 100) gives us the confidence that algorithm will scale for higher order of magnitude as well.

### 3.2.2 Observation 2:Varying the Number of Service Domains, $M$

This observation helps us to obtain the partial answer for **Q1**,**Q3** and **Q4**. **Q1** states specifically "does the algorithm attain target $P_m$ 'eventually'"? The current observa-

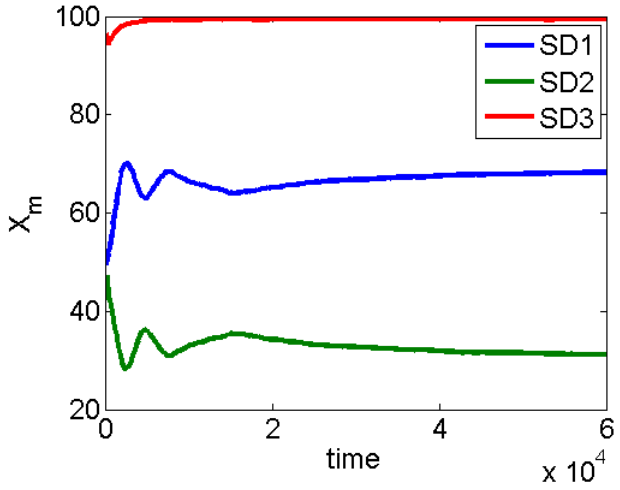


Figure 3.10: Observation 2: Variation of $X_m$ with M=2

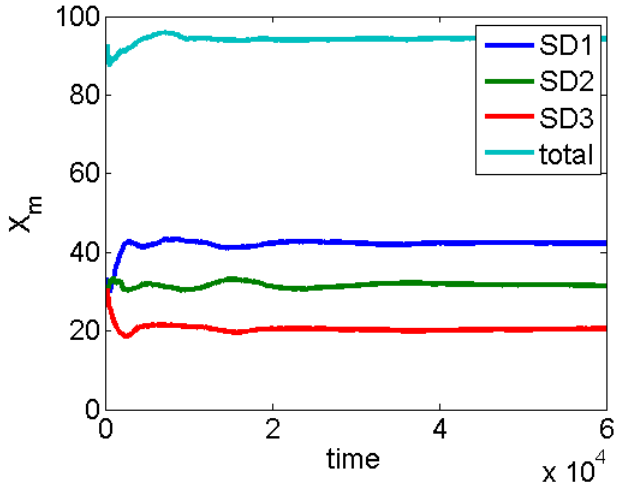


Figure 3.11: Observation 2: Variation of $X_m$ with M=3

tion checks, whether the algorithm can sustain different numbers of applications running in the system. **Q3** interrogates the behavior of algorithm (i.e. $B_{nm}$ and $P_m$) with different parameters. We demonstrate the effect of $M$ on the system in this observation. **Q4** questions the scalability of the algorithm. In this experiment we observe the system behavior with varying number of service domains and conclude whether it scales with different number of

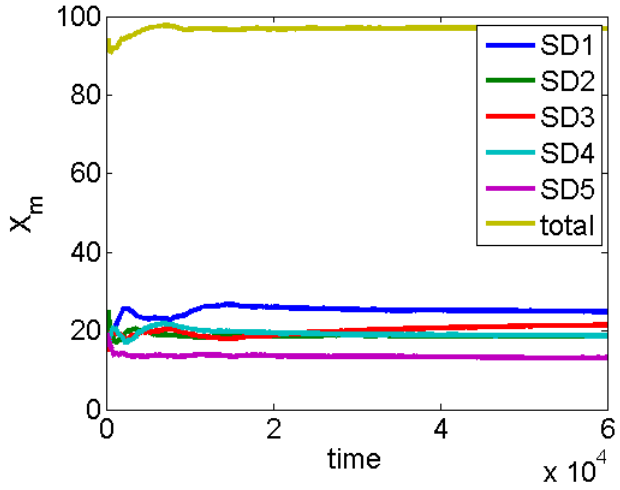


Figure 3.12: Observation 2: Variation of $X_m$ with M=5

service domains.

We set $N = 4$ (here any number of $N$ is applicable, as we observed the response of the algorithm in , $N$ does not affect the behavior of the algorithm), the number of appliances on which service domain requests will be preprocessed. For three experiment with $M = 2, 3, 5, 10$ (Any number of service domains,$M > 0$ is appropriate here, however $M > 10$ would make it hard to visualize the plots for the reader and $M = 1$ doesn't provide any service differentiation, which is the very goal of this algorithm), we set the target goals (chosen arbitarily, details are left for section 3.2.4) for service domains as $\{P_m\} = \{65\%\, 33\%\}$,$\{P_m\} = \{43\%, 32\%, 21\%\}$ and $\{P_m\} = \{24\%, 19\%, 19\%, 19\%, 14\%\}$ respectively with 1% threshold tolerance both ways. In this set of simulations, the total arrival rate was sufficient to keep the CPUs busy to achieve total utilization that approaches 100%

Fig. 3.10, 3.11, Fig. 3.12 shows that SDA1/SAA1 algorithm meets the desired goal despite the initial unfavorable instantiation (see page 28) in the entire cluster. This reconfirms, with observation 1, that the algorithm can shape the incoming service requests to achieve the goal. Fig. 3.10, 3.11, Fig. 3.12 shows that $X_m$ approaches $P_m$ within the tolerance for a system with two, three and five service domains activated.

In Fig. 3.13,3.14,3.15, we observe how the algorithm changes the number of in-

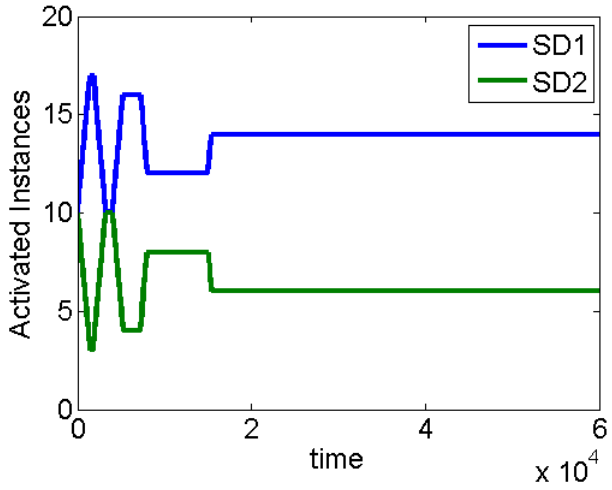


Figure 3.13: Observation 2: Variation of $B$ with M=2 (only appliance 1 shown)

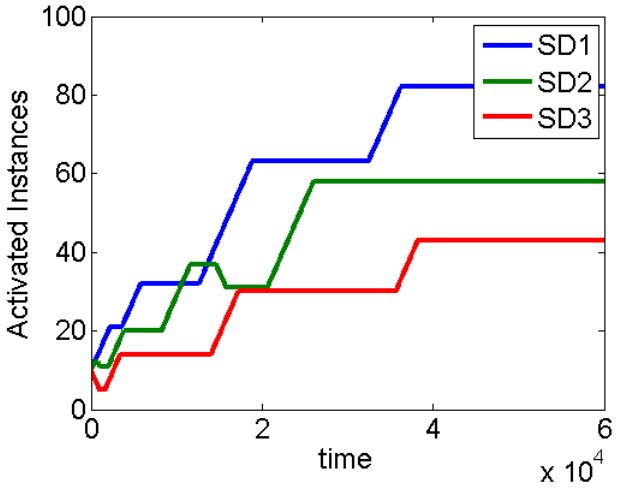


Figure 3.14: Observation 2: Variation of $B$ with M=3 (only appliance 1 shown)

stances (only for appliance 1, as we set the value of initial instantiation to be same, other graphs are same). The algorithm causes oscillation in the beginning as for lower value of $k$, $X_m(k)$ changes abruptly which in turn causes oscillations in the values of $B_{mn}$.

As Fig. 3.10, 3.11, Fig. 3.12 shows that $X_m$ approaches $P_m$ within the tolerance for a system with two, three and five service domains activated, we have the answer for **Q1**

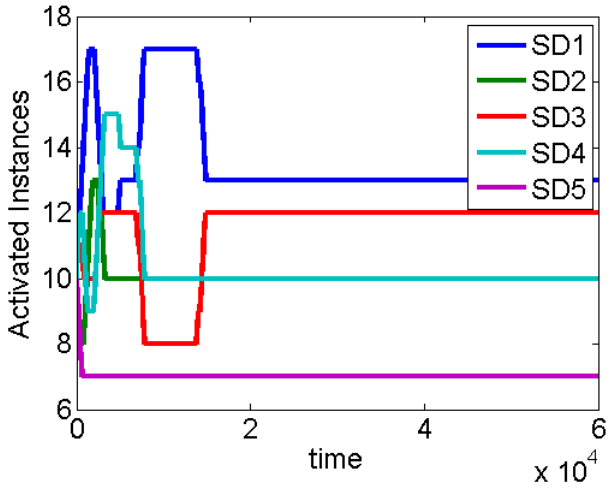


Figure 3.15: Observation 2: Variation of $B$ with M=5 (only appliance 1 shown)

partially, the algorithm works with varying number of service domains. As we see for varying number of service domains, the algorithm reaches the target $P_m$ "eventually", we can answer **Q3** as there is little or no effect of the number of service domains $M$. However, at this point, we can fully answer **Q4**, yes, the algorithm response to the required $P_m$, irrespective of the number of appliances (see section 3.2.2) and service domains, or alternatively, the algorithm scales for different number of appliances and service domains.

### 3.2.3 Observation 3:Varying the threshold tolerance,$UT_m/DT_m$

In this observation, we study the effect of threshold tolerance $UT_m/DT_m$, which helps us to answer **Q2**(is the algorithm better than open loop static allocation?) and **Q3** (how does the algorithm reacts with different threshold values?). To answer **Q2**, we selected a very high threshold tolerance (100%), that $B_{nm}$ never gets updated that it replicate the scenario of open loop static allocation. To answer **Q3**, we observed change in pattern for $B_{nm}$ and $P_m$ with different threshold tolerance.

While to answer **Q2**, we set the tolerance to a ridiculously large value (e.g. 75%, 100%), the algorithm behaves like a open loop static allocation algorithm. In Fig. 3.24, we see the static allocation doesn't throttle incoming requests by activating and deactivating instances (see Fig. 3.23) the incoming requests to achieve the goal. We will also see

in 3.2.6 that, the algorithm doesn't utilize the white space in case of starvation. From this observation we see, the proposed algorithm is responsive to the target demand and shapes the incoming traffic accordingly, in contrast to manual static allocation. And also, proposed algorithm utilizes the white space. Here, we have the conclusive answer for **Q2**, proposed algorithm is better than open loop static manual allocation algorithm in terms of responsiveness and utilizing unused white space.

To answer **Q3**, we carried out two experiment with $B_{nm} = 30, \forall n, m$ and $\{P_m\} = \{43\%, 32\%, 21\%\}$. For the first simulation to present the effect of strict tolerance, threshold was chosen as $UT_m = DT_m = 0.1\%$, while for the second simulation, to present the effect of looser tolerance, we chose $UT_m = DT_m = 2\%$. A larger tolerance (i.e. $UT_m = DT_m = 10$ or larger) was not tested to answer **Q3**, because of the irrelevancy. As the algorithm can satisfy the goals with very strict tolerance of $UT_m = DT_m = 0.1$,it can be inducted that loose tolerance will easily satisfy the goal. But, it is to be notified, the algorithm considers the target met for $P_m - DT_m \ P_m + UT_m$. Larger tolerance (say 10%) with target goal of $\{P_m\} = \{44\%, 33\%, 22\%\}$ would mislead the algorithm that it will accept $\{34\%, 33\%, 32\%\}$ as a target satisfied scenario.

In Fig. 3.16,3.17 demonstrates algorithm behaviors for strict tolerance. While, Fig. 3.18,3.19 demonstrates for flexible tolerance. The most evident effect of tolerance parameters is "oscillatory" behavior as depicted in Fig. 3.17 which is caused by a rather strict setting of $UT_m = DT_m = 0.1\%$ for all domains. In general, stricter tolerance causes more oscillation in both the goals and instancing matrix values. For a lower value of $B_{nm}$ (e.g. 5) and $UT_m = DT_m = 0.1\%$, one interesting phenomenon we can see in Fig. 3.20. Very strict tolerance may causes one service domain to loose all it's activated instances, while setting with a less restricted tolerance is less prone to this. In Fig. 3.22, we see combination very strict tolerance and target CPU percentage (approaches to 100%) has similar effect. While Fig. 3.19 shows, with same $P_m$, with less strict tolerance demonstrates more stable behavior.

In a nutshell, the stricter tolerance will cause oscillation in $P_m$ and $B_{nm}$, which is basically increasing the overhead of the algorithm. However, stricter tolerance will result in more accurate target attainment (as $X_m \approx P_m$ for this case). For loose tolerance, we have a more stable algorithm, while sacrifice is made in terms of accuracy ($X_m - P_m$). With very low number of activated instances ($B_{nm}$, the stricter tolerance will exhaust all the activated

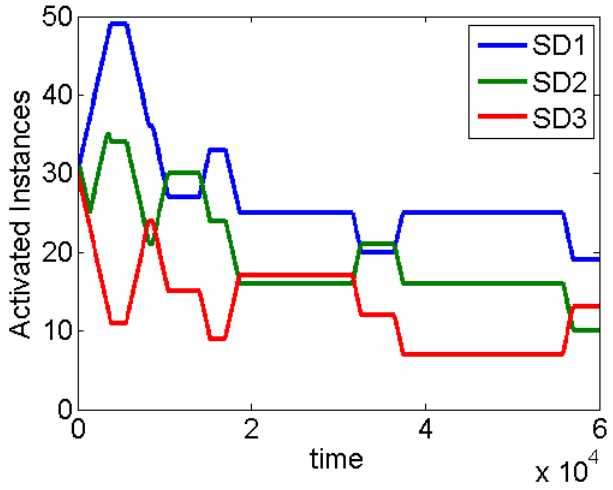


Figure 3.16: Observation 3: Variation of $B$ with $UT_m = DT_m = 0.1\%$

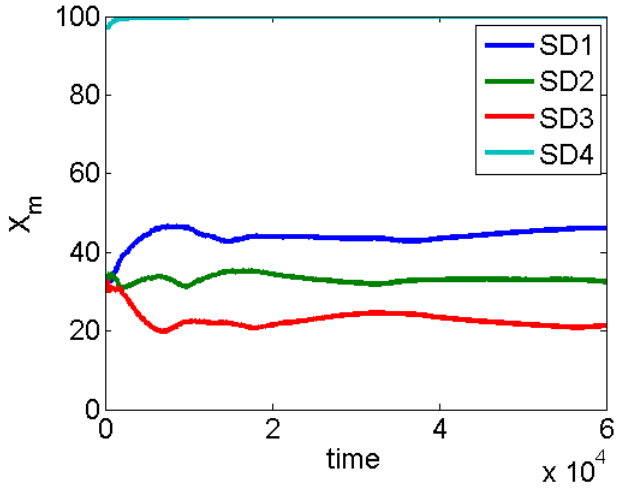


Figure 3.17: Observation 3: Variation of $X_m$ with $UT_m = DT_m = 0.1\%$

instances for overachieving service domains and that service domain, may never be able to achieve the goal as described in $P_m$. While looser tolerance, most of the cases results in low oscillation, thus less overhead and less prone to the exhaustive deactivation. So, we can address **Q3**, the effect of tolerance on algorithm as, stringent tolerance give us more precise target achievement, as a trade off, we have to sacrifice stability.

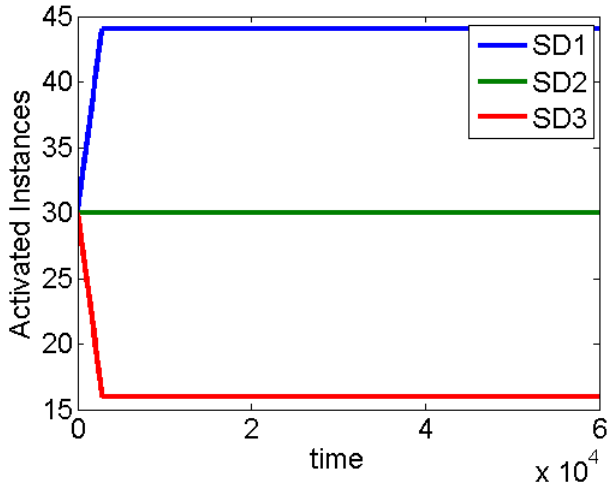


Figure 3.18: Observation 3: Variation of $B$ with $UT_m = DT_m = 2\%$

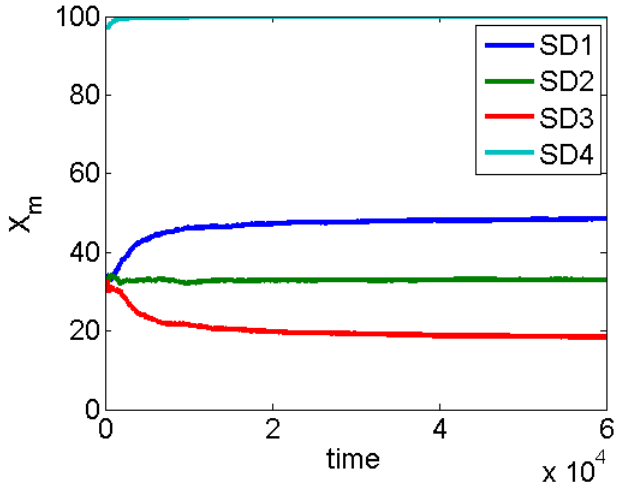


Figure 3.19: Observation 3: Variation of $X_m$ with $UT_m = DT_m = 2\%$

### 3.2.4 Observation 4:Varying the target percentage, $P_m$

In this observation, we experiment whether the algorithm can satisfy goals with different target CPU utilizations. Here, we try to answer **Q1**, whether the algorithm meats the goal eventually for any goal we set. And the experiments designed to answer **Q1** partially, also helps us to address **Q3**, the effect of $P_m$ on the algorithm behavior.

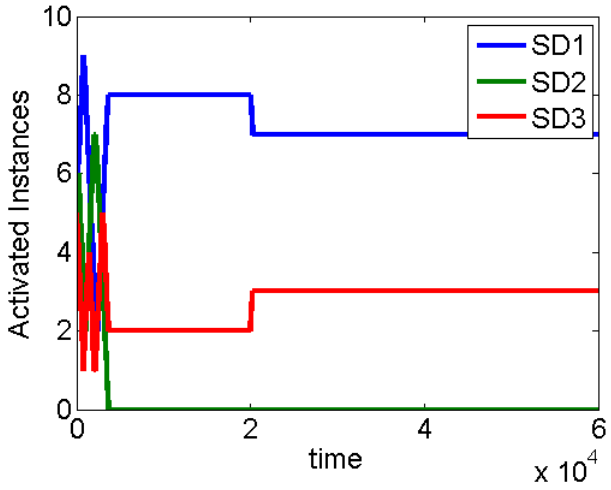


Figure 3.20: Observation 3: Variation of $X_m$ with $UT_m = DT_m = 0.1\%$ with $B_{nm}$=5

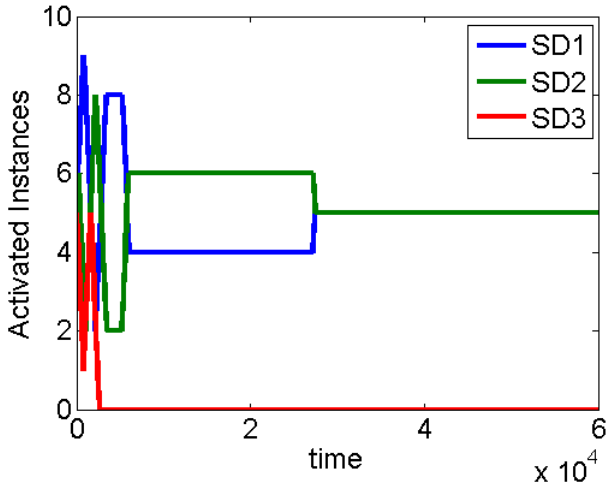


Figure 3.21: Observation 3: Variation of $B$ with $UT_m = DT_m = 0.1\%$

We can broadly classify $P_m$ allocation as equal preference and unequal preference. For equal preference, with $M = 3$ (throughout this observation we used $M = 3$ and $N = 4$), with 1% threshold tolerance both ways, $\{P_m\} = \{32\%, 32\%, 32\%\}$ is a very reasonable choice among few available options (e.g. $\{P_m\} = \{33\%, 33\%, 33\%\}$). However, for unequal preference, we have numerous choices. But, we can roughly classify those choices into three

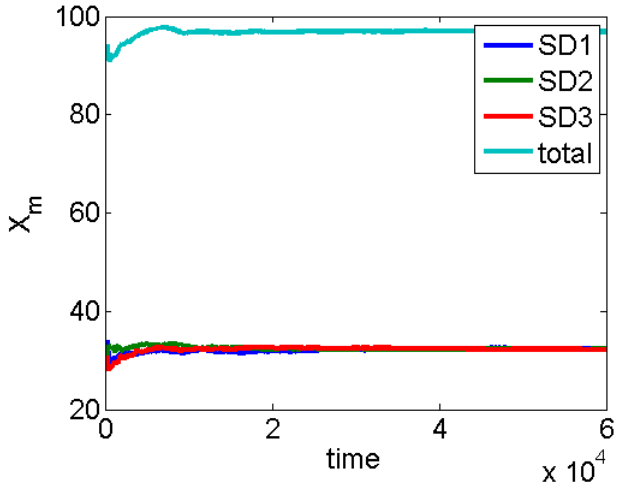


Figure 3.22: Observation 3: Variation of $X_m$ with $UT_m = DT_m = 0.1\%$

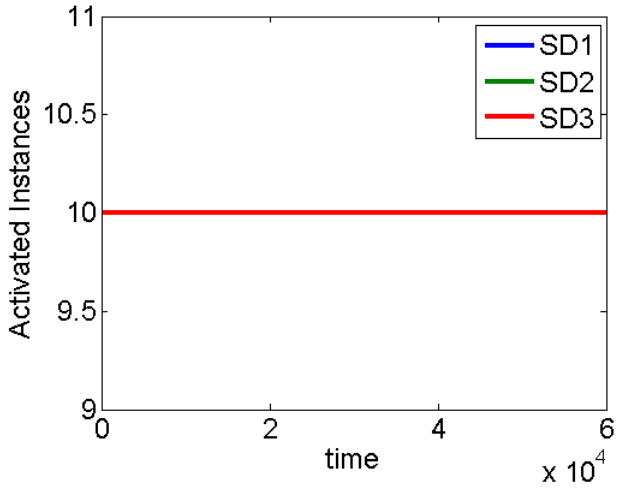


Figure 3.23: Observation 3: Variation of $B$ with $UT_m = DT_m = 100\%$

groups. One contains the cases where one service domain has significantly higher target CPU utilization than the others, the second consists of cases where one service domain have very little demand in terms of CPU utilization compare to the other service domains. If the algorithm can satisfy these two extreme cases, then we can have confidence that the algorithm will be able to perform in any other case in between which is exemplified by

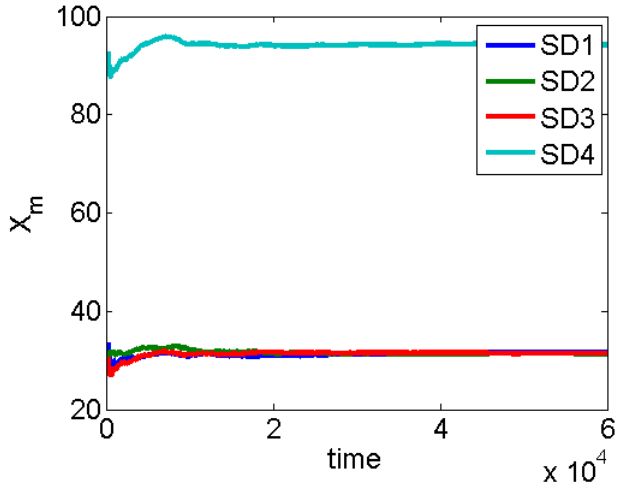


Figure 3.24: Observation 3: Variation of $X_m$ with $UT_m = DT_m = 100\%$

the third class, where we do not emphasize or deemphasize one service domain compare to others.

We carried out experiment with $\{P_m\} = \{43\%, 32\%, 21\%\}$ (unequal preference) ,$\{P_m\} = \{32\%, 32\%, 32\%\}$ (equal preference to the service domains) ,$\{P_m\} = \{90\%, 3\%, 3\%\}$ (very high preference to one of the service domain) and $\{P_m\} = \{47\%, 47\%, 3\%\}$ (very low preference to one of the service domain) with 1% threshold tolerance both ways. Fig. 3.25, 3.26,3.27,3.28 demonstrates that, the algorithm adjusts $B_{nm}$ to honor the goal (as they depicts $X_m \rightarrow P_m$), with different target $P_m$.

Using the observation depicted in Fig. 3.25, 3.26,3.27,3.28 we can answer **Q1**, whatever target CPU utilization the system administrator chooses, the algorithm honors it. Now we are in a position, to response a qualified answer to **Q1** that, if the system has enough incoming request to meet the goal of reaching $P_m$ (share of CPU), it will approach $P_m$. An interesting result will be to see how the system behaves if it doesn't receive enough request to achieve the goal, which is discussed in 3.2.6. Using the same observation, we also address **Q3**. The algorithm's performance in terms of attaining the goal, does not depend on the target goal we set, it invariably meets the goal whatever $P_m$ we set.

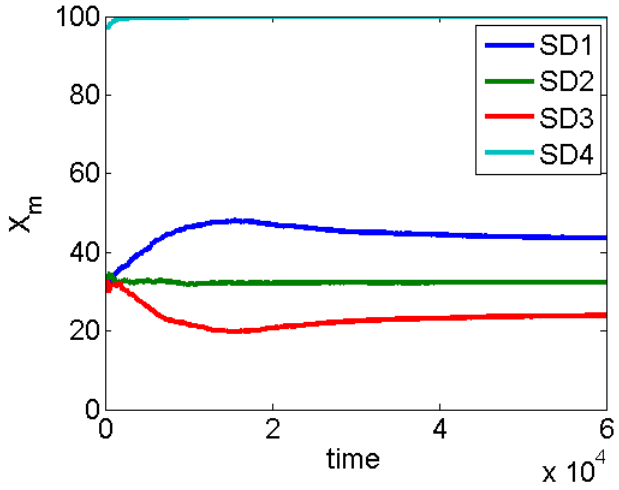


Figure 3.25: Observation 4: Variation of $X_m$ with $\{P_m\} = \{43\%, 32\%, 21\%\}$

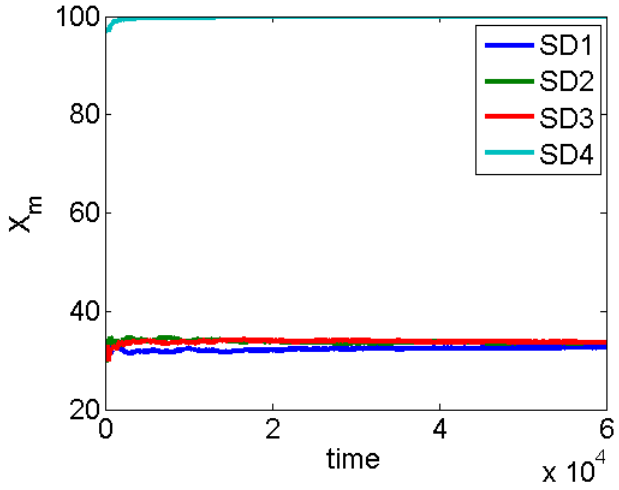


Figure 3.26: Observation 4: Variation of $X_m$ with $\{P_m\} = \{32\%, 32\%, 32\%\}$

### 3.2.5 Observation 5:Varying Instantiation Matrix, $B_{nm}$

In order to answer part of **Q3**, the effect of initial instantiation of service instances in the appliances on the algorithm behavior, we carry out the following experiment.

In the experiment, with $N = 3, M = 4, UT_m = DT_m = 1\%$ and $P_m = \{43\%, 32\%, 21\%\}$ we see for a higher number of instantiation of $B_{nm} = 10, \forall n, m$ and a lower number of in-

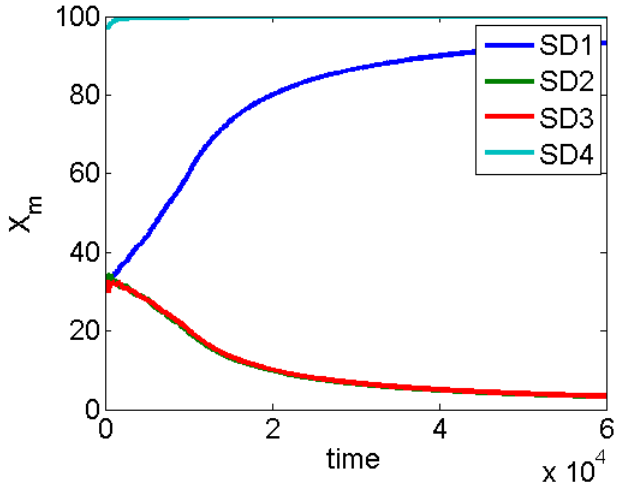


Figure 3.27: Observation 4: Variation of $X_m$ with $\{P_m\} = \{90\%, 3\%, 3\%\}$

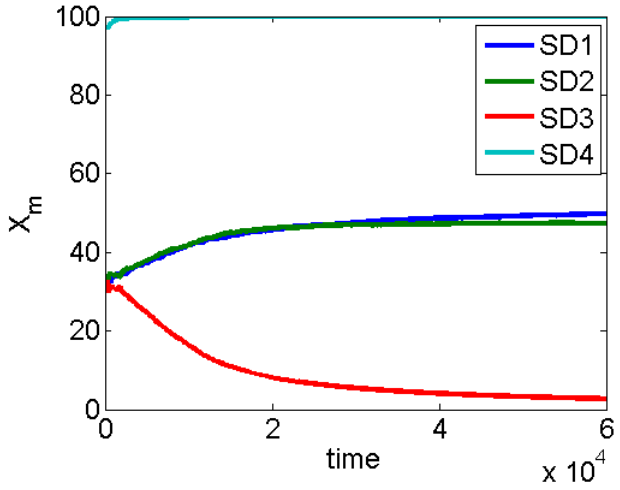


Figure 3.28: Observation 4: Variation of $X_m$ with $\{P_m\} = \{47\%, 47\%, 3\%\}$

stantiation of $B_{nm} = 10, \forall n, m$ to check whether the algorithm exhausts all the activated instances thus preventing processing of some of the service domains.

Fig. 3.29,3.31,3.30,3.32 demonstrates the behavioral difference of the algorithm for the simulation set up we discussed above. We see in Fig. 3.31 that the algorithm achieves the target as the algorithm cannot deactivate all the instances of service domain 3. While

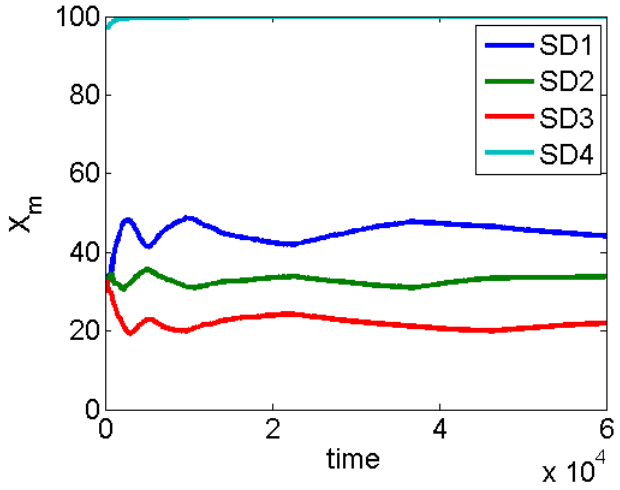


Figure 3.29: Observation 5: Variation of $X_m$ with $B_{nm} = 10$

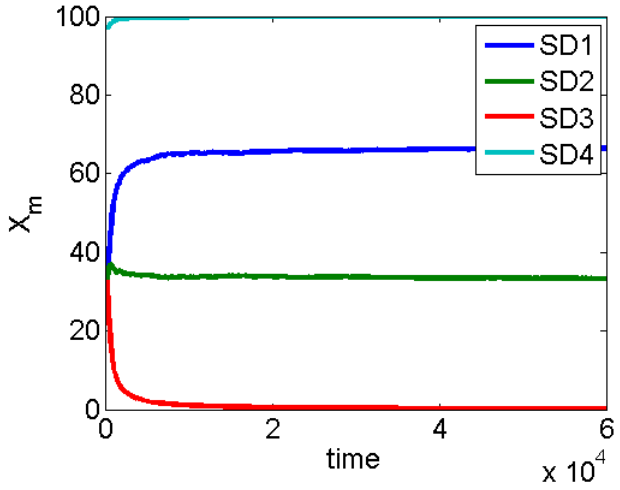


Figure 3.30: Observation 5: Variation of $X_m$ with $B_{nm} = 2$

for $B_{nm} = 2, \forall n, m$ initial burst of traffic for service domain 3 causes overachieving in the beginning, which in turn instigate the algorithm to deactivate instances for service domain 3 as clearly seen in Fig. 3.32. As the algorithm cannot activate instances for service domain 3 after all of them gets deactivated, algorithm fails to meet target for service domain 3.

This observation, answers how initial value of $B_{nm}$ affects the algorithm, which is

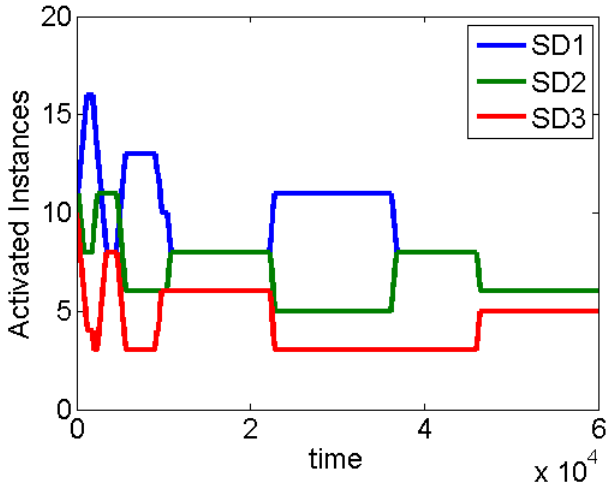


Figure 3.31: Observation 5: Variation of $B$ with $B_{nm} = 10$

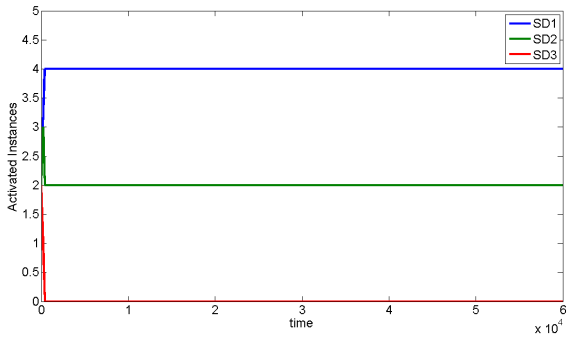


Figure 3.32: Observation 5: Variation of $B$ with $B_{nm} = 2$

part of the questions asked in **Q3**. Starting with a low $B_{nm}$, the algorithm deactivates all the available instances for the overachieving service domain and fails to meet the target at the end, while having a higher $B_{nm}$ that it can sustain the initial surge of deactivation, the algorithm shows more stability and achieves the goal eventually.

### 3.2.6 Observation 6:Varying the Rate of Arrival

So far, in all the observations discussed, we assumed that the arrival of service requests was high enough that the system receives enough service requests to attain the

goal. Here we will discuss, what if the system doesn't receive enough request for one particular service domain. Using Observation 6, by controlling the arrival rate for service domains, we answer **Q3**, how does arrival rate for a service domain affects the algorithm behavior. As we try to answer **Q3**, we also find a case, where we have a more complete answer for **Q1**.

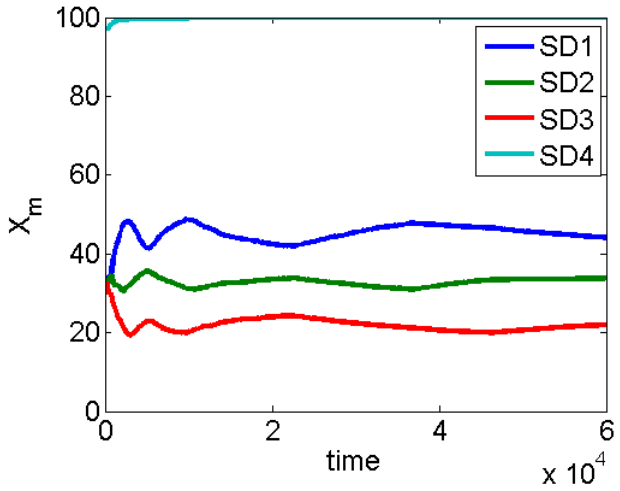


Figure 3.33: Observation 6: Variation of $X_m$ with Sufficient Service Request Arrival

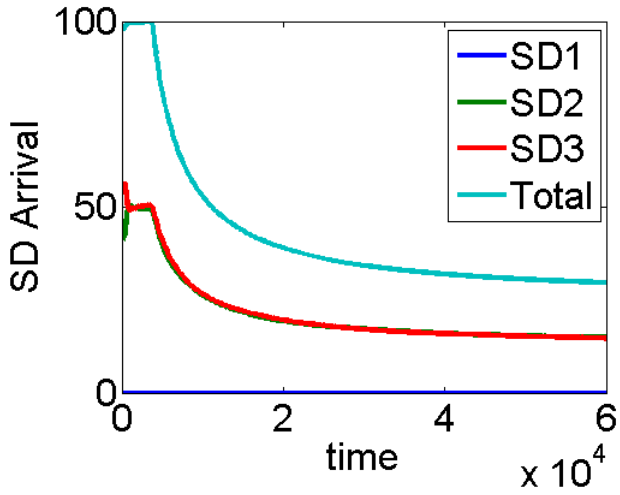


Figure 3.34: Observation 6: Variation of $X_m$ with insufficient Service Request Arrival

The system administrator can set the $P_m$ parameter can be set in one of two possible ways: "achievable" or "non-achievable". In this observation, we set the arrival rate $\lambda_m$ and average service times $ES_m$ of the domain are such that $\lambda_m \cdot ES_m \geq P_m$; in other words, there is enough traffic to take advantage of the allocated CPU resource, in the first case, the "achievable". In the second ("non-achievable"), $\lambda_m \cdot ES_m < P_m$, which simulates the case where the domain does not have enough traffic to take advantage of the allocated CPU resource. As with all feedback-based algorithms, this situation may "mislead" the algorithm into always activating additional instances of the domain, causing "instability" and eventually affecting other domains too.

Figure 3.33 and all the observations made in Observation 1 through 5, demonstrates the first scenario, where $\lambda_m \cdot ES_m \geq P_m$.

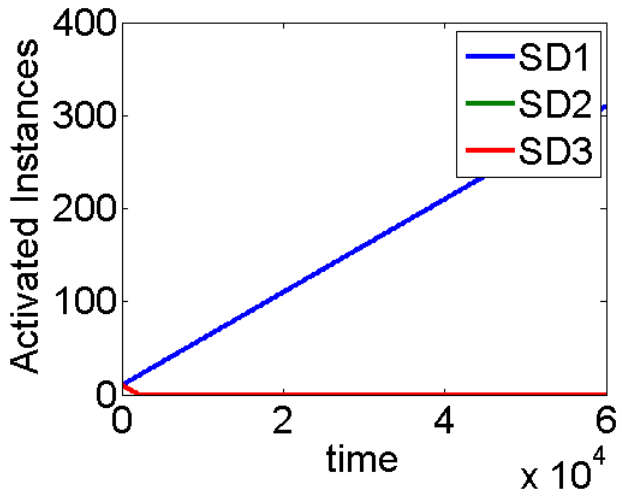


Figure 3.35: Observation 6: Variation of $B$ with insufficient Service Request Arrival

Figure 3.34 exemplifies what can happen when "non-achievable" goals are set. In this experiment, we set again $\{P_1, P_2, P_3\} = \{43\%, 32\%, 21\%\}$. The arrival rate for SD1 was set low, so that this domain would never reach a 43% CPU utilization, even if it was given full access of the CPUs; its maximum utilization will eventually approach $\lambda_1 \cdot ES_1 \approx 4\%$ in this experiment. As the other two domains, over achieves in the beginning, all the activated instances gets deactivated, and total utilization of the system eventually goes down. To utilize the white space, which may be of interest for the system administrator, we propose a

modification to the algorithm in section 3.3.2. Figure 3.35 depicts how starvation can cause adverse affect on $B_{nm}$. So, arrival rate that doesn't provide system with enough incoming request cannot achieve the goal and destabilize the system by continual activation of $B_{nm}$, while if the system is provided with sufficient service requests that the system can achieve the goal, the algorithm behaves as per expectation. Here, we answer **Q3** that desired $P_m$ goals depend heavily on the actual arrival rates.

At this point, we came across another insight to **Q1**, by saying, as long as we have sufficient service request coming in to the system to meet the goal, the algorithm works.

## 3.3 Suggested Modification in Algorithm

Based on above observations, we suggest two modifications to the original algorithm (SDA1/SAA1). In the first modification, we restrict the number of activation during execution to be less or equal to the number of deactivation carried out in the same execution. We hope, with this modification, the number of activated instances cannot go indefinitely that causes thrashing eventually. In the second modification, we restrict the algorithm to deactivate when the number of activated instance for one service domain in one appliance is more than one. Here, our expectation is, as the number of activated instances will not go down to zero, it will be able to sustain higher degree of opposably biased traffic.

### 3.3.1 Modification 1: Restricting the algorithm to activate equal or less than the number of deactivations it carried out in the same execution (SDA2/SAA2)

We have seen, because of lower arrival rate (section 3.2.6) and stringent target setting (section 3.2.4 and 3.2.3), sometimes the algorithm keep on activating instances indefinitely, which causes thrashing. To eradicate this issue, we propose a modification which limits the number of activation by number of deactivation carried out in the same algorithm execution instance. This modification will ensure that total number of activated instances never goes beyond the number of instances the system started with.

For the results depicted in Fig. 3.36, we have$N = 4$, $M = 3$, with goal $\{P_m\} = \{44\%, 33\%, 22\%\}$ including 2% up and down threshold tolerances.The instantiation matrix was initialized with same values in all four appliances to create an "unfavorable" (see page

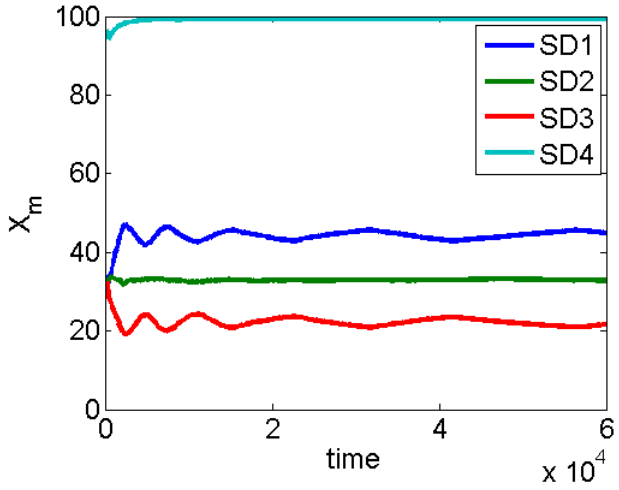


Figure 3.36: Modification 1: Utilization $X_m(T_k)$ vs time.

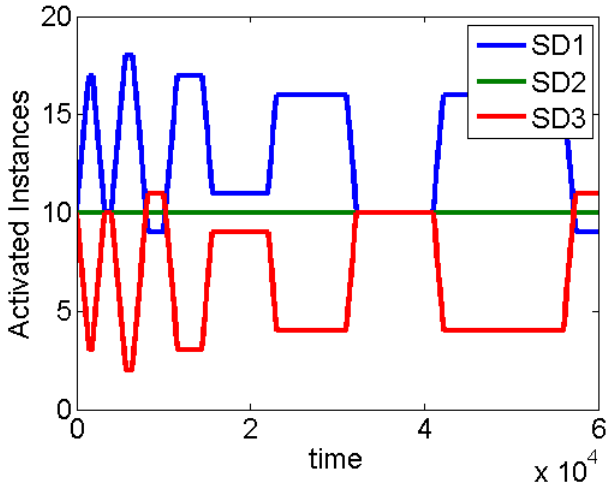


Figure 3.37: Modification 1: Variation in $B_{nm}$ values, appliance 1.

28) situation for the algorithm, the number of instances initialized were 10 for all three service domains. $SDA2/SAA2$ algorithm meets the desired goal despite the unfavorable initial instantiation as shown in Fig. 3.36. In this simulation, the total arrival rate was chosen high enough to keep the CPUs busy, hence the total utilization, as shown in Fig. 3.36 approaches 100%, which also presumably answer our question **Q1**.

The algorithm alters the number of instances of service domains in the appliances to achieve the goal as seen in Fig. 3.37. For lower value of $k$, $X_m(k)$ (explanation given in page 29) changes abruptly which in turn causes oscillations in the values of $B_{mn}$.

To answer **Q2**, we found desired $P_m$ goals depend on the actual arrival rates. For example, with $\{P_1, P_2, P_3\} = \{44\%, 33\%, 22\%\}$ and the arrival rates and the average service times for the three service domains are equal. Manual static allocation, would allocate CPU times in the ratios $44\% : 33\% : 22\%$, wasting 11% for SD1, depriving SD3 of 33-22=11% and leaving a 22% "white space" (unused CPU resource). Figure 3.36 shows how the algorithm eliminates the white space altogether.

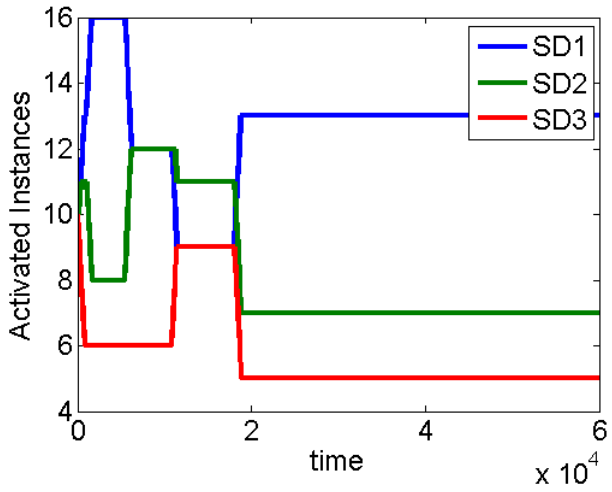


Figure 3.38: Modification 1: Variation in $B_{nm}$ Values, in Appliance 1, for $N = 1$

Effect of variation in the algorithm parameters $N, M, UT_m/DT_m, P_m$, initial $B_{nm}$ asked in **Q3**. With this modification, algorithm didn't change it's behavior (i.e., the nature of variations in the $B_{nm}$ values) as well as its performance (i.e., the achieved percentages) did not change with variation in the number of appliances $N$ or the number of service domains $M$ (as discussed in section 3.2.2 and 3.2.2). Figure. 3.38, 3.39, 3.40 and 3.41 shows some results. As expected, the result is in coherent with the result shown in Fig. 3.36. So, here we also have the answer for the scalability questions asked in Question **Q4**.

The effect of the tolerance parameters $UT_m/DT_m$, as we stated before, causes oscillation as Fig. 3.43. For comparison we can check Fig. 3.42 to Fig. 3.36 and Fig.

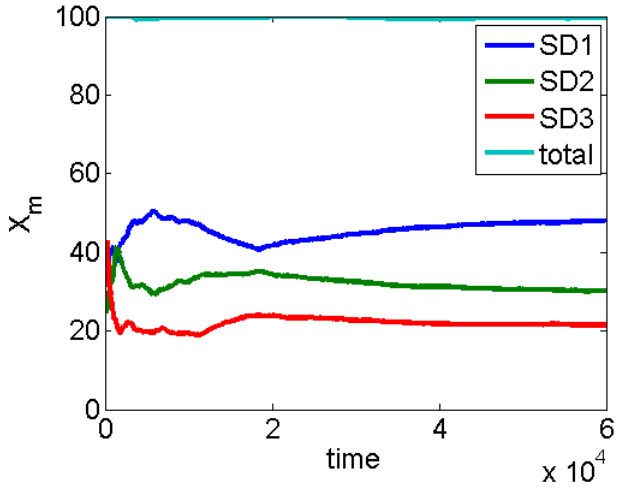


Figure 3.39: Modification 1: Utilization $X_m(T_k)$ vs time, in Appliance 1, for $N = 1$

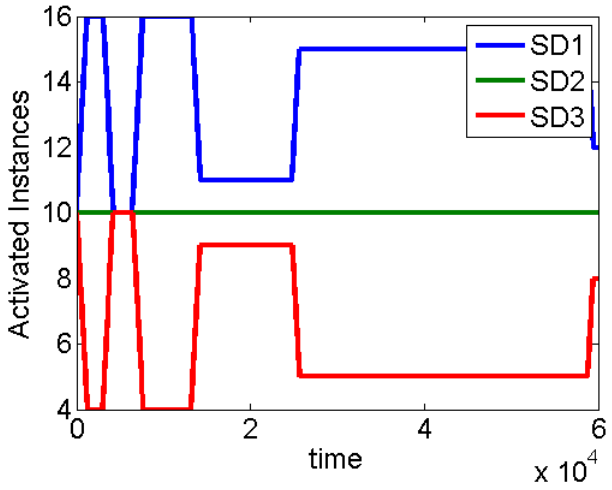


Figure 3.40: Modification 1: Variation in $B_{nm}$ Values, in Appliance 1, for $N = 10$

3.43 to Fig. 3.37 (later has $UT_m/DT_m = 2\%$ while earlier has $UT_m/DT_m = 5\%$) . After comparing, we conclude stricter tolerances cause more oscillations in both the goals and the instantiation matrix values. An interesting observation seen with $UT_m/DT_m = 0.2\%$, where the algorithm exhaust $B_{nm}$ (thus it fails to achieve the goal, see Fig. 3.44 and Fig. 3.45), remedy for this is discussed in Modification 1 and 2 by liming $Min[B_{nm}] = 1$ compare

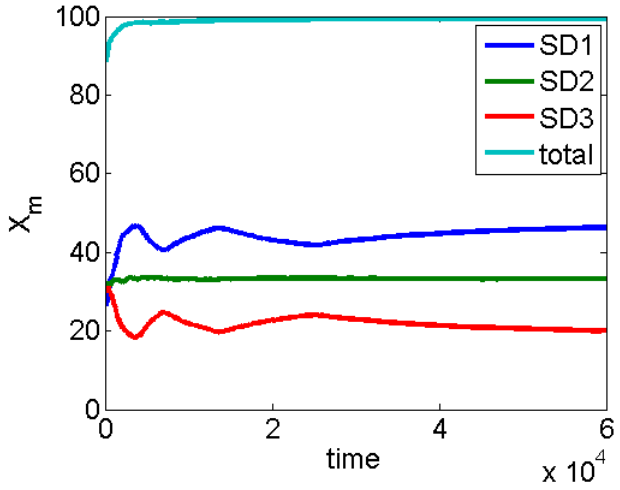


Figure 3.41: Modification 1: Utilization $X_m(T_k)$ vs time, in Appliance 1, for $N = 10$

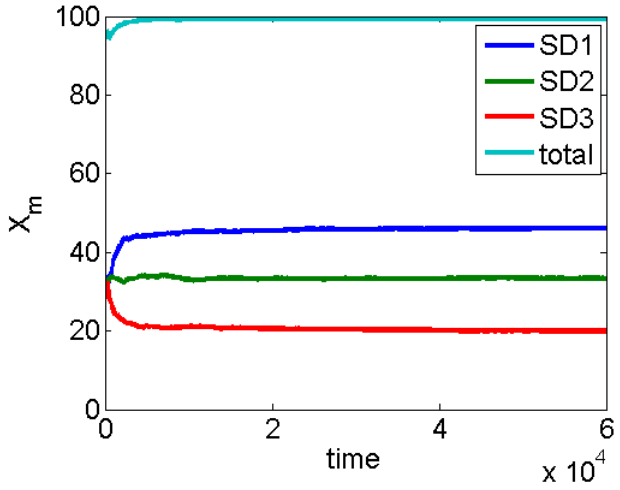


Figure 3.42: Modification 1: Utilization $X_m(T_k)$, effect of loose tolerances of 5%.

to current $Min[B_{nm}] = 0$.

As we stated before in section 3.2.6. "Achievable" and "non-achievable" (see section 3.2.6). Figure 3.36 demonstrates "achievable" case. For "non-achievable", in coherence with all feedback-based algorithms, this situation may "mislead" the algorithm which activate additional instances of the domain. However this modification unlike SAA1/SDA1

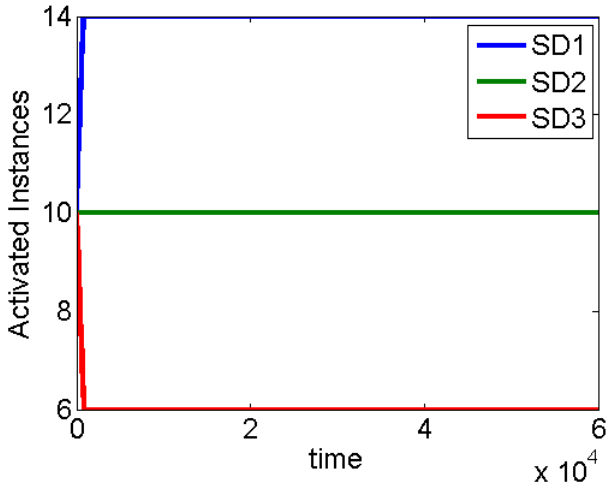


Figure 3.43: Modification 1: Variation in $B_{nm}$, effect of loose tolerances of 5%.

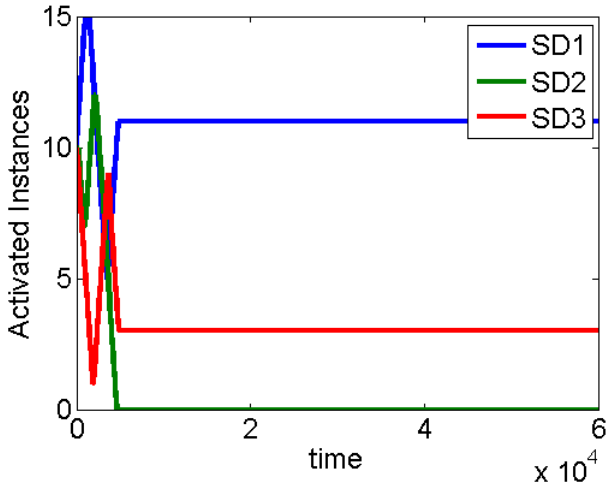


Figure 3.44: Modification 1: Utilization $X_m(T_k)$, effect of strict tolerances of 0.2%.

algorithm, restricts activation indefinitely and they system stays stable. Figure 3.46 exemplifies the effect of "non-achievable" goals. With same target goal, $\{P_1, P_2, P_3\} = \{44\%, 33\%, 22\%\}$, the arrival rate for SD1 was set low, so that this domain would never reach a 44% CPU utilization, even if it gets full access of the CPUs. However, the other two domains produced enough traffic to fully utilize their desired percentages however due

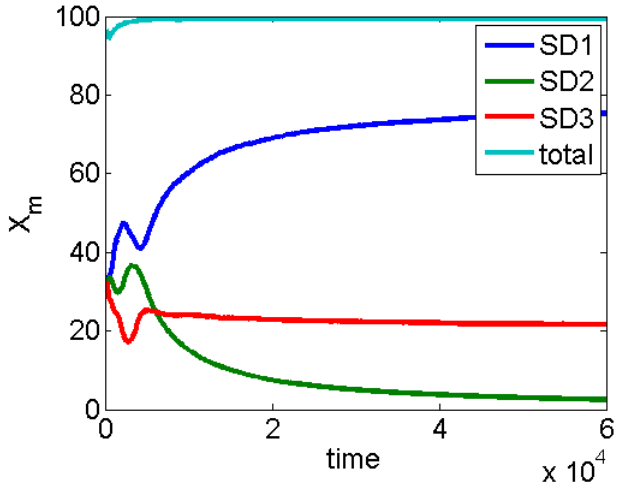


Figure 3.45: Modification 1: Variation in $B_{nm}$, effect of strict tolerances of 0.2%.

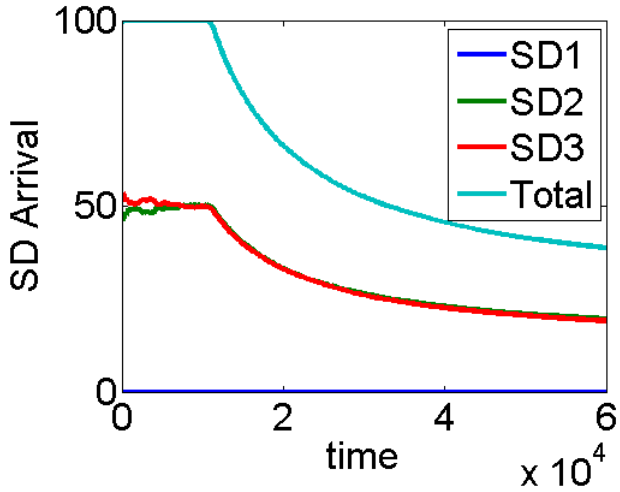


Figure 3.46: Modification 1: Utilization $X_m(T_k)$, "non-achievable" $P_m$ goals.

to overachieving, algorithm goes through exhaustive deactivation and after a while, no instance left for SD2 and SD3. As Figure 3.46 shows, these two domains (over)achieve their desired percentages in the beginning but later we see sharp fall due to the fact the system doesn't process them anymore . Figure 3.47 answers why the algorithm keeps activating instances for SD1, the "underachieving" domain, at the expense of the other two domains.

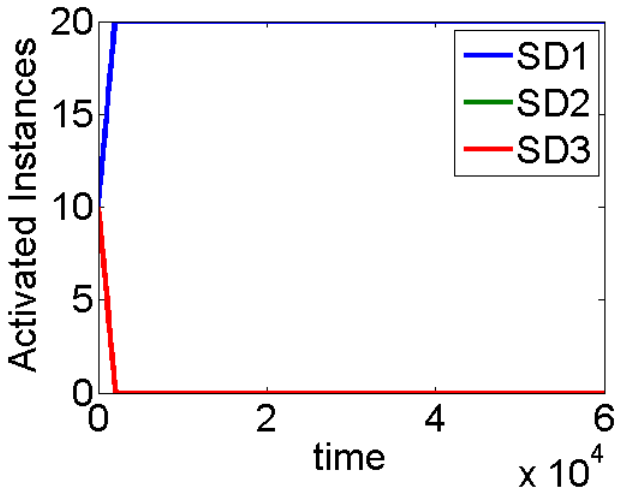


Figure 3.47: Modification 1: Potential for instability is Solved, "non-achievable" $P_m$ goals.

However, due to low arrival of SD1, the total utilization approaches 4% (as processing for SD2 and SD3 are stopped). And we are left with undesired white space. To utilize the white space, we propose a modification in section 3.3.2.

### 3.3.2 Modification 2: Restricting the algorithm not to deactivate service domain instances if that is the last activated instances of a service domain in an appliance ($SDA_{M1}/SAA_{M1}$)

In this modification, to restrict instances of one service domain to go down to zero (as we have seen in section 3.2.5,3.2.6 and 3.2.4) we only deactivate when the number of activated service domain is greater than 1 (without modification, it is greater than 0). Here our anticipation is, the algorithm will be able to provide service differentiation goal as we described in section 2.2. We try to find that out by answering the four questions we asked in page 3.

For the results depicted in Fig. 3.48, we set $N = 4$, $M = 3$, the desired goals are $\{P_m\} = \{44\%, 33\%, 22\%\}$ with 2% up and down threshold tolerances. We initialized the instantiation matrix to the same values in all four appliances; in order to create an "unfavorable" (see page 28) situation for the algorithm, the number of instances initialized were $\{2, 5, 10\}$ for service domain 1, 2 and 3 respectively Fig. 3.48 shows that the $SDA_M1/SAA_M1$ algorithm meets the desired goal despite the unfavorable initial instan-

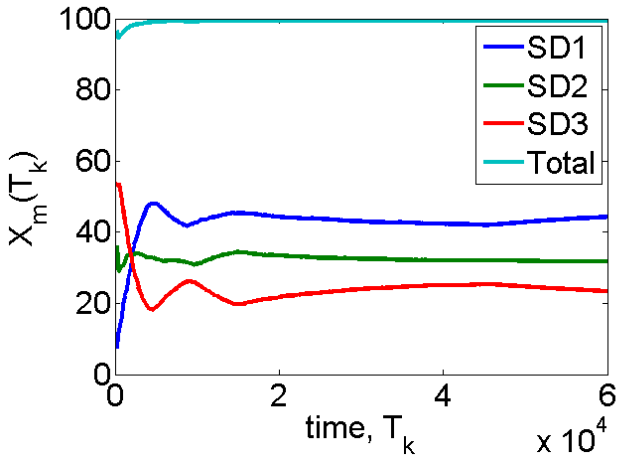


Figure 3.48: Modification 2: Utilization $X_m(T_k)$ vs time.

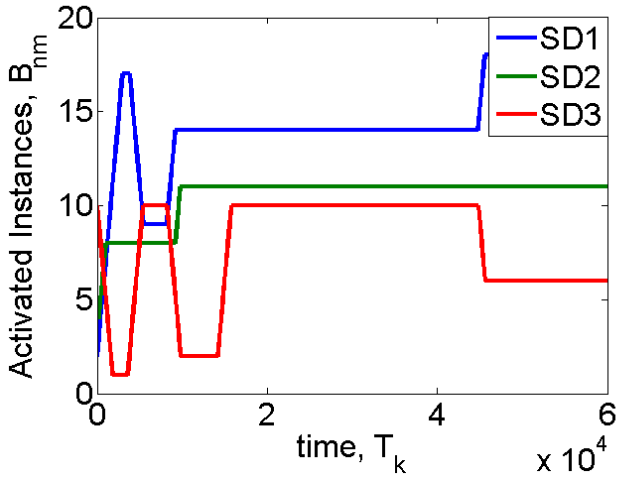


Figure 3.49: Modification 2: Variation in $B_{nm}$ values, appliance 1.

tiation.In this simulation, the total arrival rate was chosen high enough to keep the CPUs busy, hence the total utilization, as shown in Fig. 3.48 approaches 100%, which also presumably answer our question **Q1**.

Fig. 3.49 depicts, how the algorithm alters the number of instances of service domains in the appliances to achieve the goal. The algorithm causes oscillation in the beginning as for lower value of $k$, $X_m(k)$ changes abruptly which in turn causes oscillations

in the values of $B_{mn}$.

To answer **Q2**,we observe that desired $P_m$ goals depend on the actual arrival rates. Suppose we specify $\{P_1, P_2, P_3\} = \{44\%, 33\%, 22\%\}$ and the arrival rates and the average service times for the three service domains are equal. A static allocation, in this case, would allocate CPU times in the ratios 44% : 33% : 22%, wasting 11% for SD1, depriving SD3 of 33-22=11% and leaving a 22% "white space" (unused CPU resource). Figure 3.48 shows how SDA/SAA could achieve an equal allocation of CPU resources in this scenario, with a total CPU allocation of 100%, which would eliminate the white space altogether.

Question **Q3** involves effect of variation in the algorithm parameters $N, M, UT_m/DT_m$, $P_m$, initial $B_{nm}$. In all our experiments, the behavior of the algorithm (i.e., the nature of variations in the $B_{nm}$ values) as well as its performance (i.e., the achieved percentages) did not change with variation in the number of appliances $N$ or the number of service domains $M$. Figure. 3.51 and 3.52 shows some results for the "boundary cases" $N = 1$ and $N = 10$. As expected, the results conforms with those depicted in Fig. 3.48 and also answers the scalability questions asked in Question **Q4**.

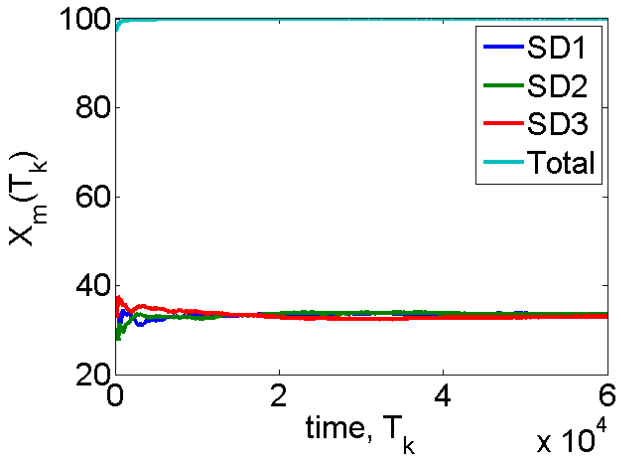


Figure 3.50: Modification 2: Utilization $X_m(T_k)$ vs time.

The effect of the tolerance parameters $UT_m/DT_m$, as we stated before, causes oscillation as Fig. 3.54. A rather strict setting of $UT_m = DT_m = 0.1\%$ for all domains caused this oscillation. Stricter tolerances cause more oscillations in both the goals and the instantiation matrix values. For comparison we can check Fig. 3.53 to Fig. 3.48 and Fig.

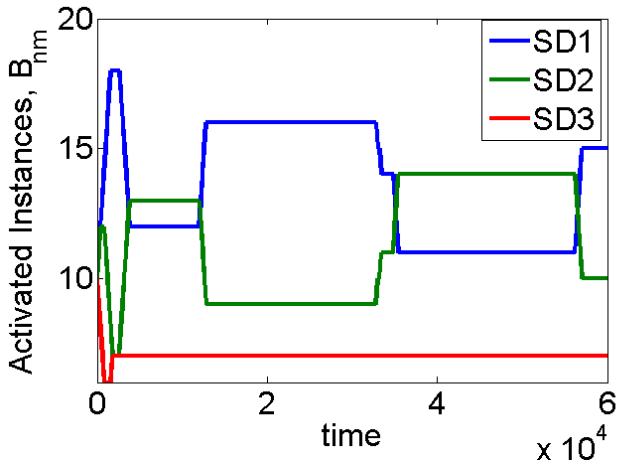


Figure 3.51: Modification 2: Variation in $B_{nm}$ Values, in Appliance 1, for $N = 1$

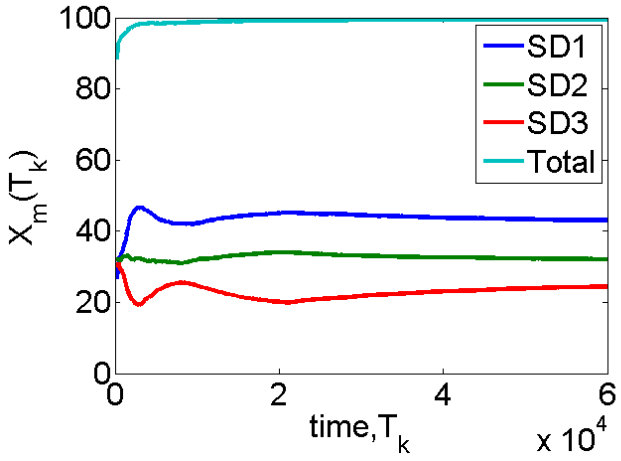


Figure 3.52: Modification 2: Utilization $X_m(T_k)$ vs time, in Appliance 1, for $N = 10$

3.54 to Fig. 3.49).

As we stated, the $P_m$ can be set by the administrator in one of two possible ways: "achievable" or "non-achievable". For "achievable" the arrival rate $\lambda_m$ and average service times $ES_m$ of the domain are such that $\lambda_m \cdot ES_m \geq P_m$, thus the system receives enough traffic to take advantage of the allocated CPU resource. Figure 3.48 demonstrates this case. For "non-achievable", we have that $\lambda_m \cdot ES_m < P_m$, which is simulating the scenario

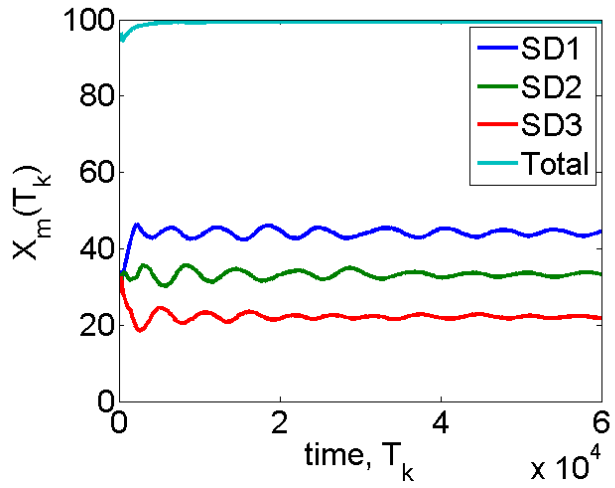


Figure 3.53: Modification 2: Utilization $X_m(T_k)$, effect of strict tolerances.

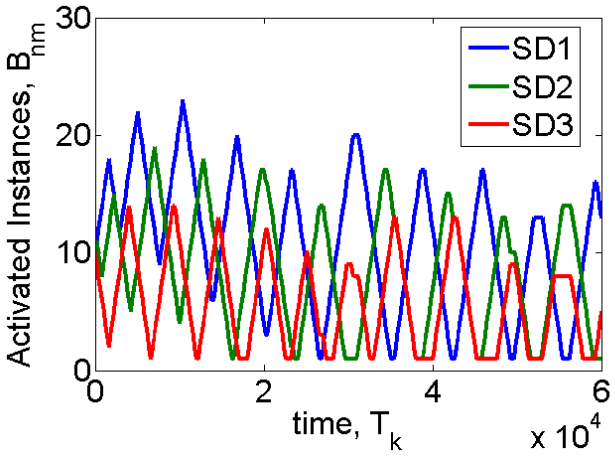


Figure 3.54: Modification 2: Variation in $B_{nm}$, effect of strict tolerances.

where the domain does not have enough traffic to take advantage of the allocated CPU resource. In coherence with all feedback-based algorithms, this situation may "mislead" the algorithm which will instigate the system to activate additional instances of the domain, causing "instability" and eventually affecting other domains.

Figure 3.55 exhibits the effect of "non-achievable" goals. With same $\{P_1, P_2, P_3\} = \{44\%, 33\%, 22\%\}$, the arrival rate for SD1 was set low, so that this domain would never reach

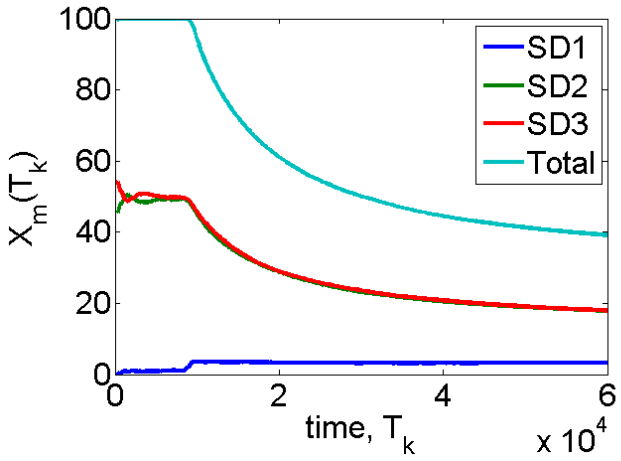


Figure 3.55: Modification 2: Utilization $X_m(T_k)$, "non-achievable" $P_m$ goals.

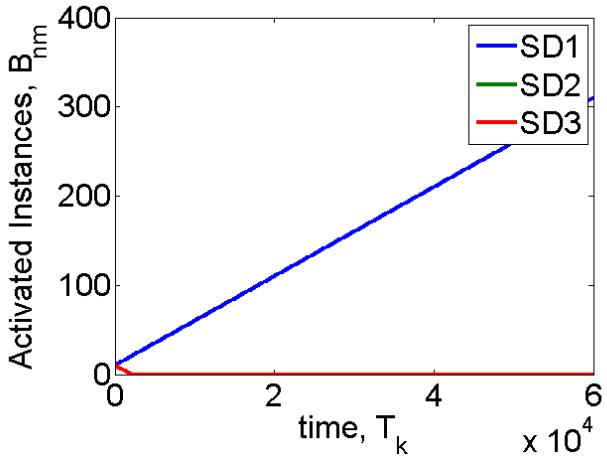


Figure 3.56: Modification 2: Potential for instability, "non-achievable" $P_m$ goals.

a 44% CPU utilization, even if it was given full access of the CPUs; its maximum utilization will eventually approach $\lambda_1 \cdot ES_1 \approx 6\%$ in this experiment. However, the other two domains produced enough traffic to fully utilize their desired percentages. As Figure 3.55 shows, these two domains (over)achieve their desired percentages. Figure 3.56 answers why the algorithm keeps activating instances for SD1, the "underachieving" domain, at the expense of the other two domains, which are left with only one activated instance each; this explains

why these two domains get an equal share of the CPU. The total CPU utilization stays at 100%, as shown in Fig. 3.55, eliminating any white space. In contrast with algorithm SAA1/SDA1, with which we started our experiments.

### 3.3.3 Modification 3: Restricting the algorithm to activate equal or less than the number of accumulated credit , accumulated by deactivations ($SDA_{M3}/SAA_{M3}$)

As we discussed in subsection 3.3.1 and 3.3.2, the original SAA1/SDA1 algorithm has two potential problem related to change in $B_{nm}$, exhaustive deactivation (causes one or more service domain fail to achieve the goal) and indefinite activation (causes thrashing). To address these, we can limit the number of activation and deactivation. In this modification, we only deactivate when the number of activated instance is at least two. And activate up to accumulated activation credit. Whenever, there is a deactivation, we increment accumulated activation credit by one. As we activate, we decrement the number of accumulated activation credit by one and we keep on activating as long we have credits. This tweak ensure, over time, the total number of activated instance over the cluster stays more or less equal to the number activated instances we started with, gives the administrator more control over the algorithm.

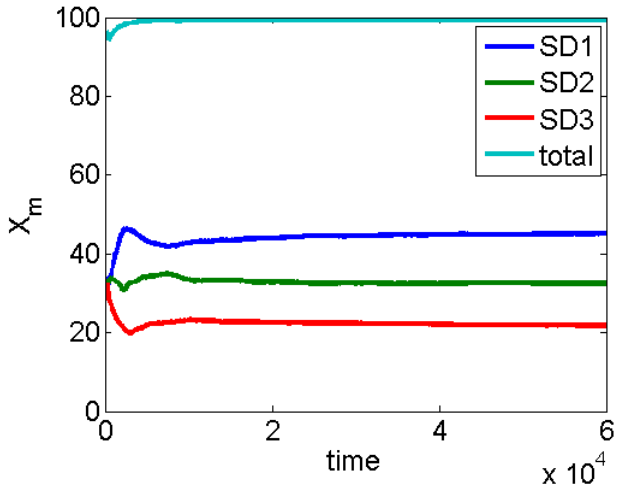


Figure 3.57: Modification 3: Utilization $X_m(T_k)$ vs time.

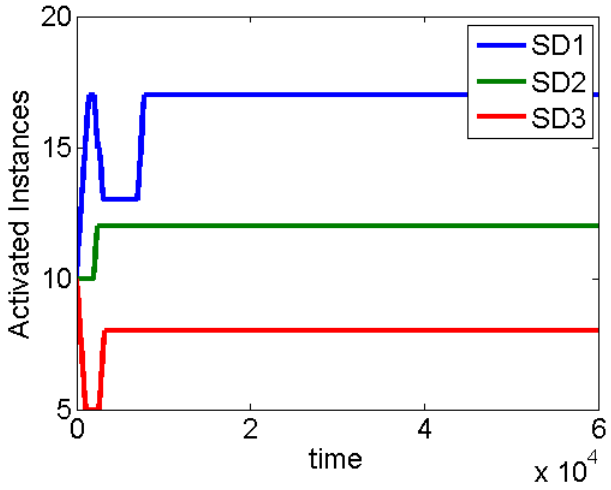


Figure 3.58: Modification 3: Variation in $B_{nm}$ values, appliance 1.

With same configuration we used in section 3.3.1 and section 3.3.2 ($N = 4$, $M = 3$, $\{P_m\} = \{44\%, 33\%, 22\%\}$, $U_{Tm}/D_{Tm} = 2\%$, $B_{nm} = 10$, arrival rate high enough to keep the CPU Busy) we get result as in Fig. 3.57. This algorithm make changes in $B_{nm}$ in order to achieve the goal as depicted in Fig. 3.58. Oscillation observed in the beginning for the reason discussed in page 29. $SDA_{M3}/SAA_{M3}$ satisfies the desired goal as shown in Fig. 3.57, which partially answers **Q1**.

Figure 3.57 shows how SDA/SAA could achieve an equal allocation of CPU resources in this scenario, with a total CPU allocation of 100%, which would eliminate the white space altogether. So, this algorithm is more responsive and utilize the white space unlike open-loop system. Thus we answer **Q2**.

Question **Q3** inquires effect of variation in the algorithm parameters $N, M, UT_m/DT_m$, $P_m$, initial $B_{nm}$. The effect of the tolerance parameters $UT_m/DT_m$, like other cases, causes oscillation as Fig. 3.60. From Fig. 3.59 to Fig. 3.57 and Fig. 3.60 to Fig. 3.58) we see stricter tolerance (0.1%) causes more oscillation compare to loose tolerance (2%).

Figure 3.61 exhibits the effect of "non-achievable" goals. With same $\{P_1, P_2, P_3\} = \{44\%, 33\%, 22\%\}$, the arrival rate for SD1 was set low, so that this domain would never reach a 44% CPU utilization. Unlike Modification 2, as we limit the activation and un deactivation in both way, we don't see exhaustive deactivation(like Modification 1) and thrashing(like

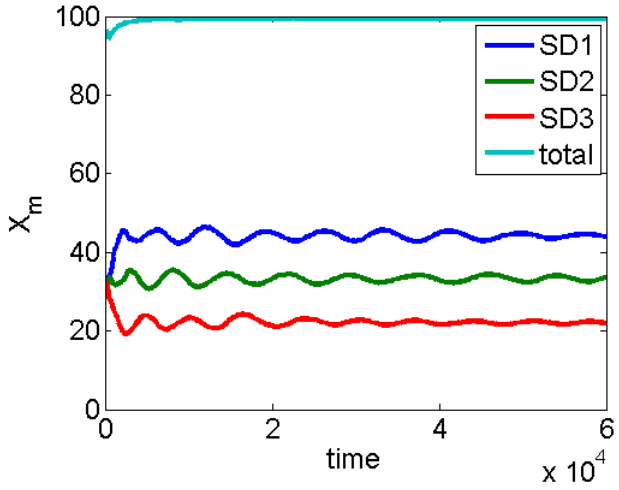


Figure 3.59: Modification 3: Utilization $X_m(T_k)$, effect of strict tolerances.

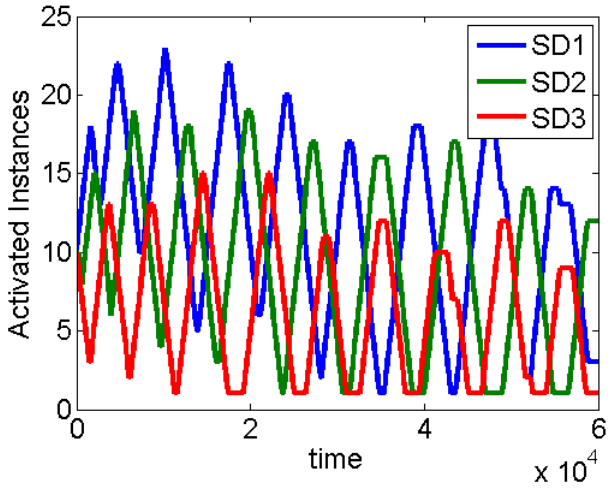


Figure 3.60: Modification 3: Variation in $B_{nm}$, effect of strict tolerances.

Modification 2). As the other two domains produced enough traffic to fully utilize their desired percentages. As Figure 3.61 shows, these two domains (over)achieve their desired percentages. Figure 3.62 answers why the algorithm keeps activating instances for SD1, the "underachieving" domain, at the expense of the other two domains. However, left with only one activated instance each; these two domains get an equal share of the CPU. Fig. 3.61

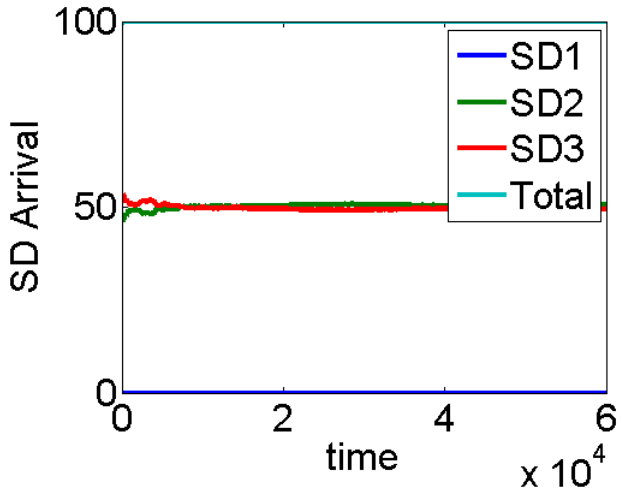


Figure 3.61: Modification 3: Utilization $X_m(T_k)$, "non-achievable" $P_m$ goals.

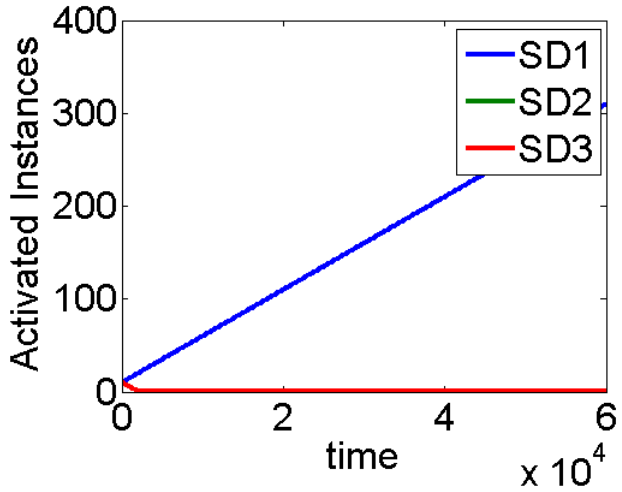


Figure 3.62: Modification 3: Potential for instability is Solved, "non-achievable" $P_m$ goals.

shows this variation of algorithm eliminates white space.

Figure. 3.63, 3.64, 3.65 and 3.66 shows some results for $N = 1$ and $N = 10$. As expected, the results conforms with those depicted in Fig. 3.57 and answers the scalability questions asked in Question **Q4**.

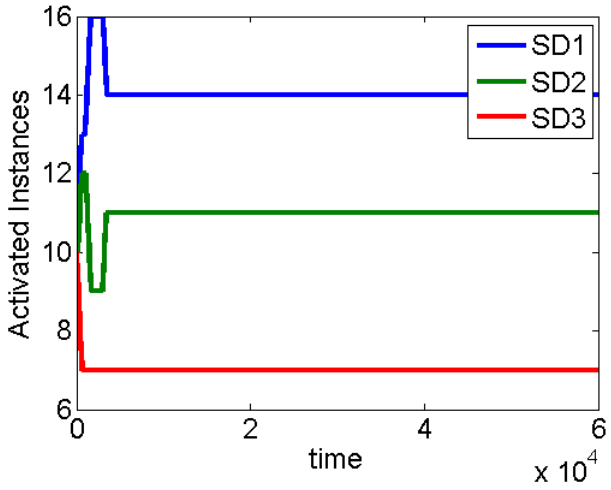


Figure 3.63: Modification 3: Variation in $B_{nm}$ Values, in Appliance 1, for $N = 1$

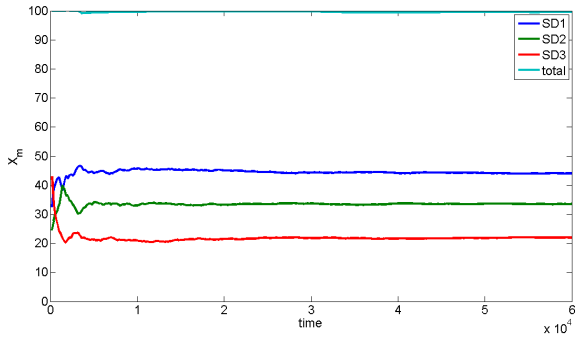


Figure 3.64: Modification 3: Utilization $X_m(T_k)$ vs time, in Appliance 1, for $N = 1$

## 3.4 Result Summary

Intentionally Left Blank

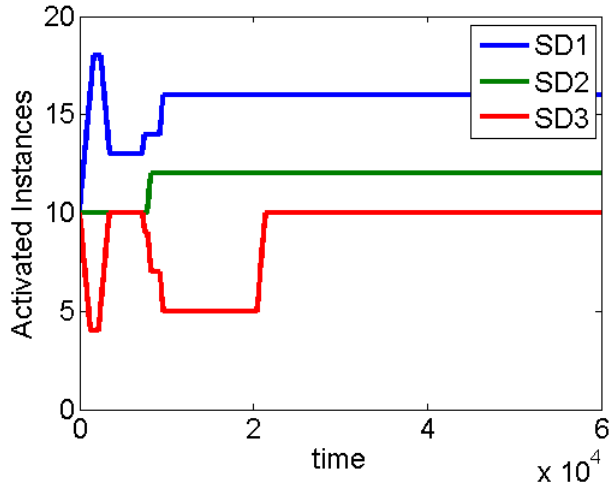


Figure 3.65: Modification 3: Variation in $B_{nm}$ Values, in Appliance 1, for $N = 10$

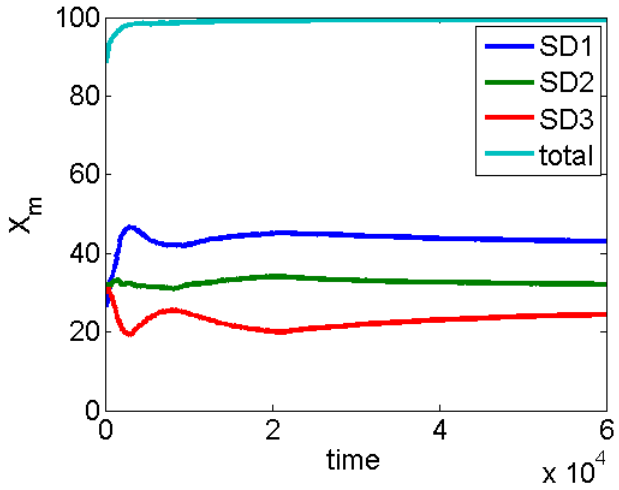


Figure 3.66: Modification 3: Utilization $X_m(T_k)$ vs time, in Appliance 1, for $N = 10$

# Chapter 4

# Conclusion and Future Work

In this work, we proposed solutions for "service differentiation" in service oriented architecture (SOA) and investigated performance of the solution by simulation.

## 4.1 Summary and Contribution

As a solution to the problem, a closed-loop feedback based reactive algorithm is proposed to provide "service differentiation" by controlling the incoming traffic as it reaches the gateway. As we name it SAA/SDA (Service Activation Algorithm ad Service Deactivation Algorithm), the algorithm deactivates and activates instances based on the measurement collected from middleware appliances to achieve the goal defined as percentage of allocated CPU resources for each services. In this research, following contributions has been made,

- Explanation of unfitness of static credit allocation is given. Thus, the emergence of closed-loop dynamic measurement based algorithm is suggested.
- Closed-loop feedback based algorithm is developed as per above mentioned suggestion that changes the state of each appliance by activating and deactivating service instances to achieve the administrator defined goal in cluster space dynamically.
- Discrete Event Simulator was developed to verify the algorithms behavior in terms of responsiveness and fidelity. With number of experiments, effect of some algorithm parameters on the algorithm behavior was observed and suggestion has been made

for the system administrator. These set of experiments gave us qualified answers to the problem we tried to solve.

- Modifications were proposed for newly discovered lapses in the algorithm found out during the experiments using the simulator. Number of experiments were executed to verify these modifications.

## 4.2 Future Research Directions

We have outlines a basic algorithm and some modification for it to address various shortcomings of the algorithm, we found out, the scenarios where our proposed algorithm can meet the goal. However, we anticipate that there are scopes for future research to design algorithms to address service differentiation in Service Oriented Architecture. Also more modifications to the current algorithm, in terms of restricting the current activation and deactivation method.

### 4.2.1 Normalizing the Activated Service Domain Instances

This is a modification suggested to the original algorithm to address the thrashing prone nature of the algorithm. In this modification, when the maximum number of activation in the cluster reaches a certain threshold, the $B_{nm}$ matrix can be normalized to reduce the threshold. We discuss one suggestive example as follows.

If the number of appliance is 2 and number of service domains is 3. Then the $B_{nm}$ will be a $2 \times 3$ matrix. Let assume, we start with,

$$B_{nm} = \begin{bmatrix} 2 & 2 & 2 \\ 2 & 2 & 2 \end{bmatrix}$$

However, after a while, if the algorithm reaches a situation, where, the total activation is more than 30 (the assumed threshold),

$$B_{nm} = \begin{bmatrix} 2 & 8 & 16 \\ 4 & 12 & 10 \end{bmatrix}$$

We can normalize the above mentioned $B_{nm}$ as,

$$B_{nm} = \begin{bmatrix} 1 & 4 & 8 \\ 2 & 6 & 5 \end{bmatrix}$$

Here, we can see, we can give the similar treatment to the service domains with less number of activated instances (25). It will also be interesting see the behavior of the algorithm in case of approximation while normalizing if the second row is changed to [1 3 3] or [1 3 2] instead of [2 6 5].

### 4.2.2 Applying Bin-Packing/Knapsack Algorithm in Activation Matrix $B_{nm}$

Bin-packing is very appropriate for this system, as we try to achieve a global target using local configuration. For a system described above with 2 appliances and 3 service domains, with same initial $B_{nm}$. And we reach after a while at the same $B_{nm}$ as follows,

$$B_{nm} = \begin{bmatrix} 2 & 8 & 16 \\ 4 & 12 & 10 \end{bmatrix}$$

Bin Packing or Knapsack Algorithm may result in following (not calculated, randomly assumed result),

$$B_{nm} = \begin{bmatrix} 2 & 20 & 0 \\ 4 & 0 & 26 \end{bmatrix}$$

Which will reduce the overhead for the whole system as each appliance only two service domains compare to three however, providing the same service. Also applying bin-packing, we can reduce the total number of activated instance in the system but achieve the same goal.

### 4.2.3 Applying Bin-Packing Algorithm in Queue

Applying Bin-Packing operation among the queues of different appliances, that transfer of service requests is possible between different appliances would be a very interesting way to enhance the performance of the algorithm. Our prediction is, with this procedure, activation/deactivation can be minimized, thus the overhead of the system.

### 4.2.4 Different Buffering and Load Balancing Scheme

Introducing queue for different service domains in the appliances would enable the appliance to process whichever service domain required to process to achieve the goal at

the time instance. Our anticipation, for the scenario where system doesn't receive enough request to fulfill the goal for one or more service domains, this mechanism would prioritize the processing of underachieving service domain that the system can provide best effort to achieve the goal, while if no service requests available for underachieving service domain, it will process other requests and the system will always be utilized.

Another way to solve the problem for the scenario of service request starvation is to use non-preemptive priority queue. Underachieving service requests will have higher priority and overachieving services requests will have lower priority thus the system can achieve the goal. Preemption would reduce the overall system efficiency (e.g. CPU Utilization), thus we are not suggesting preemptive queue to implement.

### 4.2.5 Design/Refine algorithm to achieve other goals than CPU Utilization

Other than CPU utilization, algorithm can be designed to achieve other system performance related goal, e.g. average waiting time, slow down, average response time.

### 4.2.6 Effect of Execution Interval on Algorithm

In this research, we have not observed the effect of execution interval on our algorithm behaviors. We believe, it will be worth while to analyze the relation between Execution Interval of the algorithm and distribution of service time of incoming requests.

### 4.2.7 Use Dynamic Programming to Achieve more than one goal

So far, the solutions to the problem can only solve one goal at a time (e.g. CPU utilization). However, using dynamic programming, optimization algorithm can be designed to achieve more than one goal at a time (e.g. overhead, waiting time).

# Bibliography

[1] Thomas Erl. *Service-Oriented Architecture : A Field Guide to Integrating XML and Web Services*. Prentice Hall PTR, April 2004.

[2] Munindar P. Singh Michael Huhns. Service oriented computing: Key concepts and principle. *IEEE Internet Computing, IEEE Computer Society*, pages 75–82, January-February 2005.

[3] Daniel A. Menascé, Daniel Barbará, and Ronald Dodge. Preserving qos of e-commerce sites through self-tuning: a performance model approach. In *EC '01: Proceedings of the 3rd ACM conference on Electronic Commerce*, pages 224–234, New York, NY, USA, 2001. ACM.

[4] Jaideep Chandrashekar, Zhi li Zhang, Zhenhai Duan, and Y. Thomas Hou. Service oriented internet. In *Proceedings of the 1st ICSOC*, 2003.

[5] A. K. Parekh and R. G. Gallager. A generalized processor sharing approach to flow control in integrated services networks: the single-node case. *IEEE/ACM Transactions on Networking*, 1(3):344–357, June 1993.

[6] Huican Zhu, Hong Tang, and Tao Yang. Demand-driven service differentiation for cluster-based network servers. In *In Proc. IEEE INFOCOM*, pages 679–688, 2001.

[7] Xiaoying Wang, Zhihui Du, Yinong Chen, Sanli Li, Dongjun Lan, Gang Wang, and Ying Chen. An autonomic provisioning framework for outsourcing data center based on virtual appliances. *Cluster Computing*, 11(3):229–245, 2008.

[8] A. Sharma, H. Adarkar, and S. Sengupta. Managing qos through prioritization in

web services. In *Proceedings on Fourth International Conference on Web Information Systems*, pages 140–148, Dec. 2003.

[9] S. Ranjan, J. Rolia, H. Fu, and E. Knightly. Qos-driven server migration for internet data centers. In *Proc. Tenth IEEE International Workshop on Quality of Service*, pages 3–12, 15–17 May 2002.

[10] Chun Zhang, Rong N. Chang, Chang-Shing Perng, Edward So, Chunqiang Tang, and Tao Tao. Leveraging service composition relationship to improve cpu demand estimation in soa environments. In *SCC '08: Proceedings of the 2008 IEEE International Conference on Services Computing*, pages 317–324, Washington, DC, USA, 2008. IEEE Computer Society.

[11] M.G. Kallitsis, R.D. Callaway, M. Devetsikiotis, and G. Michailidis. Distributed and dynamic resource allocation for delay sensitive network services. In *Proceedings of 51st Annual IEEE Global Telecommunications Conference (GLOBECOM)*, pages 1–6, 30 2008-Dec. 4 2008.

[12] R.D. Callaway, M. Devetsikiotis, and Chao Kan. Design and implementation of measurement-based resource allocation schemes within the realtime traffic flow measurement architecture. In *Proc. IEEE International Conference on Communications (ICC)*, volume 2, pages 1118–1122 Vol.2, June 2004.

[13] Xiaoying Wang, Zhihui Du, Yinong Chen, Sanli Li, Dongjun Lan, Gang Wang, and Ying Chen. An autonomic provisioning framework for outsourcing data center based on virtual appliances. *Cluster Computing*, 11(3):229–245, 2008.

[14] Daniel F. Garca, Javier Garca, Joaqun Entrialgo, Manuel Garca, Pablo Valledor, Rodrigo Garca, and Antonio M. Campos. A qos control mechanism to provide service differentiation and overload protection to internet scalable servers. *IEEE Transactions on Services Computing*, 2(1):3–16, 2009.