Misikir Eyob Gebrehiwot

Energy-Aware Queueing Models and Controls for Server Farms

School of Electrical Engineering

Thesis submitted for examination for the degree of Master of Science in Technology.

Espoo 21.05.2014

Thesis supervisor:

Dr. Samuli Aalto

Thesis advisor:

Dr. Pasi Lassila

Aalto University
School of Electrical
Engineering

AALTO UNIVERSITY
SCHOOL OF ELECTRICAL ENGINEERING

ABSTRACT OF THE
MASTER'S THESIS

Author: Misikir Eyob Gebrehiwot

Title: Energy-Aware Queueing Models and Controls for Server Farms

Date: 21.05.2014 Language: English Number of pages:7+82

Department of Communications and Networking

Professorship: Networking Code: S-38

Supervisor: Dr. Samuli Aalto

Advisor: Dr. Pasi Lassila

Data centers are known to consume substantial amounts of energy. Together with the rising cost of energy, this has created a major concern. Server farms, being integral parts of data centers, waste energy while they are idle. Turning idle servers off may appear to eliminate this wastage. However, turning the server back on at the arrival of the next service request incurs a setup cost in the form of additional delays and energy consumption. Thus, a careful analysis is required to come up with the optimal server control policy.

In this thesis, a queueing theoretic analysis of single server systems is carried out to determine optimal server control policies. Additionally, multiple server systems are also be studied through numerical methods. In this case, the task assignment policies that define how incoming requests are routed among the servers are also studied along with the control policies.

The results of this study illustrate that the optimal control policy for a single server system leaves an idle server on or switches it off immediately when there is no request to serve. This is a general result that does not depend on service, setup and idling time distributions. However, in the case of multiserver systems, there is a plethora of choices for task assignment and server control policies. Our study indicates that the combination of the Join the Shortest Queue and Most Recently Busy task assignment policies can save up to 30% of the system cost if the control policy applied can wait for a specific amount of time before turning a server off. Moreover, a similar gain can be achieved by the simple Join the Shortest Queue task assignment policy when it is used along with a control policy that leaves an optimized number of servers on while switching the remaining servers off when they become idle.

Keywords: Server Farms, Queueing Models, Energy Efficiency

Acknowledgement

This thesis was done in Aalto University, department of communications and networking. I would like to thank everybody involved in the process as it was a valuable experience for me.

I owe a dept of gratitude to my supervisor Dr. Samuli Aalto, without whose lasting support this thesis would not have come to fruition. His consistent comments and readiness to help are what any thesis worker could hope for.

I am also grateful to my instructor Dr. Pasi Lassila for his thoughtful comments and suggestions. One of his lectures introduced me to the simulation tool I applied in this thesis.

Finally, I would like to extend my sincere thanks to my family, specially my parents. Their ceaseless love and support has been my inspiration to strive for better things in life.

Contents

Abbreviations

DVS	Dynamic voltage scaling
EDP	Energy delay product
FIFO	First in first out
JSQ	Join shortest queue routing
MRB	Most recently busy routing
MRB-JSQ	MRB combined with JSQ with priority to MRB
MRB-RND	MRB combined with RND with priority to MRB
PS	Processor sharing
RND	Random routing
SRPT	Shortest processing remaining time

List of Figures

1 Introduction

Due to the ever growing demand for data storage and processing services, data centers have become a vital element of a typical ICT system. Technology giants like Google operate their own data centers, while small-scale businesses usually outsource their data services to data center providers. In either case, there is a demand for an almost instantaneous access to information stored in the data centers. This results in high performance and availability requirements on the data center providers. To meet these requirements, the data centers should usually be over-dimensioned for robustness and should also be equipped with backup power. Due to these factors, data centers consume a considerable amount of energy which is estimated to be 1.5% of the global electricity consumption [25]. This has created a global concern and several measures were suggested to tackle it, see [37, 1].

Supporting facilities, such as the cooling and lighting systems, are known to consume a large portion of the electrical energy. Hence, several innovative efforts, such as those applied in [39, 12], are being widely considered. However, this thesis will focus on the energy efficiency of the cluster of servers contained in the data center. These clusters are commonly referred to as server farms because they are housed in the same location with the ability to process service requests in a coordinated manner.

Server farms of a typical data center consume peak energy while they are serving requests and 60% of this peak power when idle [9]. Hence, a considerable amount of energy is wasted during this idle period. This condition is aggravated by the fact that the server farms are usually over-dimensioned. To tackle this problem, several server control policies have been suggested. A server control policy considers a certain set of metrics out of which a cost function is defined for an optimization task. The most commonly used metrics are the mean delay experienced by a request and the mean energy consumed to serve it. Alternatively, the mean number of requests in the system queue and the mean energy consumption per unit of time metrics can also be used. The mean rate at which the servers change their energy states is also another important metric.

In this thesis, control policies that utilize the aforementioned metrics will be studied from a queueing theoretic perspective. In the following discussion the control policy to be studied will be introduced and the task will be defined more precisely.

1.1 Problem Formulation

Once turned on, a traditional single server queueing system would stay idle if there is no job to serve without regard to energy wasted in this state. A tempting solution for this problem would be to switch a server off whenever it becomes idle and to turn it back on when the next job arrives. Although this simple approach may solve the problem in some cases, it can as well end up wasting more energy. This is due to

the fact that turning a server off introduces additional energy states to the system which makes the problem more complicated. Hence, the system needs to be modeled and studied carefully, to ensure energy efficiency.

1.1.1 System Model

As already mentioned, turning the server off results in additional states. That is, while turned off, a server needs to be setup before it can start to serve requests. Altogether, the system will have four energy states, *off*, *setup*, *busy* and *idle*. Each state is characterized by the associated energy consumption level and the time it takes to transition to the next possible state. In the *off* state the system does not consume energy but it needs to be setup before it starts serving requests. The *setup* state is a transient state between the *off* and *busy* states with its own energy consumption level. Jobs that arrive while the system is in this state need to wait until setup is finished in addition to whatever delay might be introduced by the service discipline.

In the busy state, the queued jobs are served according to the chosen service discipline, consuming the highest level of energy in the process. Once there are no more jobs to serve, there are three main options for choosing the next possible state. We might choose to turn the server off immediately after the last job leaves the system, hence eliminating the *idle* state completely, or to leave it idle until the next job arrives. In between these two extremes there exists one other option - letting the server idle for some time and then turn it off if no job arrives. The transition from *idle* to *busy* state is immediate while that of the *off* state is not, as already discussed. A *server control policy* is responsible for choosing between these options and applying it on the system.

1.1.2 Server Control Policy

A server control policy, also known as server management policy, is a set of rules that define the normal operation of the system. It can be designed to address a specific requirement such as load balancing or maximizing throughput. A control policy defines a cost function out of relevant system metrics and tries to minimize this cost function by varying system parameters. In this thesis, control policies that target energy efficiency of server farms will be studied. Therefore, the obvious choices as system metrics are the mean delay experienced by a job and the mean energy consumed in serving it. In addition, the mean rate at which the server changes state can also be of interest. Utilizing these metrics, the goal of a control policy is to come up with an operation rule that results in the optimal combination of performance and energy consumption.

Control policies can be categorized into three major groups based on what they emphasize to achieve.

Idling control policies: These policies leave the server idle while it becomes idle, allowing energy to be wasted in this state but maximizing performance.

Turn-off control policies: These sets of policies try to determine the right time to turn an idle server off so as to balance between the energy and delay costs.

Turn-on control policies: These sets of control policies emphasize on determining the right time to turn a server on. One possible rule under this policy is allowing a certain number of jobs to accumulate before turning the server on. This approach will have a negative effect on the performance by increasing the mean delay but it might reduce the mean energy consumed.

There is no straightforward rule in choosing between these policies. Even the *idling control policy*, which leaves a server idle while it is not processing jobs, might out-perform the others under some conditions. Generally, system parameters such as the load, the time it takes to setup a server, the relative energy consumption of the *idle* and *setup* states and the proportion of time spent in these states are the deciding factors on which policy is optimal.

In this thesis, a hybrid of these policies is applied on a single server system and the resulting cost function is optimized. Starting from the *off* state the server waits for a threshold number of jobs to accumulate before it is setup. Once setting up is finished, the queued jobs will be served according to the FIFO service discipline and on the departure of the last job, the server will go into the *idle* state. The length of this *idle* state is left for optimization and might range from 0 to ∞. If we choose to turn the server off, then the same cycle repeats starting from the *off* state.

1.1.3 Task Assignment Policies

In practice, data centers contain multiple servers. With each server usually having its own queue, a dispatching decision needs to be made in addition to the control policies discussed above. The task assignment policy makes these decisions by routing an incoming job to one of the available queues based on a predefined rule.

Figure 1 illustrates the model used in this thesis. The *random routing*, *most recently busy* (MRB) and *join shortest queue* (JSQ) task assignment policies will be studied from the energy efficiency perspective.

1.2 Outline of the thesis

In Section 2, queueing theoretic and technological backgrounds relevant to the task at hand will be summarized while Section 3 will cover a review of the state-of-the-art solutions proposed for energy efficient server farms. With the problem formulated, Section 4 gives a detailed analysis of the single server system followed by the multi-server system discussion in Section 5. Section 6 will provide a numerical illustration

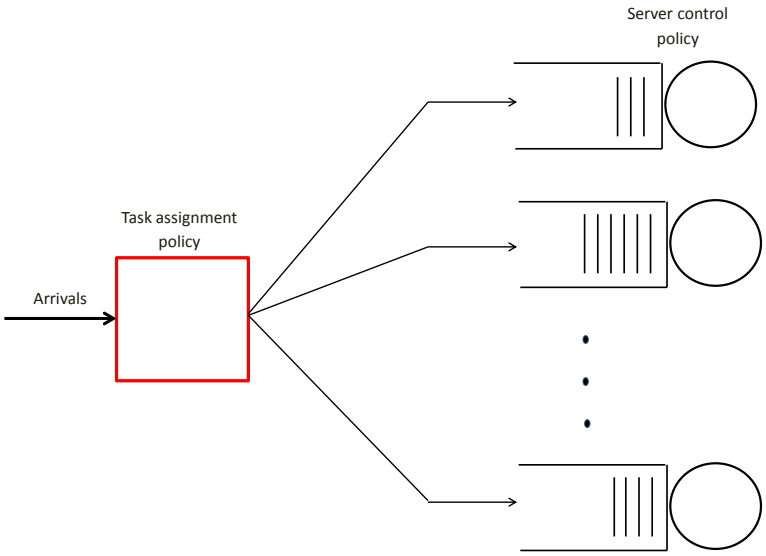


Figure 1: A task assignment policy, modeled by the red square, routs arriving jobs according to predefined set of rules. Once the job is in one of the queues, it will be handled by the respective server control policy and service discipline.

of the system, and Section 7 will conclude the thesis.

2 Background

The basic technological and theoretical knowledge required for this thesis will be provided in this section.

2.1 Technological background

A high level discussion of the technological aspects of server farms would provide a better intuition ahead of the stochastic analysis of the energy-aware control policies. For this reason, data center architectures and system level energy-aware designs that are relevant to our model will be summarized in the subsequent discussion.

2.1.1 Data Center Architecture

A data center is composed of several networked ICT equipments for data transmission, processing and storage. Server farms, also known as server clusters, lying at the heart of this network are responsible for processing incoming service requests. A server cluster is defined as "a parallel or distributed system that consists of a collection of interconnected whole computers, that is utilized as a single unified computing resource´´ in [32]. Server clusters are known to enhance the availability, scalability and manageability of the system [14]. Figure 2 illustrates the logical overview of a simplified data center, with the main components briefly described below.

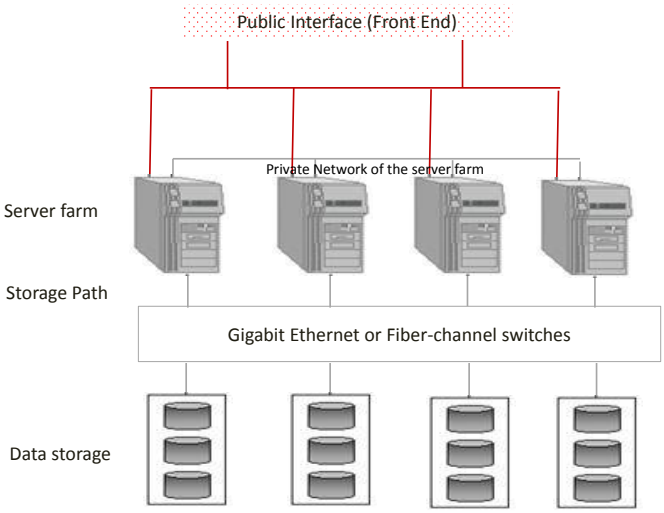


Figure 2: Data Center: A simplified architecture

Front End: This component provides external access to/from the server farm. It is an interface for inbound and outbound traffic.

Server(s): Service requests are handled in these servers. Depending on the data center design, they may or may not have a direct interface to the public network.

Data Storage: This is a common file system where data is stored which is connected to servers by a high speed storage path.

Storage Path: This component is responsible for providing high speed connection between the common storage and the servers. It can either be implemented using a gigabit ethernet or fiber-channel.

This discussion is only an overview of a simple data center architecture. More detailed explanation can be found in [4, 13, 14].

2.1.2 Energy-aware system level designs

Modern computing systems are designed to couple energy awareness with high performance and server farms are not any different. In the succeeding discussion, an overview of the most common models and engineering efforts that address this issue will be discussed at system level.

Model

Power dissipation in a typical processor can take on two forms: *dynamic* and *leakage* power. Dynamic power is consumed due to the normal operation of the processor which needs the switching of gates which in turn needs a power source. On the other hand, leakage power is caused by leakage current at a transistor level and accounts for about $20-30\%$ of the total power consumed [23]. The stochastic models in this thesis are targetted at reducing the dynamic power, more detailed studies about leakage power can be found in [23].

Dynamic power consumption, as discussed in [23], can be modeled by

$$P = C_L \cdot V^2 \cdot A \cdot f, \tag{1}$$

where C_L is the load capacitance, V is the supply voltage required for normal operation and A is the measure of circuit switches per clock cycle. In other words, it is a fraction between 0 and 1 that measures how active a circuit component is. Finally, f denotes the clock frequency which dictates the speed at which the processor operates.

At first glance, based on (1), the dynamic power might seem to have a quadratic relationship with voltage (V) and a linear relationship with frequency (f). However, the clock frequency is also proportional to the supply voltage. Thus, for a processor running at speed s, the proportionality can be given by

$$P \propto s^{\alpha},$$

where α represents the degree of proportionality which usually is in the range $(1, 3)$ [42].

Energy-aware designs

A brief review of the main design techniques applied to reduce dynamic power consumption is provided below.

Dynamic Voltage Scaling (DVS): DVS is a popular mechanism for reducing energy consumption in modern designs [45, 38, 5, 23]. Under DVS, the processor of a computing system does not always operate at full speed. When the task at hand does not require peak performance, energy can be saved by reducing the supply voltage, and hence reducing the clock frequency. Figure 3 shows a comparison of a DVS system with one that processes its tasks at full speed. It can be seen that the DVS system introduces additional delay to the completion time of a task, that is the processor will need to work for additional time, t_{add}, in order to complete the task. However, the energy reduction achieved by reducing the voltage, $V_{max} - V_i$, has much more significance. The speed scaling control policy, which will be discussed in Section 3, is implemented using the DVS technique.

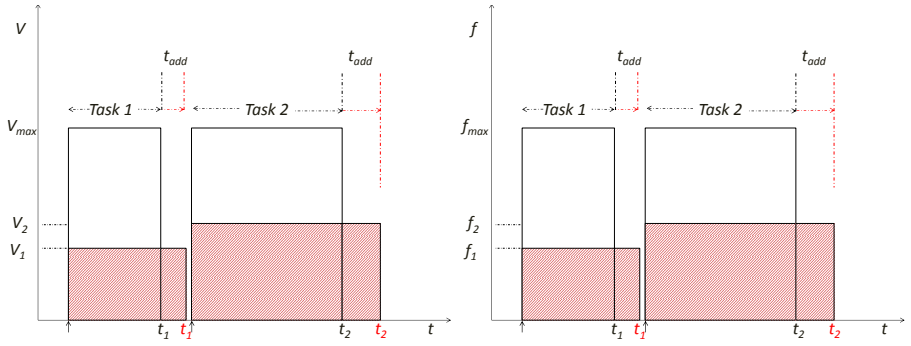
Clock Gating: Circuit components require clock signals for their normal operation. However, sending the clock signal to idle components will cause wastage of energy by increasing the activity factor, A, as shown in (1). This mechanism aims at reducing the power consumption by selectively sending the clock signal to only active circuit components. This will result in an activity factor, A, of 0 at that particular time for that unit which in turn saves power that would have been consumed by the unit as given by (1). The gated control policy, a stochastic representation of this technique, is going to be discussed in Section 3 and studies thereafter [23].

Other efforts: Many studies have been done aiming to achieve energy efficiency of computing systems. Intra-task DVS, a mechanism that brings energy-awareness to the application level is studied in [5] and references therein. Finally, a more complete survey of energy-aware mechanisms, including efforts on memory organization and input/output devices, can be found in [38].

2.2 Server farms: calculating energy consumption

As discussed in Section 1, a server in a server farm might have up to four energy states depending on the control policy. These are *off*, *setup*, *busy* and *idle*. Obviously, the total energy consumption of the server farm is the sum of energy consumptions at each state. If we are interested in computing the energy consumption in an interval $(0, T)$, then the total energy consumption, E, would read as

$$E = E_{\text{setup}} + E_{\text{busy}} + E_{\text{idle}},$$



(a) Supply voltage of a DVS system varies depending on the workload

(b) Clock frequency varies proportionally to supply voltage

Figure 3: Operation of a DVS system as compared to a system that operates at peak performance. The DVS sytem, shaded in red, achieves lower energy consumption by introducing addtional delay to the completion time of a task.

with each term representing energy consumption at the respective state. Each term in this calculation is a product of the power consumption at that state and the time spent in the respective state. For example, if P_{setup} and T_{setup} represent the power consumption and the time spent in the setup state then, $E_{\text{setup}} = T_{\text{setup}} P_{\text{setup}}$.

However, one may also compute the total energy by first calculating the expected power consumption. If the proportion of time spent in each state is given by $\pi_{\text{off}}, \pi_{\text{setup}}, \pi_{\text{busy}}$ and π_{idle}, then the total expected power consumption can be given as

$$\mathrm{E}[P] = \pi_{\text{setup}} P_{\text{setup}} + \pi_{\text{busy}} P_{\text{busy}} + \pi_{\text{idle}} P_{\text{idle}}.$$

Then the mean energy consumption over the entire period will be

$$\mathrm{E}[E] = \mathrm{E}[P]T = T_{\text{setup}} P_{\text{setup}} + T_{\text{busy}} P_{\text{busy}} + T_{\text{idle}} P_{\text{idle}}. \tag{2}$$

In this thesis, the power consumption values of $P_{\text{setup}} = 240W, P_{\text{busy}} = 240W$, $P_{\text{idle}} = 150W$ and $T_{\text{Setup}} = 200s$ will be employed. These values are attained from measurements on the Intel Xeon E5320 server as performed in [16]. In addition to the actual value for the setup time, its relative value compared to the mean service and idling times is of interest [16].

2.3 Theoretical background

Many phenomena have some degree of uncertainty associated with them. Some examples of stochastic phenomena in communications systems are: the number and duration of call requests coming to a call center, the number of users being served by a base transceiver station at a given time and the number of visits to a webpage. Designing and analysis of such systems involves the modeling of the underlying phenomena as stochastic processes.

A stochastic process is a set of random variables, X_t, representing one aspect of such a phenomenon, with the subscript t often referring to a time instant in the evolution of the process [36]. A random variable, as its name indicates, is a variable that can take on any value within a possible set of values, with each of these values having a corresponding probability of occurrence. The set containing all the possible values of the random variable is referred to as the *state space* of that random variable.

2.3.1 Markov processes

Let $(t_0, t_1, t_2, ..., t_n, t_{n+1}...)$ represent an increasing sequence of time values. A stochastic process X is said to be a Markov process if the possible outcome of $X(t_{n+1})$ solely depends on $X(t_n)$, without regard to how the process reached there [36]. Formally, a stochastic process is a Markov process if it satisfies the property

$$P\{X(t_{n+1}) = x | X(t_0) = x_0, X(t_1) = x_1, ..., X(t_n) = x_n\} = P\{X(t_{n+1}) = x | X(t_n) = x_n\},$$

where $\{x_0, x_1, \cdots, x_{n+1}\}$ is a subset of the state space of the random variable X.

If any state is reachable from any other state, either directly or through other states, the process is said to be an *irreducible* Markov process. Let $\pi_i(t)$ denote the probability of being in state i at time t for such an irreducible Markov process. Then, the set of values π_i, i referring to any state in the state space, is said to be the equilibrium distribution of the random variable X if

$$\pi_i = \lim_{t \to \infty} \pi_i(t).$$

Since these values are representing probabilities of being in the respective states, it holds that

$$\sum_i \pi_i = 1. \tag{3}$$

Moreover, for a system at equilibrium, the rate out of each state should be balanced with the rate into that state which leads us to the *global balance equations,*

$$\sum_i \pi_j q_{ji} = \sum_i \pi_i q_{ij}, \tag{4}$$

where q_{ij} denotes the transition rate from state i to j for all states i and j in the state space. The equilibrium distribution of the system can be solved by combining (3) and (4).

2.3.2 Birth-death processes

Simple queueing systems can often be represented by birth-death processes. Once exponential assumptions are taken, many stochastic processes fall into the birth-death category. A Markov process $X(t)$ defined on the state space $\{0, 1, ..., N, N+1, ...\}$

is said to be a birth-death process if state transition is possible only between two adjacent states [36]. In other words $q_{i,j} = 0$ for all non-neighboring states i and j.

Therefore, the state transition rate matrix, Q, of a birth-death process takes the form

$$Q = \begin{pmatrix} -q_{0,1} & q_{0,1} & 0 & 0\dots \\ q_{1,0} & -(q_{1,0}+q_{1,2}) & q_{1,2} & 0\dots \\ 0 & q_{2,1} & -(q_{2,1}+q_{2,3}) & q_{2,3}\dots \\ \vdots & \vdots & \vdots & \vdots \end{pmatrix}.$$

2.3.3 The $M/M/1$ queue

Being one of the simplest models in queueing theory, the $M/M/1$ queue provides the basis for performance analysis and design of single server systems. The term $M/M/1$ refers to Kendall's notation of the system. In general, Kendall's notation of a queueing system is given by $A/B/n/k$. The first letter, A, represents the distribution of the interarrival time while the second letter shows the service time distribution. The third and fourth letters represent the number of servers and the total capacity of the system respectively [21].

In an $M/M/1$ system, jobs arrive according to a Poisson process. Hence, the interarrival time is exponentially distributed. Moreover, service times are independent with exponential distribution. This is also depicted by the letter 'M' in Kendall's notation, which stands for 'Memoryless' [21]. Figure 4 illustrates a Markov process representing an $M/M/1$ queue with arrival rate λ and service rate μ.

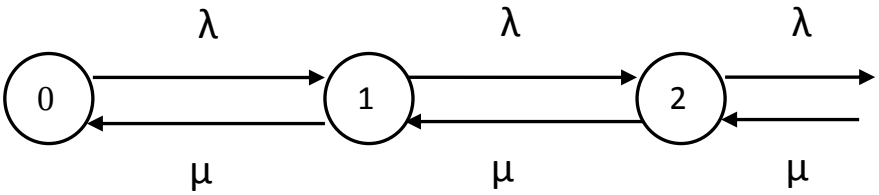


Figure 4: An $M/M/1$ queue with service rate μ and arrival rate λ.

Steady state analysis of the system yields the basic, yet very important, expression for the expected number of jobs in the system as

$$\mathrm{E}[N] = \frac{\rho}{1-\rho},$$

where the load $\rho = \lambda/\mu$. Using Little's formula, the expected time a job spends in

the system, E[T], is given as

$$\mathrm{E}[T] = \frac{\mathrm{E}[N]}{\lambda} = \frac{1}{\mu - \lambda}.$$

2.3.4 The $M/G/1$ queue

The $M/G/1$ queue is a basic part of queueing theory in which jobs arrive to the single server system according to a Poisson process with rate λ. Service times are independent and can assume any distribution with mean $1/\mu$. Throughout this thesis the service discipline is assumed to be FIFO, hence relevant results for the $M/G/1 - FIFO$ system will be summarized in the following discussion.

The performance of an $M/G/1$ cannot be studied by a simple Markov process, as was the case with $M/M/1$ queue. This is due to the fact that the Markov process representing the number of jobs in the system needs to keep track of the remaining service time of the job in service as well. This turns out to be a two dimensional Markov process which is not simple to analyze. For this reason, the study of an $M/G/1$ queue often involves the analysis of the imbedded Markov chain [24].

The expected waiting time of a job before it starts being served, E[W], can be given by

$$\mathrm{E}[W] = \frac{\rho}{1-\rho}\mathrm{E}[S^R] \tag{5}$$

where E[S^R] is the mean remaining service time of the job in service, provided that the test job finds the server busy on its arrival [21]. However, the mean remaining service time is given by

$$\mathrm{E}[S^R] = \frac{\mathrm{E}[S^2]}{2\mathrm{E}[S]}.$$

Using this in (5) will give the Pollaczek-Khinchin formula for he mean waiting time,

$$\mathrm{E}[W] = \frac{\lambda \mathrm{E}[S^2]}{2(1-\rho)}.$$

Once the waiting time is determined, the mean total delay of a job, E[T], can simply be calculated as

$$\mathrm{E}[T] = \mathrm{E}[W] + \mathrm{E}[S].$$

2.3.5 Renewal theory and regenerative processes

Queueing systems, such as the $M/G/1$ queue, are usually characterized by points in time beyond which the behavior of the system can be studied without regard to how it evolved to that point. Such time instants are referred to as regeneration points

and the process being studied as regenerative process. The work cycle analysis performed in this thesis applies such properties. Hence, regenerative processes and the associated renewal theory will be discussed briefly.

Let $\{X_n, n = 1, 2, ...\}$ be a sequence of random variables with $X_n > 0$. Furthermore, let the random variables be independent and identically distributed. If S_n and $N(t)$ represent the sum sequence and counter process of the random variable X, that is, if

$$S_0 = 0,\ S_n = \sum_{i=1}^{n} X_i \text{ and } N(t) = \sup\{n : S_n \leq t\},$$

then the process $N(t)$ is a renewal process [34, 21].

For example, let X represent the interarrival times in a Poisson arrival process with rate λ shown in Figure 5. The interarrival times, X_n, are independent and identically distributed. Hence, the process $N(t)$ that keeps track of the number of arrivals up to time t is a renewal process. Here, S_n would simply be the summation of n interarrival times. Clearly, one can see that

$$\frac{N(t)}{t} \to \lambda \text{ as } t \to \infty \text{ and } \frac{n}{S_n} \to \lambda \text{ as } n \to \infty.$$

See [34] for a complete discussion.

A random variable N is said to be the stopping time of the renewal sequence $\{X_n, n = 1, 2, ...\}$ if the event $N = n$ does not depend on $X_{n+1}, X_{n+2}, \therefore$. Taking the arrival process in Figure 5 as an example, the stopping time related to t_1 is $N(t_1) + 1 = 3$ and that of t_2 is $N(t_2) + 1 = 4$ [34]. This is intuitive since the event that we have 3 arrivals does not depend on the arrival time of the 4^{th} job.

Wald's equation
Let $\{X_n, n = 1, 2, ...\}$ be independent and identically distributed random variables with finite mean. In addition, let N be a stopping time for the renewal process $X_1, X_2, ...$ then

$$\mathrm{E}[\sum_{i=1}^{N} X_i] = \mathrm{E}[N] \cdot \mathrm{E}[X], \tag{6}$$

where $\mathrm{E}[N] < \infty$ [34, 21]. This result is known as *Wald's equation.*

Regenerative processes
A stochastic process $X(t)$ is said to be a regenerative process if its probabilistic characteristics repeat accross time spans $T_1, T_2, ...$ That is, the stochastic nature of the random variable X during time period T_1 is identical to, but does not depend on, that of T_2 and so on [34]. Here, we should note that the sequence $\{T_1, T_2, ...\}$

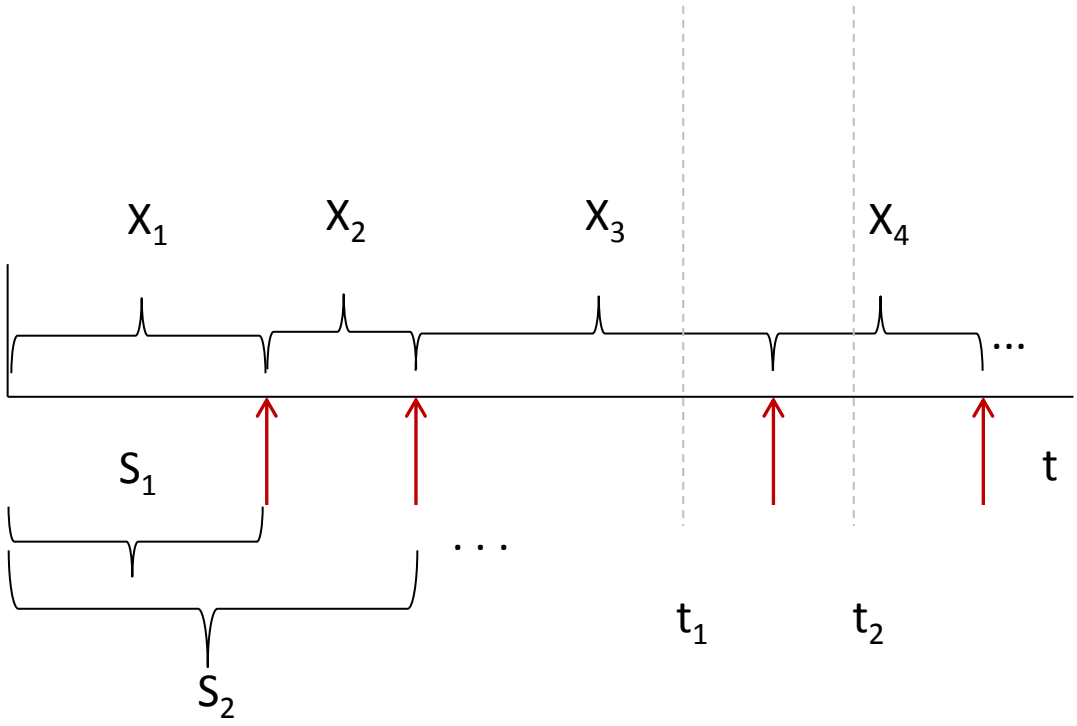


Figure 5: Interarrival times as renewal process .

is a renewal process. The steady state expectation of the regenerative process $X(t)$ can be computed as

$$\mathrm{E}[X] = \frac{\mathrm{E}[\int_0^T X(t)dt]}{\mathrm{E}[T]}$$

where $\mathrm{E}[T] = \mathrm{E}[T_1]$ is the expected length of the renewal cycle.

3 Literature review

In the past few years, the optimization problem between performance and energy in server farms has gained the attention of the research community and in this section a brief summary of the proposed solutions will be provided. There are a number of suggested design considerations and server farm management schemes so we will categorize them according to the models they utilize, namely:

- Single server models with a central queue,
- Multiple server models with a central queue and
- Multiple server models with parallel queues and task assignment policies.

But before starting the review on the models, let us discuss the cost models involved and the metrics utilized in these models.

3.1 Cost Models

Obviously, the task is an optimization problem involving a trade-off between energy and performance. For this, one should choose a cost model for the system. The goal of the task will be to determine the optimal server control policy based on the chosen cost model. Hence we need to define metrics with respect to which the optimization is going to be done. Different cost models with their associated metrics are discussed below.

Weighted sum cost function

As its name indicates, weighted sum of delay and energy consumption metrics is used in this cost model, see [20, 8, 6, 42, 3, 31, 27]. The cost function is given as

$$\mathrm{E}[T] + \frac{\mathrm{E}[E]}{\beta}, \tag{7}$$

where $\mathrm{E}[T]$ is the mean delay of a job, $\mathrm{E}[E]$ is the average energy consumed per job and β is a weighting parameter that can be manipulated according to our need to emphasize the importance of delay or energy[1].

Energy-Delay Product (EDP)

The Energy-Delay Product (EDP) is another cost function proposed in [23, 16]. The product of mean power consumption, $\mathrm{E}[P]$, and mean delay is used in this cost function. Here, the main concern over not using the weighted sum cost function is that although it captures changes in magnitude of the metrics, it does not capture

[1] It is easy to see that this equation can readily be converted to contain mean number of jobs instead of mean delay by utilizing Little's Formula, $E[N] = \lambda \mathrm{E}[T]$. In this case, the second term will become mean energy consumption per unit of time (power), $\mathrm{E}[P] = \lambda \mathrm{E}[E]$.

the relative change of the magnitudes. For example, mean response time improvement from 10s to 5s and from 1000s to 995s are regarded equally by the weighted sum objective function. However, it is intuitively clear that the former has a more significant improvement. One can also use the mean energy consumption instead of power by applying the simple relationship $\mathrm{E}[P] = \lambda \mathrm{E}[E]$. If we choose to use the mean power consumption instead, the cost function will read as,

$$\mathrm{E}[P]\mathrm{E}[T]. \tag{8}$$

Hybrid cost function

An objective function that can be seen as a hybrid of the above two is also proposed in [28]. This approach has an additional metric to the two already discussed. This is the expected rate at which a server switches between two energy states, more specifically from a higher energy state to a lower one. The objective function is a sum of weigthed terms with each of them containing a product of mean delay, mean energy consumption and mean switching rate,

$$\sum_{i=1}^{M} \beta_i \mathrm{E}[T]^{w_{T,i}} \mathrm{E}[E]^{w_{E,i}} \mathrm{E}[C]^{-w_{s,i}}, \tag{9}$$

where β_i, $w_{T,i}$, $w_{E,i}$ and $w_{s,i}$ are the weighting factors for a cost function with M terms. The mean switching rate is just the inverse of the mean time it takes a server to toggle between the lowest and highest energy states, which simply is the mean busy cycle of the system. Thus, if we denote the mean busy cycle by $\mathrm{E}[C]$, then the mean switching rate will be $1/\mathrm{E}[C]$. The cost function in (9) can easily be converted into (7) or (8) by adjusting the choice of β_i, $w_{T,i}$, $w_{E,i}$ and $w_{s,i}$. For example, if we set $\beta_1 = 1$, $w_{T,1} = 1$, $\beta_2 = 1/\beta$ and $w_{E,2} = 1$ while leaving the rest of the weights to be zero, we will have the weighted sum cost function given in (7). On the other hand, if we set $\beta_1 = \lambda$, $w_{T,1} = 1$, $w_{E,1} = 1$ and set all the rest weight parameters to zero, we will get a product form cost function given in (8).

Constrained optimization

In some practical scenario such as CPU scheduling of an operating system or any QoS service, the metrics under consideration might only be able to take on values limited within a specific interval. For example, there might be a service level agreement (SLA) between a data center provider and its customer that specifies the maximum acceptable delay. Depending on which metric is constrained, the optimization task can be divided into performance or energy constrained optimization.

Performance constrained optimization: In the above example, the data center operator will try to minimize its energy costs while guaranteeing the delay limits specified in the SLA, thus, falling into the performance constrained optimization category. In [44, 2] a similar constrained optimization scenario is discussed where

all jobs in the system have their own maximum acceptable delay. Under this cost function, an arriving job is specified by three variables: its arrival time, the deadline time before which the job must be processed fully and its service requirement which can be expressed in terms of number of CPU cycles required, size of data to be processed or by any other appropriate metric. Any feasible control policy shall allocate enough processing resources to jobs such that they will at least be completed before their deadline expires. In other words, the optimization of the control policy is constrained by the performance requirement of the jobs being processed. To illustrate this mathematically, let j be one of the jobs in the system at time t while $J(t)$ represents the current job being serviced. In addition, let a_j and d_j be the arrival and deadline times for job j with the time dependent service rate $\mu(t)$. Then a control policy S is feasible if

$$\int_{a_j}^{d_j} \mu(t) 1_{\{J(t)=j\}} dt = R_j,$$

for all j, where R_j is resource requirement of job j. In this case, the objective of the constrained optimization is to find a control policy among the feasible set which minimizes the mean energy consumption of the system.

Energy constrained optimization: Contrary to the above discussion, one might need to determine the best performance possible for a certain energy budget. This type of optimization might become more practical since most systems do not set delay requirement at job level [33]. Energy constrained optimization has recently gained more attention due to the popularity of battery-powered systems in which available energy is limited, see [11, 33].

Pareto optimality

Let vector $\mathbf{C}$ contain all possible combinations of energy and delay metrics incurred by a family of server management policies. That is, $\mathbf{C}$ contains all possible vectors $\mathbf{c}$ such that $\mathbf{c} = [c_E, c_T]$, where c_E and c_T are the energy and delay metrics that define the vector $\mathbf{c}$. Then a vector $\mathbf{p} \in \mathbf{C}$ is said to be *Pareto optimal* if there does not exist another vector $\mathbf{q} \in \mathbf{C}$ that satisfies one of the following two conditions

- $q_E \leq p_E$ and $q_T < p_T$ or
- $q_T \leq p_T$ and $q_E < p_E$

In other words, a certain delay and energy combination is said to be Pareto optimal if it is not possible to further reduce either delay or the energy consumption without causing the other one to increase [10]. A set of such Pareto optimal points constitute the Pareto frontier of the server management policy under study. Figure 6 illustrates an ideal example of a Pareto optimality study of an energy aware server. The dark points are Pareto optimal whereas the lightly shaded ones are not. The lightly shaded points on the vertical and horizontal part of the red line are not optimal

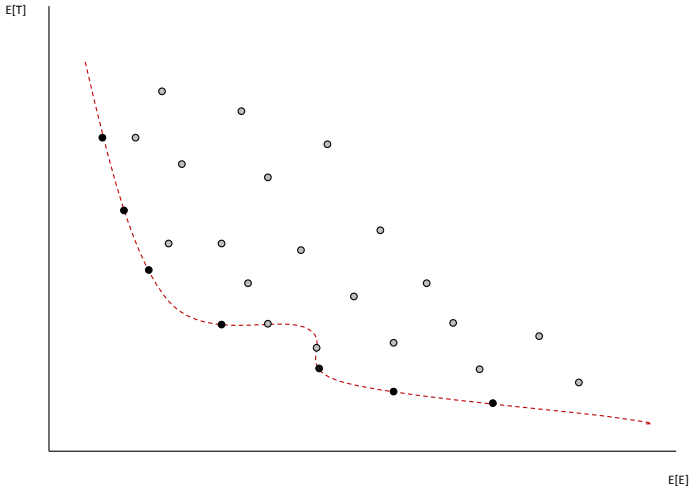


Figure 6: Pareto optimality for an energy aware server

because further reduction of either delay or energy consumption is possible. The part of the red line that connects the dark points constitutes the Pareto frontier.

The Pareto optimality method will narrow down the choice of system parameter values greatly by considering only those values that result in a point that is in the Pareto frontier. Here the energy and delay metrics can also be weighted to take the associated costs into consideration. The points obtained from the Pareto optimality cost model can be used for further study of the system in combination with either the weighted sum cost model or the EDP cost model discussed above.

3.2 Queuing Models

The energy-performance trade-off can be seen from different view points depending on the number of servers involved. Even in a single server case, the task can get quite complicated. One may think of to switch or not to switch the server off while it becomes idle depending on several factors such as system load and energy budget. In the case where we decide to switch the server off, we are faced with yet another decision to make. Should we wait for some time interval before switching off or not? If yes how should this interval be chosen? Now that the server is off, should we start it up at the arrival of the first job in the next busy period or should we wait till a certain amount of work accumulates? If yes how should we choose this amount of work? A separate question can be, should we have multiple energy states between on and off or not? In addition to these there can be speed scaling mechanism that tries to minimize energy consumption by varying the speed at which the server runs based on the amount of jobs at hand.

In the case of multiple servers, in addition to issues already mentioned above,

one may also come up with lots of server management questions, such as should we turn all the servers off if they are all idle or should we leave some of them idle. The question of using staggered bootup can also be asked here. Staggered bootup is a mechanism in which servers are restricted to bootup one by one in order to prevent huge amounts of power being drawn in case of bursty arrivals. That is, a job that arrives to a system that uses staggered bootup will first look for an idle server, if it does not find one it will look for an off server and start it up only if there is no other server in the bootup state. If there is already a server in the bootup state, then the job should wait in the queue.

Several models are proposed, in the literature, in an attempt to determine the optimal control policies under different scenarios. In the following section these models will be summarized.

3.2.1 Single Server Models

Here we are going to discuss prior studies that considered the energy-performance trade-off utilizing a single server with a central queue model. Mainly, three control policies have drawn the attention of researchers in the single server case. These are,

- Optimized static service speed
- Gated static speed
- Dynamic speed scaling

Optimized static service speed

This is the simplest of the three control policies. Under this policy, a server runs on a static speed and is never turned off even if it is idle. Hence, it will only have the *busy* and *idle* states. However, its speed is optimized taking both performance and energy consumption into consideration. If the offered traffic can be modeled by a Poisson process, then the mean delay is given by that of a standard M/G/1 system, $\mathrm{E}[T] = E[T_{M/G/1}]$. On the other hand, the mean power consumption will simply be

$$\mathrm{E}[P] = \rho P_{\mathrm{busy}} + (1-\rho) P_{\mathrm{idle}},$$

where ρ is the fraction of time the server is busy and π_{busy}, π_{idle} are the respective power consumptions in the *idle* and *busy* states. With the mean values of delay and energy metrics determined, the optimization can be done by taking the service speed that minimizes the cost function of our choice. A case of the weighted sum cost function can be found in [42, 27].

Gated static speed

This control policy tries to reduce energy consumption by turning the server off when there is no job to serve. Hence, the server will have three states: *busy*, *off*

and *setup*. In the *busy* state jobs will be served with an optimized static speed and when there are no more jobs to serve, the server will instantly be turned off changing its state to *off*. The time it takes to turn the server off and the associated energy consumptions are assumed to be negligible. However, on the arrival of the first job the server will go into the *setup* state. Once setup is completed, the system state will once again change to *busy* in which the queued jobs will be serviced. Even though turning an idle server off might seem to reduce the energy cost of the server, the system should be studied in more detail before drawing any conclusion. This is due to the fact that turning the server on incurs an additional cost known as the *setup cost*, see [18, 19, 17, 27]. The setup cost may have two components, energy consumed to turn the server on and the additional delay experienced by the jobs that arrive while the server is in the setup state. Stochastic analysis of such a system in [18, 19, 17, 27] showed that the total mean delay can be decomposed into the mean delay of a standard M/G/1 system and the mean setup time of the server, that is,

$$\mathrm{E}[T] = E[T_{M/G/1}] + \frac{1}{\gamma}, \tag{10}$$

for exponentially distributed setup time with mean $1/\gamma$. In fact, this decomposition property is a well known result from previous studies for a general system in [41] and for vacation models in [26, 29, 30]. If the exponential assumption for setup time is relaxed to a more general distribution, [41] showed that (10) will have the form

$$\mathrm{E}[T] = E[T_{M/G/1}] + \frac{\mathrm{E}[T_{\mathrm{setup}}] + \frac{\lambda}{2}\mathrm{E}[T^2_{\mathrm{setup}}]}{1 + \lambda \mathrm{E}[T_{\mathrm{setup}}]}. \tag{11}$$

The mean power consumption of this system will simply be

$$\mathrm{E}[P] = \rho P_{\mathrm{busy}} + \pi_{setup} P_{\mathrm{setup}},$$

where π_{setup} is the fraction of time the system spends in the *setup* state and P_{setup} is the corresponding power consumption.

Dynamic speed scaling

The dynamic speed scaling control policy, as studied in [44, 45, 7, 42, 3, 27], adjusts the speed of the server as a function of the number of jobs in the system. At system level, this policy is implemented by the dynamic voltage scaling mechanism discussed in Section 2. Due to the complex nature of this control policy, one needs to first define the associated cost function and perform the optimization based on that. That is, the aim will be to determine the optimal service speed in terms of the minimum achievable average cost. For this, we need to determine a vector of service rates $\mu = (\mu_1, \mu_2, ..., \mu_n, ...)$ that result in the minimum achievable average cost. The subscript n indicates the number of jobs in the system. Based on the general form given in [20], the average cost of the system can be modeled using the

weighted sum cost function as

$$\sum_{n=0}^{\infty} \pi_n(\mu)(n + \frac{f(\mu_n)}{\beta}),$$

where $\pi_n(\mu)$ is the steady state probability of having n jobs when the vector of service rates μ is chosen and $f(\mu_n)$ represents the power required to run the server at speed μ_n. This model assumes that there are no delay and power costs for changing the processing speed of the server.

A comparison of the dynamic speed scaling and gated control policies in [42, 27] illustrates that the gated system performs almost as good as the dynamically scaled one for a sufficiently small setup time. However, even in the case of sufficiently small setup time, system load may vary from the design load or the system might have to handle bursty traffic. In such cases, the gated system will suffer from being either wasteful or unstable depending on the volume of the actual load with respect to the design load. On the contrary, dynamic speed scaling is oblivious to this kind of load mis-estimation and it is also known to be robust as compared to its gated counterpart.

In addition to this, the impact of speed scaling on fairness, robustness and optimality is studied in [3] for the Processor Sharing (PS) and Shortest Remaining Processing Time (SRPT) disciplines. Therein, it is noted that it is impossible to achieve all three objectives at the same time with the speed scaling scheme.

Gated system with turn-on threshold and idling time

In the above discussion of the system with a gated static speed, the aim of the control policy was to determine the optimal static speed under the specified conditions. However, the optimization task can also be seen from a different perspective. For example, from the above studies it is not clear whether it pays off to switch the server off at the instant it becomes idle or not. Once the server is off another thing to consider can be, whether we should allow a certain number of jobs to accumulate before turning the server on. To address these issues a system with an *idling time* and *turn-on threshold* is studied in [28]. The hybrid cost function given in (9) is used to study the behavior of the system with the following model assumptions:

- The server is assumed to have four different energy states: *off, setup, busy* and *idle* with their respective mean energy consumptions $P_{\text{off}} = 0$, P_{setup}, P_{busy} and P_{idle}.
- The system has a threshold of k jobs up to which the server is not going to be turned on if it has been off. We will refer to this value as *Turn-on threshold* throughout the succeeding discussion.

- As the k^{th} job arrives and the server is off, then it will enter the *setup* state. Once setting up the server is completed, it will go into the *busy* state, in which state the queued jobs are served.
- When there is no job to serve the system goes to the *idle* state. Once the server is in the *idle* state, it will stay idle for a certain time interval and goes to the off state. This time interval will be named as *idling time* in the following sections. This interval will initially be assumed to have exponential distribution with a rate parameter α.
- Setup time is also exponentially distributed with a rate γ.
- Transition from high energy state to low energy state is assumed to be instantaneous.

Taking a weighted sum of mean delay, mean power consumption and mean switching rate metrics as an objective function, an important result of [28] is that once the server becomes idle, the optimal scheme will be to either turn it off immediately ($\alpha \to \infty$) or to leave it idle ($\alpha = 0$). We will study this and other related results in more detail in the following sections.

3.2.2 Multiple server models with a central queue

In this section we will discuss policies related to multiple server models with a central queue. We assume k homogeneous servers each of which has an exponentially distributed service time with rate μ. The load on the system will be denoted by ρ where $0 \leq \rho < k$. We also assume the setup time of the servers to be exponentially distributed with rate γ. Figure 7 illustrates the model of the system for which we will discuss policies proposed to study the energy-performance trade-off under the given assumptions.

BUSY/IDLE policy

When this policy is applied, all the k servers will be either in the *busy* or *idle* states, see [19, 16]. The BUSY/IDLE policy is the multiserver analogy of the static service speed policy mentioned for the single server case. As one can expect, the BUSY/IDLE policy gives the lower bound for delay but will waste energy while it is in the *idle* state.This wastage will be worsened if the system is lightly loaded. This policy can simply be modeled by a standard $M/M/K$ system from which the mean delay and energy consumption metrics can be computed accordingly [19].

BUSY/SLEEP policy

In this control policy, jobs are served in the busy state, and whenever a server finishes serving a job and there are no more jobs in the queue it goes in to a *sleep* state [16]. In this state, the energy consumption is considerably lower than that of the *idle* state. However, the server needs to be setup on the arrival of new jobs before it starts normal service.

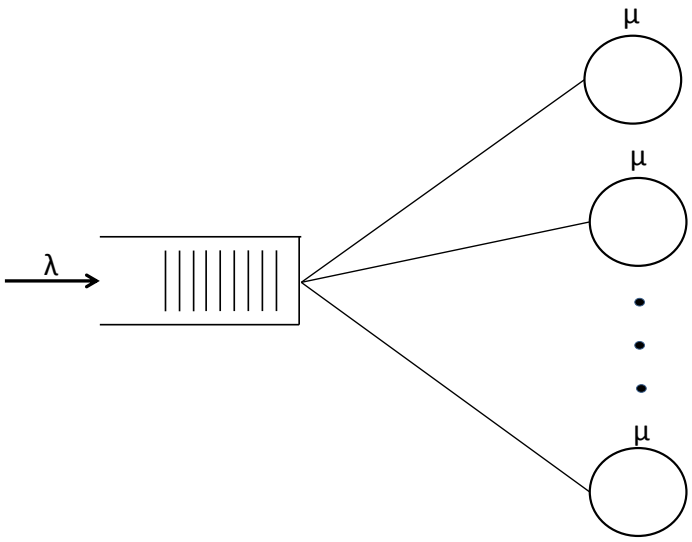


Figure 7: Multiserver system with central queue

BUSY/OFF policy

This policy can be regarded as the multi-server extension of the gated static speed policy studied for the single server systems. Under this policy, only busy servers are left on while all the idle ones are turned off instantly [19, 16, 15]. Hence, the servers will be in one of the *busy*, *off* or *setup* states and they will be turned on/off individually as opposed to another variant of this policy we will discuss shortly. When a new job arrives, it turns a switched off server on if it finds one, otherwise it will join the queue.

As already discussed earlier there will be a penalty for turning a server on in the form of extra delay and energy consumption. For a small setup time or lighter load, the BUSY/OFF policy outperforms the BUSY/IDLE policy while the advantage is turned around in the case both setup time and load are higher [19]. A more detailed explanation on how to choose between the BUSY/IDLE, BUSY/SLEEP and BUSY/OFF policies is also presented in [16].

DELAYEDOFF policy

The DELAYEDOFF policy, as studied in [16, 15] is a hybrid of the BUSY/IDLE and BUSY/OFF policies in which a server that becomes idle will wait for some time, t_{wait}, before being turned off. If a job is routed to this server during its idling time, its timer will be reset and the server will be working on the job. This policy can further be enhanced by routing the arriving jobs to the most recently busy (MRB) server [16].

BUSY/IDLE(t) policy

This policy, as proposed in [19], can also be seen as a hybrid of the BUSY/IDLE and BUSY/OFF policies. Under the BUSY/IDLE(t) policy we allow a certain number, t, of the servers to be in idle or on state. Once this threshold is reached, the next server that becomes idle is switched off. It is obvious that the mean delay, E[T], will decrease as t increases from 0 to k. However, the behavior of E[P] needs a closer observation as t goes from 0 to k. E[P] is found to decrease until it reaches some threshold value t^* and increases afterwards (this observation is done for a load value $\rho = k/2$). This can be reasoned as follows:

- For $t = 0$ the policy simply reduces to BUSY/OFF which is known to waste energy by turning on and off if the load on the system is not low enough
- For $t = k$ the policy becomes BUSY/IDLE which causes wastage of energy while the servers are not processing jobs.

Hence, the optimal value t^* gives the right amount of servers that consume the lowest energy. Therefore, depending on the weighting factor for energy in our cost function, the threshold can be chosen to be at t^* or somewhere near it. With this optimally choosen value of t, the BUSY/IDLE(t) policy is shown, in [19], to have a superior performance (in terms of both delay and energy consumption) than the BUSY/IDLE and BUSY/OFF policies.

BUSY/OFF with group turn-on policy

This is another variant of the BUSY/OFF policy studied in [35]. Contrary to the BUSY/OFF policy, servers are turned on/off in groups. For this purpose, the servers are divided into two groups: *base-line* and *reserved*. The base-line servers (k_b) are always on independent of the number of jobs in the system while the reserve servers will only be turned on if number of jobs in the system queue reaches a certain threshold, k_on. Once turned on, the reserve servers will remain in this state as long as the number of jobs in service does not drop below another threshold value k_off.

Assuming other traffic parameters to be constant, optimization of this policy can be done with respect to the $\{k_\mathrm{b}, k_\mathrm{on}, k_\mathrm{off}\}$ parameters. Pareto optimization of this policy is presented in [35] with the parameter values that result in the Pareto optimal values being further studied. As can be expected, the number of base-line servers, k_b, required decreases as the waiting time increases. This is found to happen in a batch-like character. That is, k_b decreases across non regular intervals of waiting time while it remains the same within these intervals. Another observation is that the threshold for turning servers on, k_b, shows a drastic increase within these intervals. Even though optimization of the system parameters under this policy is studied in [35], its performance compared to the other policies discussed above is not known.

BUSY/OFF with staggered setup policy

This is yet another variant of the BUSY/OFF policy in which only one server is allowed to be in the *setup* state at a time [19, 17]. This policy can particularly be helpful in systems where energy consumed in the setup state is considerably large because it limits the number of servers that can be started at the same time. Assuming that inter-arrival, service and setup times are exponentially distributed, the decomposition result given in (10) for the mean delay of a single server system still holds for this policy [18]. That is, for a multiserver controlled by this policy,

$$\mathrm{E}[T] = E[T_{M/M/k}] + \frac{1}{\gamma},$$

where $1/\gamma$ refers to the mean setup time. Furthermore, [17] shows that this decomposition result provides a very good approximation on the total delay for an $M/G/K$ system with an exponentially distributed setup time. Although the staggered setup mechanism may prevent excessive power from being drawn in the case of bursty arrivals, this advantage is diminished by the extra delay introduced [19]. Therefore, when both delay and energy metrics are considered in the system cost, this policy is outperformed by the others mentioned above.

3.2.3 Multiple server models with parallel queues

Up to this point, we have considered server farms with a single central queue. However, in a more practical setup, jobs will be routed instantly to the servers where they might have to queue independent of the queues of the other servers. Hence if we have k servers, we will also have k parallel queues feeding to these servers. Now we are faced with a new type of challenge, that is - how to assign an incoming job to one of the k queues. Some of these task assignment policies as discussed in [21] are:

- **Random:** As its name indicates, this policy assigns tasks to the k parallel queues in a random fashion.
- **Round Robin:** Tasks are assigned to the queues sequentially. In other words, an incoming task is assigned to the queue that has stayed the longest time without taking a task.
- **JSQ (Join Shortest Queue)**: An arriving task is routed to the queue that has the fewest number of jobs.
- **LWL (Least Work Left):** A task is routed to a queue where it is *supposed* to wait the least. Obviously, this works only for FIFO since knowing the amount of workload at present does not necessarily determine the amount of delay experienced by an arriving job.
- **SITA (Size Interval Task Assignment):** Each server takes care of jobs of a certain size interval only, incoming tasks are assigned to the corresponding server according to their size.

JSQ, LWL and SITA are known to be optimal under certain conditions when the metric in question is only performance, that is E[T]. But little is known about which one is optimal when both energy and performance should be considerded. For example, in the case where servers can be turned OFF if they are idle, would it benefit to use JSQ so that every arriving task that sees a server in the OFF state would try to start the server up, or should we route it to a server that is already serving customers and hence avoiding the energy penality of setting up the server in the OFF state?

Similar task assignment problems are discussed in [31]. The model considerations in this paper are multiple servers with their own queues, each of which has a constant speed, s, and a constant power consumption at this speed. Servers are turned off whenever they are idle. A weighted sum of power and mean number of jobs at each server is used as an objective function[2]. The following dispatching policies are discussed in this paper

- **Myopic Policy:** This policy tries to make dispatching decisions based only on the instantaneous effect it will have on the cost function, ignoring the future behavior of the system.
- **FPI dispatcher:** As opposed to the myopic policy, this policy takes the future behaviour of the system into account. This is achieved by trying to minimize the change in relative values caused by inserting the arriving customer to a queue.

An important result in this work is that the advantage of the FPI dispatcher over the myopic one is marginal.

[2]Note: [31] does not consider the setup delay introduced by turning a server off. However, [28] takes the setup delay in to consideration, providing a brief study on the parallel servers model.

4 Analysis of the turn-on threshold and idling time policy

In this section the results of [28] will be discussed in more detail as an extension of what is already discussed in Section 3. In addition, we will also extend the results of this work by considering the idling time to reset every time the system comes out of the idling state.

4.1 System Model

In the succeeding analysis, a single server system is considered, with jobs arriving according to a Poisson process with rate λ. The server has four different operational states, which are *off, setup, busy* and *idle*. These states are characterized by their power consumption values $P_{\text{off}} = 0$, P_{setup}, P_{busy} and P_{idle}. Once the system is in the *off* state, k jobs need to be accumulated before the server is put in the *setup* state. At the arrival of the k^{th} job, the server will enter the *setup* state and at the completion of setup, it will go into the *busy* state, in which the queued jobs are served according to the FIFO discipline. When there are no jobs to serve, the system goes into the *idle* state. Once the server is in the *idle* state, it will stay idle for a certain time interval and if no job arrives within this interval it will be turned off, thus, completing one cycle. This time interval will be named as *idling time* in the following sections. Initially, all the service, setup and idling times are assumed to have exponential distribution with $E[S], 1/\gamma$ and $1/\alpha$ being their respective mean values. However, these assumptions will be relaxed later on. Finally, transition from high energy state to low energy state is assumed to be instantaneous, while the reverse will incur setting up cost in the form of delay and energy consumed.

4.1.1 Notation

Throughout this thesis, the composition notation introduced in [28] will be used to denote the system being analyzed. This composition consists of two sets of Kendall's notation as $\{\}\circ\{\}$. The first set is just the standard Kendall's notation of the system while the second set is introduced due to the possibility to turn off the server. For example, $\{M/G/1\}\circ\{M/G/k\}$ would represent an $M/G/1$ system having an exponentially distributed setup time with an idling time that has a general distribution and a threshold of k jobs. Furthermore, service requests processed by the server will be referred to as *jobs*.

4.2 $\{M/M/1\} \circ \{M/M/k\}$ system

Due to the exponential assumptions, the system can be modeled as a Markov process and analyzed to yield the following result for the mean delay metric:

$$\mathrm{E}[T] = \mathrm{E}[T]_{M/M/1} + \frac{1}{\gamma}\frac{\alpha(\lambda + k\gamma)}{k\alpha\gamma + \alpha\lambda + \lambda\gamma} + \frac{1}{2\lambda}\frac{k\alpha\gamma(k-1)}{k\alpha\gamma + \alpha\lambda + \lambda\gamma}, \tag{12}$$

which reduces to (10) by letting the turn-on threshold $k = 1$ and $\alpha \to \infty$. On the other hand, if we set $\alpha = 0$ keeping $k = 1$, the mean delay of the system will reduce to that of a standard $M/M/1$. Thus, for any other combination of α and γ, keeping $k = 1$ will bound the delay as

$$\mathrm{E}[T_{M/M/1}] \leq \mathrm{E}[T] \leq \mathrm{E}[T_{M/M/1}] + \frac{1}{\gamma}.$$

The mean power consumption is then computed based on the limiting probabilities as

$$\mathrm{E}[P] = \rho P_{\text{busy}} + \frac{(1-\rho)\lambda}{k\alpha\gamma + \alpha\lambda + \lambda\gamma}(P_{\text{idle}}\gamma + P_{\text{setup}}\alpha). \tag{13}$$

In addition, the mean switching rate, which is just the inverse of the mean busy cycle of the server $\mathrm{E}[C]$, is given as

$$\mathrm{E}[C]^{-1} = (1-\rho)\frac{\alpha\gamma\lambda}{\lambda\gamma + \alpha\lambda + \alpha\gamma k}. \tag{14}$$

The mean delay and mean power consumption of the single server system are given by (12) and (13) respectively. However, for a standard $M/M/1$ system the mean power consumption is computed as $\mathrm{E}[P_{M/M/1}] = \rho P_{\text{busy}} + (1-\rho)P_{\text{idle}}$. Substituting $\rho P_{\text{busy}} = \mathrm{E}[P_{M/M/1}] - (1-\rho)P_{\text{idle}}$ in (13) we will have,

$$\mathrm{E}[P] = \mathrm{E}[P_{M/M/1}] + \frac{(1-\rho)\alpha}{k\alpha\gamma + \alpha\lambda + \lambda\gamma}(\lambda P_{\text{setup}} - (\lambda + k\gamma)P_{\text{idle}}). \tag{15}$$

One can note that letting $\alpha = 0$ will reduce the system to a standard $M/M/1$ system. However, when

$$P_{\text{idle}} > \frac{\lambda}{\lambda + k\gamma}P_{\text{setup}},$$

the second term of (15) will become negative which means we can save energy by turning the server off while it is idle. In this case, letting $\alpha \to \infty$ will give the maximum possible magnitude for the second term, which in turn gives the minimum power consumption. For the case where

$$P_{\text{idle}} < \frac{\lambda}{\lambda + k\gamma}P_{\text{setup}},$$

the second term will become positive. Hence, the smallest possible value is when $\alpha = 0$ which will leave us with the power consumption of a standard $M/M/1$. In this

case, turning the server off when it is idle will only increase the energy consumption because the setup state consumes considerably more energy than the idle state. Another observation which can be made from (13) is that when $\rho \to 1$, the second term will diminish which is an intuitive result since the system will rarely be idle for a larger load. This also means it will rarely be in the setup state since it will not be turned off in the first place.

4.2.1 Optimization with respect to the threshold and idling time values

With the closed form expressions already determined for $\mathrm{E}[T]$, $\mathrm{E}[P]$ and $\mathrm{E}[C^{-1}]$ optimization can be done using the cost function given in (9). Giving appropriate values to the weights assigned to each metric, we can get a simple cost function such as

$$f = \mathrm{E}[T] + \beta \mathrm{E}[E] + \beta^{'} \mathrm{E}[C]^{-1}. \tag{16}$$

For example, setting $W_{T,1} = 1$ and $W_{T,i} = 0$ for all the remaining terms will result in the first term of (16) [3]. As discussed earlier, the optimal values for α depend on the sign of the second term in $\mathrm{E}[P]$. That is, for

$$P_{\text{idle}} < \frac{\lambda}{\lambda + k\gamma} P_{\text{setup}},$$

setting $\alpha = 0$ will minimize our cost function given in (16). In this case, since the server is never going to be switched off, there is no need to optimize with respect to k. For the other case where

$$P_{\text{idle}} > \frac{\lambda}{\lambda + k\gamma} P_{\text{setup}},$$

while $\alpha = 0$ will still give minimal results for $\mathrm{E}[T]$ and $\mathrm{E}[C]^{-1}$, $\mathrm{E}[E]$ will suffer from it. Hence performing a partial derivative of the cost function with respect to α gives us

$$\begin{aligned}\frac{\partial f}{\partial \alpha} = \frac{\lambda(\lambda + k\gamma)}{(k\alpha\gamma + \alpha\lambda + \lambda\gamma)^2} + \frac{\gamma^2 k(k-1)}{2(k\alpha\gamma + \alpha\lambda + \lambda\gamma)^2} + \\ \frac{\beta}{\lambda}((1-\rho)\gamma \frac{P_{\text{setup}}\lambda - P_{\text{idle}}(\lambda + k\gamma)}{(k\alpha\gamma + \alpha\lambda + \lambda\gamma)^2}) + \\ \beta^{'}((1-\rho)\frac{\lambda^2\gamma^2}{(k\alpha\gamma + \alpha\lambda + \lambda\gamma)^2})\end{aligned}$$

Parameter α exists only in the denominator of each term. So as long as the partial derivative stays positive, that is as long as the inequality

$$\beta(1-\rho)P_{\text{idle}}(1 + \frac{k\gamma}{\lambda})\gamma \leq \lambda(\lambda + k\gamma) + \gamma^2 \frac{k(k-1)}{2} + \beta(1-\rho)\gamma P_{\text{setup}} + \beta^{'}(1-\rho)\lambda^2\gamma^2$$

[3] Note that in the second term, $\mathrm{E}[E] = \mathrm{E}[P]/\lambda$.

holds, $\alpha = 0$ minimizes the cost function. If this inequality does not hold then $\alpha \to \infty$ will be the optimal choice.

To determine the optimal value for k, $\alpha \to \infty$ is assumed and partial derivative of the resulting cost function is done with respect to k which gives us the following quadratic equation,

$$\frac{\gamma^2}{2\lambda}k^2 + \gamma k - \left(\frac{\gamma}{2} + (1-\rho)\lambda\gamma(\beta\frac{P_{\text{setup}}}{\lambda} + \beta'\gamma)\right) = 0.$$

Solving this quadratic equation will be the final task of the optimization with respect to the idling time and turning-on threshold under the given assumptions.

The above analysis, based on the weighted sum cost function, showed that the optimal policy for the $\{M/M/1\} \circ \{M/M/k\}$ system is to either switch the server off immediately when it becomes idle or to leave it idle until the next job arrives. In [16] a similar analysis showed that this result also holds for the Energy-Delay-Product (EDP) cost function.

4.3 $\{M/G/1\} \circ \{G/G/k\}$ system

Up to this point, we have covered the analysis of the system based on the exponential assumptions for service time, setup and idling times in order to model the system as a Markov process. However, a more general approach is also introduced in [28] by modeling the behavior of the system in the long run, which the authors named the work-cycle analysis of the system. In this analysis there are three main assumptions, which are: jobs arrive to the system as a Poisson process, queued jobs are served according to the FIFO service discipline and the idling time will be drawn once and will be remembered if a job arrives before it expires.

In this analysis, the service, setup and idling times are generally distributed. Hence, the system can be modeled as an M/G/1 queue with generally distributed setup and idling times and turn on threshold. While mean values for energy and switching rate were obtained based on a general assumption in [28], exponentially distributed setup time had to be assumed in the mean delay analysis of the system. However, we will relax this assumption on setup time to general distributions in this work.

In one complete work-cycle, the system goes through the *off* and *setup* states, iterates between the *busy* and *idle* states and finally goes back to the *off* state. If we represent the length of a single work-cycle by C, then obviously

$$C = T_{\text{off}} + T_{\text{setup}} + T_{\text{busy}} + T_{\text{idle}},$$

where $T_{\text{off}}, T_{\text{setup}}, T_{\text{busy}}$ and T_{idle} are random variables represeting time spent in the respective states. Therefore, the mean length of the work cycle can be given as

$$\text{E}[C] = \text{E}[T_{\text{off}}] + \text{E}[T_{\text{setup}}] + \text{E}[T_{\text{busy}}] + \text{E}[T_{\text{idle}}].$$

It is clear that the mean time spent in each of the *off, setup* and *idle* states is

$$\mathrm{E}[T_{\mathrm{off}}] = \frac{k}{\lambda},$$

$$\mathrm{E}[T_{\mathrm{setup}}] = \frac{1}{\gamma},$$

$$\mathrm{E}[T_{\mathrm{idle}}] = \frac{1}{\alpha}.$$

To determine the fraction of time spent in each state let us denote each proportion as π_{off}, π_{setup}, π_{busy} and π_{idle}. Then,

$$\begin{aligned}
\pi_{\mathrm{off}} &= \frac{\mathrm{E}[T_{\mathrm{off}}]}{\mathrm{E}[C]} = \frac{\frac{k}{\lambda}}{\mathrm{E}[C]}, \\
\pi_{\mathrm{setup}} &= \frac{\mathrm{E}[T_{\mathrm{setup}}]}{\mathrm{E}[C]} = \frac{\frac{1}{\gamma}}{\mathrm{E}[C]}, \\
\pi_{\mathrm{idle}} &= \frac{\mathrm{E}[T_{\mathrm{idle}}]}{\mathrm{E}[C]} = \frac{\frac{1}{\alpha}}{\mathrm{E}[C]}, \\
\pi_{\mathrm{busy}} &= \frac{E[T_{\mathrm{busy}}]}{\mathrm{E}[C]} = \rho.
\end{aligned}$$

Adding up all the proportions we will have

$$\rho + \frac{\frac{k}{\lambda}}{\mathrm{E}[C]} + \frac{\frac{1}{\gamma}}{\mathrm{E}[C]} + \frac{\frac{1}{\alpha}}{\mathrm{E}[C]} = 1. \tag{17}$$

Rearranging this we will get

$$\mathrm{E}[C] = \frac{\lambda\gamma + \alpha\lambda + \alpha\gamma k}{\lambda\alpha\gamma(1-\rho)}.$$

Once the mean length of the work cycle is determined, the mean delay of the system can be studied using a heuristic treatment as discussed in [29] for M/G/1-FIFO vacation models. The mean delay, $\mathrm{E}[T]$, can be given as

$$\mathrm{E}[T] = \mathrm{E}[S] + \mathrm{E}[W],$$

where S and W are the service and waiting time random variables. Furthermore, let ω denote the state of the system at the time a test customer arrives. Obviously, the set of possible values for ω is $\{\mathrm{off}, \mathrm{setup}, \mathrm{busy}, \mathrm{idle}\}$.

Hence, the waiting time of an arbitrary test job can be given as

$$W = \sum_{i=1}^{N_W} S_i + S^R 1_{\{\omega=\mathrm{busy}\}} + (T_{\mathrm{off}}^R + T_{\mathrm{setup}}) 1_{\{\omega=\mathrm{off}\}} + T_{\mathrm{setup}}^R 1_{\{\omega=\mathrm{setup}\}}, \tag{18}$$

where N_W denotes the mean number of jobs in the queue waiting for service and the superscript R denotes the remaining lifetime of the respective variable. For example, S^R is the remaining service time of the job in service at the arrival time of the test job. In the steady state, indicator functions in (18) can be replaced by the

probabilities of being in the respective states. Clearly, these probabilities are just the proportions discussed above. Therefore, the mean waiting time of the system is

$$\mathrm{E}[W] = E[N_W]E[S] + \pi_{\text{busy}}E[S^R] + \pi_{\text{off}}(E[T^R_{\text{off}}] + E[T_{\text{setup}}]) + \pi_{\text{setup}}E[T^R_{\text{setup}}]. \quad (19)$$

Utilizing Little's formula, the mean number of waiting jobs, $E[N_W]$, can be re-written as $E[N_W] = \lambda \mathrm{E}[W]$. Now the remaining work is to determine the mean residual lifetimes in each state. The mean remaining service time, given that the server is busy, is known to be

$$E[S^R] = \frac{E[S^2]}{2E[S]}.$$

The mean remaining time in the *off* state can be determined by considering the mean interarrival times of the number of threshold jobs required to accumulate before the server is put in the *setup* state. This value can be calculated as

$$E[T^R_{\text{off}}] = \frac{1}{k}\left(\frac{k-1}{\lambda} + \frac{k-2}{\lambda} + \frac{k-3}{\lambda} + \cdots + \frac{1}{\lambda}\right) = \frac{k-1}{2\lambda}.$$

To determine the mean remaining setup time, let us regard this state as a special job that arrives at the beginning of every work cycle of the server as illustrated in Figure 8. As already given earlier for the remaining service time, the mean remaining setup time can be given by

$$\mathrm{E}[T^{\mathrm{R}}_{\text{setup}}] = \frac{E[T^2_{\text{setup}}]}{2E[T_{\text{setup}}]}$$

With all the required values derived, the $\mathrm{E}[W]$ will read as

$$\mathrm{E}[W] = \frac{\lambda E[S^2]}{2(1-\rho)} + \frac{\lambda\alpha\gamma}{k\alpha\gamma + \alpha\lambda + \lambda\gamma}\left(\frac{k}{\lambda}\left(\frac{k-1}{2\lambda} + \frac{1}{\gamma}\right) + \frac{E[T^2_{\text{setup}}]}{2}\right).$$

Using this result to calculate the total mean delay of a job will yield,

$$\mathrm{E}[T] = \mathrm{E}[T_{M/G/1}] + \frac{\lambda\alpha\gamma}{k\alpha\gamma + \alpha\lambda + \lambda\gamma}\left(\frac{k}{\lambda}\left(\frac{k-1}{2\lambda} + \frac{1}{\gamma}\right) + \frac{E[T^2_{\text{setup}}]}{2}\right). \quad (20)$$

From (20) one can see that setting $\alpha = 0$ will simply give the mean delay of a standard M/G/1-FIFO system. Furthermore, if the server is turned off immediately after becoming idle and it can be turned on instantly on the arrival of a job, that is if $\alpha \to \infty$ and $\gamma \to \infty$, then (20) will reduce to

$$\mathrm{E}[T] = \mathrm{E}[T_{M/G/1}] + \frac{k-1}{2\lambda},$$

which is just the standard delay plus the time required to accumulate the threshold number of jobs. Setting $k = 1$ in this case will again leave us with the mean delay of a standard M/G/1-FIFO system.

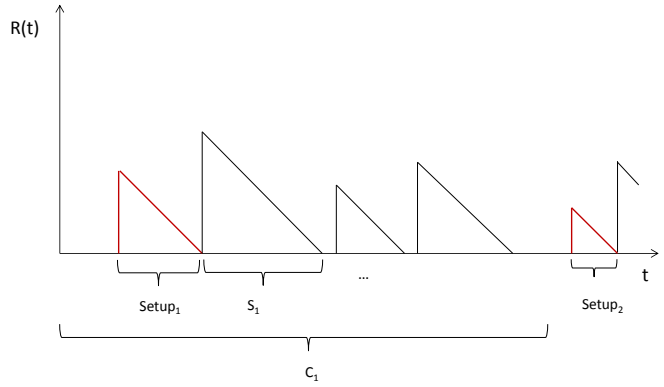


Figure 8: Mean remaining service time of the system considering the setup time as a special job at the beginning of each cycle. The setup jobs are drawn in red.

Using the appropriate proportions, the mean power consumption can be computed as

$$\mathrm{E}[P] = \rho P_{\text{busy}} + \pi_{\text{setup}} P_{\text{setup}} + \pi_{\text{idle}} P_{\text{idle}}, \tag{21}$$

which will give the same result as (13). The mean switching rate, which is just the inverse of $\mathrm{E}[C]$, will also have the same form as (14).

Theorem 1. *For an $\{M/G/1\}\circ\{G/G/k\}$ the mean delay, mean power consumption and mean switching rate metrics are given by:*

$$\mathrm{E}[T] = \mathrm{E}[T_{M/G/1}] + \frac{\lambda\alpha\gamma}{k\alpha\gamma + \alpha\lambda + \lambda\gamma}\left(\frac{k}{\lambda}\left(\frac{k-1}{2\lambda} + \frac{1}{\gamma}\right) + \frac{E[T^2_{\text{setup}}]}{2}\right),$$

$$\mathrm{E}[P] = \mathrm{E}[P_{M/M/1}] + \frac{(1-\rho)\alpha}{k\alpha\gamma + \alpha\lambda + \lambda\gamma}(\lambda P_{\text{setup}} - (\lambda + k\gamma)P_{\text{idle}})$$

and

$$\mathrm{E}[C]^{-1} = (1-\rho)\frac{\alpha\gamma\lambda}{\lambda\gamma + \alpha\lambda + \alpha\gamma k}$$

provided that idling time is ***remembered****.*

Remark 1. *The mean power consumption and mean switching rates are insensitive to the service, setup and idling time distributions. One can see that the mean delay of the system is sensitive to the setup time distribution in addition to the well known sensitivity of FIFO systems to service time distribution.*

Rearranging the mean delay equation in Theorem 1 will result in the following remark.

Remark 2. *The mean delay of an* $\{M/G/1\} \circ \{G/G/k\}$ *system, that applies the FIFO service discipline with the idling time remembered, can be rewritten as*

$$\mathrm{E}[T] = \mathrm{E}[T_{M/G/1}] + \frac{\frac{k(k-1)}{2}\frac{1}{\lambda} + k\mathrm{E}[T_{\text{setup}}] + \frac{\lambda}{2}\mathrm{E}[T_{\text{setup}}^2]}{\lambda\mathrm{E}[T_{\text{idle}}] + k + \lambda\mathrm{E}[T_{\text{setup}}]}.$$

It can be seen from Theorem 1 and Remark 2 that $\mathrm{E}[T_{\text{idle}}] \to \infty$, or $\alpha = 0$, will minimize $\mathrm{E}[T]$ and $\mathrm{E}[C^{-1}]$ while its effect on $\mathrm{E}[P]$ depends on the sign of the second term in the $\mathrm{E}[P]$ equation. This brings us to the next corollary:

Corollary 1. *For an* $\{M/G/1\} \circ \{G/G/k\}$ *system with a remembered idling time, the optimal idling time value is either* $T_{\text{idle}} \to \infty$ *or* $T_{\text{idle}} = 0$*. The rule for choosing between these values is:*

1. *If* $P_{\text{idle}} < \frac{\lambda}{\lambda + k\gamma} P_{\text{setup}}$*, then* $T_{\text{idle}} \to \infty$ *is the optimal value*
2. *If* $P_{\text{idle}} > \frac{\lambda}{\lambda + k\gamma} P_{\text{setup}}$ *and the first derivative of the weighted sum cost function with respect to* α *is positive, then* $T_{\text{idle}} \to \infty$*, or* $\alpha = 0$ *is the optimal choice. If the derivative is negative then* $T_{\text{idle}} = 0$*, or* $\alpha \to \infty$ *is the optimal choice.*

4.3.1 Optimization with deterministic setup time

Corollary 1 provides the general set of rules for choosing the optimal control policy. Assuming a deterministic setup time, these rules can further be refined explicitly. In this case, the cost function given in (16) will read as

$$\begin{aligned} f = {} & \mathrm{E}[T_{M/G/1}] + \frac{\lambda\alpha\gamma}{k\alpha\gamma + \alpha\lambda + \lambda\gamma}\left(\frac{k}{\lambda}\left(\frac{k-1}{2\lambda} + \frac{1}{\gamma}\right) + \frac{1}{2\gamma^2}\right) + \\ & \frac{\beta}{\lambda}\left(\mathrm{E}[P_{M/M/1}] + \frac{\alpha(1-\rho)}{k\alpha\gamma + \alpha\lambda + \lambda\gamma}(\lambda P_{\text{setup}} - (\lambda + \gamma k)P_{\text{idle}})\right) + \\ & \beta'\left((1-\rho)\frac{\alpha\lambda\gamma}{k\alpha\gamma + \alpha\lambda + \lambda\gamma}\right). \end{aligned}$$

Taking the partial derivative with respect to α,

$$\begin{aligned} \frac{\partial f}{\partial \alpha} = {} & \frac{\gamma\lambda}{(k\alpha\gamma + \alpha\lambda + \lambda\gamma)^2}\left(k\gamma\left(\frac{k-1}{2\lambda} + \frac{1}{\gamma}\right) + \frac{\lambda}{2\gamma} + \right. \\ & \left. \frac{\beta}{\lambda}(1-\rho)(\lambda P_{\text{setup}} - (\lambda + k\gamma)P_{\text{idle}}) + \beta'\gamma\lambda(1-\rho)\right). \end{aligned}$$

All α values reside in the denominator of the first derivative. Hence, if

$$\frac{\partial f}{\partial \alpha} > 0,$$

that is if

$$k\gamma(\frac{k-1}{2\lambda}+\frac{1}{\gamma})+\frac{\lambda}{2\gamma}+\beta(1-\rho)P_{\text{setup}}+\beta^{'}\gamma\lambda(1-\rho)>\beta(1-\rho)(1+\frac{k\gamma}{\lambda})P_{\text{idle}},$$

then f is a non-decreasing function of α. Therefore setting $\alpha = 0$ ($T_{\text{idle}} \to \infty$) will be optimal. On the other hand, if

$$\frac{\partial f}{\partial \alpha} < 0,$$

that is if

$$k\gamma(\frac{k-1}{2\lambda}+\frac{1}{\gamma})+\frac{\lambda}{2\gamma}+\beta(1-\rho)P_{\text{setup}}+\beta^{'}\gamma\lambda(1-\rho)<\beta(1-\rho)(1+\frac{k\gamma}{\lambda})P_{\text{idle}},$$

the cost function is decreasing with α, hence, making $\alpha \to \infty$ ($T_{\text{idle}} = 0$) the optimal choice.

Corollary 2. *For an $\{M/G/1\} \circ \{D/G/k\}$ system with a remembered idling time:*

1. *If $P_{\text{idle}} < \frac{\lambda}{\lambda+k\gamma}P_{\text{setup}}$, then BUSY/IDLE policy ($T_{\text{idle}} \to \infty$) is the optimal control policy.*
2. *If $P_{\text{idle}} > \frac{\lambda}{\lambda+k\gamma}P_{\text{setup}}$ and*

   $$k\gamma(\frac{k-1}{2\lambda}+\frac{1}{\gamma})+\frac{\lambda}{2\gamma}+\beta(1-\rho)P_{\text{setup}}+\beta^{'}\gamma\lambda(1-\rho)>\beta(1-\rho)(1+\frac{k\gamma}{\lambda})P_{\text{idle}},$$

   then BUSY/IDLE policy ($T_{\text{idle}} \to \infty$) will still be optimal.

   On the other hand, if

   $$k\gamma(\frac{k-1}{2\lambda}+\frac{1}{\gamma})+\frac{\lambda}{2\gamma}+\beta(1-\rho)P_{\text{setup}}+\beta^{'}\gamma\lambda(1-\rho)<\beta(1-\rho)(1+\frac{k\gamma}{\lambda})P_{\text{idle}},$$

   then BUSY/OFF policy ($T_{\text{idle}} = 0$) will be optimal.

4.4 The impact of resetting idling time

The mean value equations determined up to this point are based on an assumption that the idling time of a server is remembered. That is, once a server becomes idle, it will wait for a certain amount of time, with mean $\frac{1}{\alpha}$, and turn off if there is no arrival. If a job arrives before the idling time expires, the residual idling time is remembered for the next idle period. However, it would be more realistic if the idling time is reset every time the idling period is interrupted by an arriving job. Therefore, we will extend the results in [28] to incorporate such scenarios.

The length of every idle period in a single busy cycle of a work cycle is given by

$$\min\{T_{\text{idle}}, A\},$$

where T_{idle} is a random variable representing the idling time and A is a random variable representing the interarrival time which is exponentially distributed with rate λ. If $\min\{T_{\text{idle}}, A\} = A$, then an arrival has occurred before the server's idling time expires. Otherwise, if $\min\{T_{\text{idle}}, A\} = T_{\text{idle}}$, the idling time of the server expires before the next arrival. This will determine the end of the current work cycle as the server is switched off. Let N denote the number of idle periods in a work cycle. Then it is clear that it has a geometric distribution with success probability

$$p = P\{T_{\text{idle}} < A\} = \int_0^\infty P\{T_{\text{idle}} < x\}\lambda e^{-\lambda x} dx.$$

Hence, the probability of having n idle periods is given by

$$P\{N = n\} = p(1-p)^{n-1}.$$

Now let us denote the total length of all the idle periods as I_T, which can be calculated as

$$I_T = \sum_{i=1}^{N} \min\{T_{\text{idle}}^i, A_i\}.$$

Since N is a stopping time for the work cycle process, we have

$$\mathrm{E}[I_T] = E[\min\{T_{\text{idle}}, A\}]E[N]. \tag{22}$$

Since N has a geometric distribution, we know that

$$E[N] = \frac{1}{p}.$$

To determine $E[\min\{T_{\text{idle}}, A\}]$, we will use the property $P\{\min\{T_{\text{idle}}, A\} > t\} = P\{T_{\text{idle}} > t\}P\{A > t\}$ because of the independence of T_{idle} and A. Hence,

$$E[\min\{T_{\text{idle}}, A\}] = \int_0^\infty P\{T_{\text{idle}} > t\}P\{A > t\}dt = \int_0^\infty P\{T_{\text{idle}} > t\}e^{-\lambda t}dt.$$

Adopting the total mean idling time of (22) into the time proportion calculations will yield the general form of the mean delay, power and switching rate metrics as given by the following theorem.

Theorem 2. *For an $\{M/G/1\} \circ \{G/G/k\}$ system the mean delay, mean power consumption and mean switching rate metrics are given by:*

$$\mathrm{E}[T] = \mathrm{E}[T_{M/G/1}] + \frac{\lambda\gamma}{\lambda\gamma\mathrm{E}[I_T] + \lambda + k\gamma}\left(\frac{k}{\lambda}\left(\frac{k-1}{2\lambda} + \frac{1}{\gamma}\right) + \frac{E[T_{\text{setup}}^2]}{2}\right),$$

$$\mathrm{E}[P] = \mathrm{E}[P_{M/M/1}] + (1-\rho)\frac{1}{\lambda\gamma\mathrm{E}[I_T] + \lambda + k\gamma}(\lambda P_{\text{setup}} - (\lambda + \gamma k)P_{\text{idle}}),$$

and

$$\mathrm{E}[C]^{-1} = (1-\rho)\frac{\lambda\gamma}{\lambda\gamma\mathrm{E}[I_T] + \lambda + k\gamma}$$

provided that idling time is ***reset*** *everytime the system comes out of the idle state.*

Remark 3. *The mean value metrics exhibit the same sensitivity property as given by Remark 1.*

Similar to the case in which the idling time is remembered, this will lead to the following remark.

Remark 4. *The mean delay of an* $\{M/G/1\} \circ \{G/G/k\}$ *system, that utilizes the FIFO service discipline with the idling time* ***reset*** *every time the server goes out of the idle state, is given by*

$$\mathrm{E}[T] = \mathrm{E}[T_{M/G/1}] + \frac{\frac{k(k-1)}{2}\frac{1}{\lambda} + k\mathrm{E}[T_{\text{setup}}] + \frac{\lambda}{2}\mathrm{E}[T_{\text{setup}}^2]}{\lambda\mathrm{E}[I_T] + k + \lambda\mathrm{E}[T_{\text{setup}}]}.$$

From Remark 4, we can see that setting $k = 1$ and $\mathrm{E}[I_T] = 0$ will give the same mean delay as given in (11) for a gated system. On the other hand, based on Theorem 2 and Remark 4 we can see that $I_T \to \infty$ minimizes the mean delay and switching rate while its effect on the power metric depends on the sign of the second term. This will lead us to the following corollary.

Corollary 3. *For an* $\{M/G/1\} \circ \{G/G/k\}$ *system that* ***resets*** *the idling time, the optimal idling time value is still either* $T_{\text{idle}} \to \infty$ *or* $T_{\text{idle}} = 0$*. The rule for choosing between these values is:*

1. *If* $P_{\text{idle}} < \frac{\lambda}{\lambda+k\gamma}P_{\text{setup}}$*, then* $T_{\text{idle}} \to \infty$ *is the optimal value*
2. *If* $P_{\text{idle}} > \frac{\lambda}{\lambda+k\gamma}P_{\text{setup}}$ *and the first derivative of the weighted sum cost function with respect to* α *is positive, then* $T_{\text{idle}} \to \infty$*, or* $\alpha = 0$ *is the optimal choice. If the derivative is negative then* $T_{\text{idle}} = 0$*, or* $\alpha \to \infty$ *is the optimal choice.*

4.4.1 Exponentially distributed idling time

If idling time is assumed to be exponentially distributed with rate α, then $p = \frac{\alpha}{\lambda+\alpha}$ and

$$E[\min\{T_{\text{idle}}, A\}] = \int_0^\infty e^{-\alpha t}e^{-\lambda t}dt = \frac{1}{\alpha+\lambda}.$$

Substituting this in (22) the total idling period will be,

$$\mathrm{E}[I_T] = \frac{1}{\alpha+\lambda}\frac{\alpha+\lambda}{\alpha} = \frac{1}{\alpha}.$$

We know this to be true due to the memoryless property of exponential distributions. Substituting this value in **Theorem 2** will give us the mean value equations given in **Theorem 1** since resetting of the idling time will not have any effect in this case.

4.4.2 Deterministic idling time

Now let us assume that when the server becomes idle, it will wait for a deterministic amount of time, $\frac{1}{\alpha}$, and switch off if there are no jobs arriving within this interval. If a job arrives within this interval, then the idling time is reset and the server starts serving the job. In this case, since the length of the idling period is known, the probability of the idling period expiring before the next arrival is simply

$$p = P\{A > \frac{1}{\alpha}\} = e^{-\frac{\lambda}{\alpha}}.$$

Substituting this in the $E[N]$ equation and using it to determine total idling period, we will get

$$\mathrm{E}[I_T] = e^{\lambda/\alpha} \int_0^{\frac{1}{\alpha}} e^{-\lambda t} dt = \frac{1}{\lambda}(e^{\lambda/\alpha} - 1) \tag{23}$$

From (23), one can see that as $\alpha \to \infty$, $\mathrm{E}[I_T] \to 0$ and as $\alpha \to 0$, $\mathrm{E}[I_T] \to \infty$, both of which are intuitive results. Now by Theorem 2, we will get the following result for the mean switching rate:

$$\mathrm{E}[C]^{-1} = (1-\rho)\frac{\lambda\gamma}{\gamma(e^{\lambda/\alpha} - 1) + \lambda + k\gamma}.$$

Using (23) to calculate the mean power consumption will result in

$$\mathrm{E}[P] = \mathrm{E}[P_{M/M/1}] + \frac{(1-\rho)}{\gamma(e^{\lambda/\alpha} - 1) + \lambda + k\gamma}(\lambda P_{\text{setup}} - (\lambda + \gamma k)P_{\text{idle}}). \tag{24}$$

The mean power consumption reduces to that of a standard $M/M/1$ system at $\alpha = 0$ while for $\alpha \to \infty$ it will become

$$\mathrm{E}[P] = \mathrm{E}[P_{M/M/1}] - (1-\rho)(P_{\text{idle}} - \frac{\lambda}{\lambda + \gamma k}P_{\text{setup}}).$$

Similarly, the mean delay can be calculated by substituting the total idling time in Theorem 2 and it will read as

$$\mathrm{E}[T] = \mathrm{E}[T_{M/G/1}] + \frac{\lambda\gamma}{\gamma(e^{\lambda/\alpha} - 1) + \lambda + k\gamma}\left(\frac{k}{\lambda}(\frac{k-1}{2\lambda} + \frac{1}{\gamma}) + \frac{E[T^2_{\text{setup}}]}{2}\right). \tag{25}$$

From (25) it is clear that for $\alpha = 0$ the second term will disappear which makes the delay to be that of a standard $M/G/1$ system. On the other hand, for $\alpha \to \infty$, the mean delay will reduce to

$$\mathrm{E}[T] = \mathrm{E}[T_{M/G/1}] + \frac{\lambda\gamma}{\lambda + k\gamma}\left(\frac{k}{\lambda}(\frac{k-1}{2\lambda} + \frac{1}{\gamma}) + \frac{E[T^2_{\text{setup}}]}{2}\right).$$

4.4.3 Optimization of a $\{M/G/1\} \circ \{D/D/k\}$ system

Here, we assume deterministic idling and setup times equal to $\frac{1}{\alpha}$ and $\frac{1}{\mu}$, respectively. The system level cost function, using the form given in (16), can be written as

$$f = \mathrm{E}[T_{M/G/1}] + \frac{\lambda\gamma}{\gamma(e^{\lambda/\alpha}-1)+\lambda+k\gamma}\left(\frac{k}{\lambda}(\frac{k-1}{2\lambda}+\frac{1}{\gamma})+\frac{1}{2\gamma^2}\right)+ \\ \frac{\beta}{\lambda}\left(\mathrm{E}[P_{M/M/1}]+\frac{(1-\rho)}{\gamma(e^{\lambda/\alpha}-1)+\lambda+k\gamma}(\lambda P_{\text{setup}}-(\lambda+\gamma k)P_{\text{idle}})\right)+ \\ \beta'\left((1-\rho)\frac{\lambda\gamma}{\gamma(e^{\lambda/\alpha}-1)+\lambda+k\gamma}\right).$$

Taking the partial derivative of the cost function with respect to α gives us

$$\frac{\partial f}{\partial \alpha} = \frac{\gamma\lambda e^{\lambda/\alpha}}{\alpha^2(\gamma(e^{\lambda/\alpha}-1)+\lambda+k\gamma)^2}\left(k\gamma(\frac{k-1}{2\lambda}+\frac{1}{\gamma})+\frac{\lambda}{2\gamma}+ \right. \\ \left. \frac{\beta}{\lambda}(1-\rho)(\lambda P_{\text{setup}}-(\lambda+k\gamma)P_{\text{idle}})+\beta'\gamma\lambda(1-\rho)\right).$$

Once again, all the α values are in the denominator, meaning no α value can equate the partial derivative to zero. Therefore, if

$$\frac{\partial f}{\partial \alpha} > 0,$$

that is, if

$$k\gamma(\frac{k-1}{2\lambda}+\frac{1}{\gamma})+\frac{\lambda}{2\gamma}+\beta(1-\rho)\lambda P_{\text{setup}}+\beta'\gamma\lambda(1-\rho) > \beta(1-\rho)(\lambda+k\gamma)P_{\text{idle}},$$

then f is a non-decreasing function of α. Therefore setting $\alpha = 0$ will be optimal. In the other case where

$$k\gamma(\frac{k-1}{2\lambda}+\frac{1}{\gamma})+\frac{\lambda}{2\gamma}+\beta(1-\rho)\lambda P_{\text{setup}}+\beta'\gamma\lambda(1-\rho) < \beta(1-\rho)(\lambda+k\gamma)P_{\text{idle}},$$

setting $\alpha \to \infty$ will be optimal.

Corollary 4. *For an $\{M/G/1\} \circ \{D/D/k\}$ system with the idling time reset every time the server comes out of the idle state:*

1. *If $P_{\text{idle}} < \frac{\lambda}{\lambda+k\gamma}P_{\text{setup}}$, then $T_{\text{idle}} \to \infty$ (BUSY/IDLE policy) is the optimal choice.*
2. *If $P_{\text{idle}} > \frac{\lambda}{\lambda+k\gamma}P_{\text{setup}}$ and*

$$k\gamma(\frac{k-1}{2\lambda}+\frac{1}{\gamma})+\frac{\lambda}{2\gamma}+\beta(1-\rho)P_{\text{setup}}+\beta'\gamma\lambda(1-\rho) > \beta(1-\rho)(1+\frac{k\gamma}{\lambda})P_{\text{idle}},$$

then $T_{\text{idle}} \to \infty$ *(BUSY/IDLE policy) will still be optimal.*

On the other hand, if

$$k\gamma(\frac{k-1}{2\lambda}+\frac{1}{\gamma})+\frac{\lambda}{2\gamma}+\beta(1-\rho)P_{\text{setup}}+\beta^{'}\gamma\lambda(1-\rho)<\beta(1-\rho)(1+\frac{k\gamma}{\lambda})P_{\text{idle}},$$

then $T_{\text{idle}}=0$ *(BUSY/OFF policy) will be optimal.*

One can note that the conditions given in Corollary 4 for determining the optimal control policy are identical to those given in Corollary 2, where the idling time of the server is remembered. Hence, we can conclude that regardless to whether the idling time is remembered or not, the optimal control policy for a single server system should either leave the server idle or turn it off immediately when there is no job to serve.

4.5 $\{M/G/1\} \circ \{G/G/k\}$ conclusions

As discussed above, the rules for choosing the optimal control policy remain the same irrespective of whether the idling time is remembered or not. Based on **Theorem 2**, similar rules can be produced for a $\{M/G/1\} \circ \{G/G/k\}$ system applying either of the idling time schemes.

Using the mean value equations in **Theorem 2**, the cost function (16) can be given as

$$\begin{aligned} f = {} & \mathrm{E}[T_{M/G/1}]+\frac{\lambda\gamma}{\lambda\gamma\mathrm{E}[I_T]+\lambda+k\gamma}\left(\frac{k}{\lambda}(\frac{k-1}{2\lambda}+\frac{1}{\gamma})+\frac{\mathrm{E}[T^2_{\text{setup}}]}{2}\right)+ \\ & \frac{\beta}{\lambda}\left(\mathrm{E}[P_{M/M/1}]+\frac{(1-\rho)}{\lambda\gamma\mathrm{E}[I_T]+\lambda+k\gamma}(\lambda P_{\text{setup}}-(\lambda+\gamma k)P_{\text{idle}})\right)+ \\ & \beta^{'}\left((1-\rho)\frac{\lambda\gamma}{\lambda\gamma\mathrm{E}[I_T]+\lambda+k\gamma}\right). \end{aligned}$$

To gain a clearer intuition, this can be rewritten as

$$\begin{aligned} f = \mathrm{E}[T_{M/G/1}]+\frac{\beta}{\lambda}\mathrm{E}[P_{M/M/1}]+\frac{1}{\lambda\gamma\mathrm{E}[I_T]+\lambda+k\gamma}\Bigg(k\gamma(\frac{k-1}{2\lambda}+\frac{1}{\gamma})+\lambda\gamma\frac{\mathrm{E}[T^2_{\text{setup}}]}{2}+ \\ \beta(1-\rho)P_{\text{setup}}+\beta^{'}\gamma\lambda(1-\rho)-\beta(1-\rho)(1+\frac{k\gamma}{\lambda})P_{\text{idle}}\Bigg). \end{aligned}$$

The expression for the total idling time, $\mathrm{E}[I_T]$, appears only on the denominator of the cost function. Thus, if the last term is positive, that is if

$$k\gamma(\frac{k-1}{2\lambda}+\frac{1}{\gamma})+\lambda\gamma\frac{\mathrm{E}[T^2_{\text{setup}}]}{2}+\beta(1-\rho)P_{\text{setup}}+\beta^{'}\gamma\lambda(1-\rho)>\beta(1-\rho)(1+\frac{k\gamma}{\lambda})P_{\text{idle}},$$

then letting $\mathrm{E}[I_T] \to \infty$ (BUSY/IDLE policy) minimizes the total system cost. If this inequality does not hold,that is if

$$k\gamma(\frac{k-1}{2\lambda}+\frac{1}{\gamma})+\lambda\gamma\frac{\mathrm{E}[T^2_{\mathrm{setup}}]}{2}+\beta(1-\rho)P_{\mathrm{setup}}+\beta^{'}\gamma\lambda(1-\rho)<\beta(1-\rho)(1+\frac{k\gamma}{\lambda})P_{\mathrm{idle}},$$

then setting $\mathrm{E}[I_T]=0$ (BUSY/OFF policy) gives the minimum system cost.

5 Task Assignment Policies

In the previous section, we have determined the optimal control policies for a single server system under the given conditions. However, in a multiserver system, the task assignment policy will affect the delay and power consumption metrics, in addition to the control policy. A task assignment policy routes incoming jobs to the available servers using a predefined rule.

In the single server case, the optimal control policy is shown to be in the set {BUSY/IDLE, BUSY/OFF}. However, for a multiserver system this might not hold depending on which task assignment policy is applied. For this reason, we will examine three different task assignment policies and try to develop rules for determining the optimal control policy in each case. The task assignment policies will also be compared with each other.

5.1 Random Routing (RND)

As its name indicates, under this task assignment policy arriving requests are randomly assigned to the servers. Assuming n servers, an arriving job will be routed to the i^{th} server with probability p_i. Obviously,

$$\sum_{i=1}^{n} p_i = 1$$

should hold with $0 \leq p_i \leq 1$. The task assignment policy does not need to know the state of the servers for making routing decisions.

Due to the random nature of this policy, each queue can be regarded as a single server system with the arrival rate determined using the Poisson splitting property. That is, the aggregate arrival rate to the system, λ, can be split in to n Poisson arrival processes with values $p_i\lambda$.

The associated cost function given in (16) can be rewritten as

$$f = \sum_{i=1}^{n} f_i$$

where f_i is the cost of each individual queue. Similar to the single server analysis in the previous section, the optimal control policy will still be either of the BUSY/OFF or BUSY/IDLE policies. In either case, one can still optimize with respect to the routing probabilities. In the following section, this kind of optimization will be numerically illustrated for a two server system.

5.2 Join Shortest Queue (JSQ)

Under this policy, an incoming job is routed to the server that has the shortest queue, in other words, to the queue with the fewest jobs. For this to happen, the task scheduler needs to know the state of each server at the arrival time of each job. Thus, unlike the RND task assignment policy, JSQ dynamically adapts to changes of state in the whole system [21].

Earlier studies have discussed the optimality of the JSQ policy with the FIFO service discipline and service time with non-decreasing hazard rate [43, 40, 22]. However, the system cost in these discussions is restricted to that of delay experienced by a job.

5.3 Most Recently Busy Routing (MRB)

The MRB task assignment policy, like JSQ, needs to know the state of all the servers for its routing decision. Upon the arrival of a job, the state of all servers is checked and if only one server is found to be idle, the job is routed to that server. If two or more servers are found to be idle, then the job is assigned to the most recently busy server. In other words, to the server with the largest remaining idling time, allowing the other server(s) to run down their idling time and be turned off.

However, there might not be an idle server on the arrival of a job. In this case, we will look into two varieties of the MRB task assignment policy:

MRB-RND: This policy applies the MRB routing if one or more servers are in the *idle* state at the arrival time of a job. If there is no idle server, the arriving job will be routed randomly.

MRB-JSQ: In a similar way to the MRB-RND policy, MRB routing is applied first. If there is no idle server, an arriving job is assigned to the queue with the fewest jobs.

Leaving an idle server on (BUSY/IDLE) or turning it off immediately (BUSY/OFF) are shown to be optimal in a single server, as well as in a parallel server system with the random routing task assignment policy. However, this might not necessarily be true for MRB routing. In fact, the DELAYEDOFF control policy is shown to be near-optimal for a central queue multiserver system with MRB routing used as a task assignment policy [16].

In Section 6, the RND, JSQ and MRB task assignment policies will be studied for an energy-aware system with parallel servers. Bound with the RND and JSQ task assignment policies, the BUSY/IDLE(t) control policy will also be studied. The BUSY/IDLE(t) policy, as defined in Section 3.2.2, allows a maximum of t servers to stay in the idle state. Once this threshold is reached all non-busy servers are turned off immediately.

6 Numerical results

In the preceding sections, a queueing-theoretic analysis of energy-aware server(s) is performed. To gain even more intuition to the trade-off involved, numerical illustration of some specific examples is provided in this section. The system cost, in the subsequent numerical analysis, is modeled by the weighted sum objective function

$$f = \mathrm{E}[T] + \beta \mathrm{E}[P].$$

6.1 Single server system

In this numerical illustration, the focus will be on the $\{M/M/1\} \circ \{M/D/k\}$ and $\{M/M/1\} \circ \{D/D/k\}$ systems. However, the distribution of the idling time has little or no significance since the optimal choice is either $\alpha = 0$ or $\alpha \to \infty$. In Section 6.1.1 we will compare the BUSY/IDLE and BUSY/OFF policies while Section 6.1.2 provides Pareto optimization of mean delay and power with respect to the threshold k.

6.1.1 BUSY/IDLE vs BUSY/OFF

In the forthcoming discussion, the $\{M/M/1\} \circ \{M/D/k\}$ system will be studied with the following parameters: $\mathrm{E}[S] = 1s, P_{\text{busy}} = P_{\text{setup}} = 240W, P_{\text{idle}} = 150W$ and $\beta = 0.1$. While the mean service time and power consumption values are taken from [16], β can be chosen arbitrarily depending on how much we want to emphasize the energy cost relative to that of delay. Based on these values, Figure 9 depicts the system cost incurred by the BUSY/IDLE and BUSY/OFF policies for different load values as a function of the setup time.

In the lightly loaded systems of Figure 9a and 9b, the BUSY/OFF policy emerges as the optimal policy for setup time values up to around $20\mathrm{E}[S]$ and $10\mathrm{E}[S]$ respectively. Within this range, one can still minimize cost by optimizing with respect to the threshold, k. It can be seen that in both systems high threshold values ($k = 7, 10$) are not favored. One reason for this can be, coupled with the light load (larger inter-arrival times), higher threshold induces a considerable amount of waiting time for those jobs that arrive while the server is turned off. It can be observed from Figure 9c and 9d that this effect diminishes as the load increases.

The system with medium load, shown in Figure 9c, has somehow similar characteristics as those discussed above. However, for the system in Figure 9d the BUSY/OFF policy is no longer optimal even for low setup time values due to the high load.

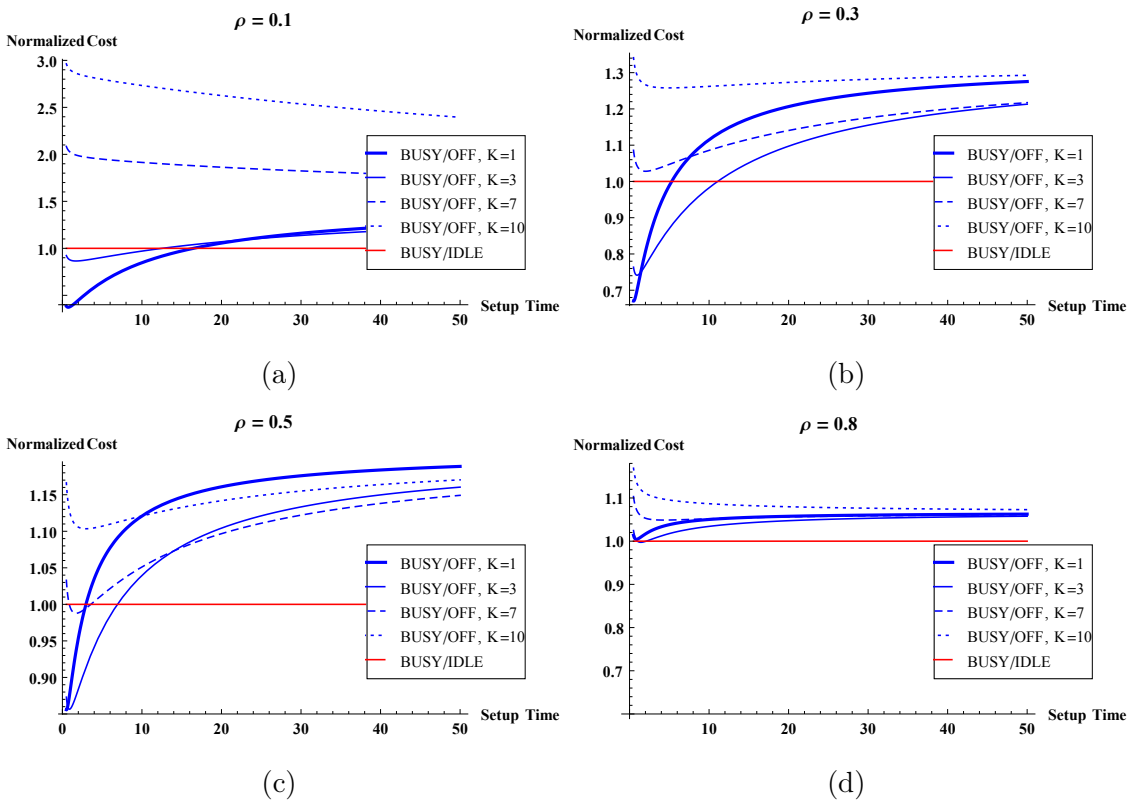


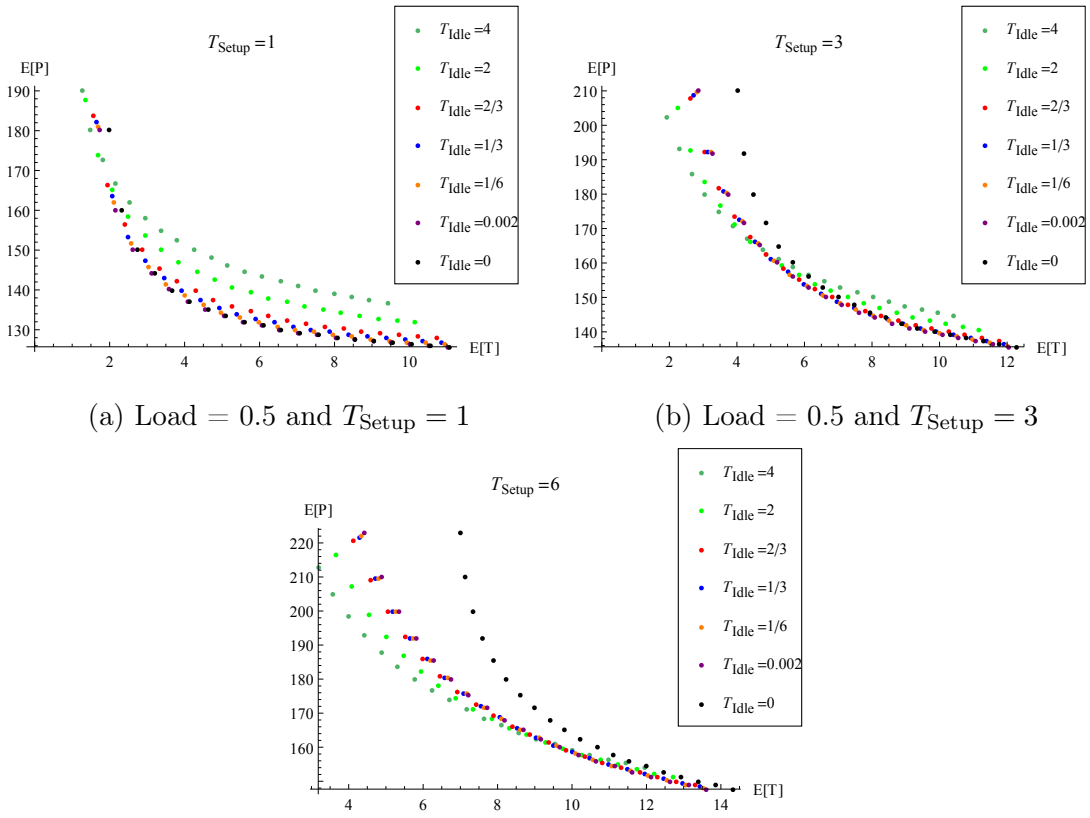
Figure 9: System cost of different control policies normalized with respect to the idling policy. $P_{\text{busy}} = P_{\text{setup}} = 240W$, $P_{\text{idle}} = 150W$ and $\beta = 0.1$

6.1.2 Pareto optimization

In an $\{M/M/1\} \circ \{D/D/k\}$ system with both the setup and idling times fixed, one can optimize with respect to the threshold k. The result of such optimization for the system given in Figure 9c is given in Figure 10. In each of the three figures, a curve of the same color is obtained by varying the threshold from 1 to 20. As can be expected, points closer to the Y-axis (E$[P]$ axis) correspond to lower values of k while those closer to the X-axis (E$[T]$ axis) represent higher k values. Once the setup time and idling time values are fixed, all the resulting set of points happen to be Pareto optimal. That is, by varying the value of the threshold we can alleviate the cost caused by one component only at the expense of increasing the cost induced by the other.

6.2 Two servers with random task assignment

In [28] the random task assignment policy is studied by considering a system with two parallel $\{M/M/1\} \circ \{M/M/k\}$ queues. In this section, we will reconstruct the results and provide detailed analysis on the optimal choices for the parameters of the energy-aware system.



(a) Load = 0.5 and $T_{\text{Setup}} = 1$

(b) Load = 0.5 and $T_{\text{Setup}} = 3$

(c) Load = 0.5 and $T_{\text{Setup}} = 6$

Figure 10: Pareto optimization of the single server system.

When a job arrives to the system, it will be routed to the first server's queue with probability p and routed to the second one with probability $1 - p$. The servers are assumed to be homogeneous with the same service rate, setup rate and energy costs. We already know that for an optimal performance we should set α_i =0 or $\alpha_i = \infty$. Thus, for the three different cases ($\alpha_1 = \alpha_2 = 0$, $\alpha_1 = \alpha_2 = \infty$ or $\alpha_1 = \infty, \alpha_2 = 0$) the optimization is done with respect to k_1, k_2 and p. The other case, where $\alpha_1 = 0, \alpha_2 = \infty$, is ignored due to symmetry. The objective is to minimize the system cost modeled by the weighted sum of energy and delay. Several examples of the system cost, for the three different conditions given above, are presented in Figure 11.

In Figure 11a, a load that can be handled by one of the servers is considered. The system load, if handled by only one of the servers will be $\rho = 0.95$. For $p = 0$, all the jobs are served by the second server. Therefore, keeping the first server on will only increase the cost, which explains the higher cost of setting $\alpha_1 = \alpha_2 = 0$ for $p = 0$. Comparing the result for the other two conditions, in which we can turn the first server off, one can see that turning the second server off will only increase our cost. This is due to the fact that the server will have to be started shortly after it is turned off due to the high load. The low setup rate will make the penalty for turning the server off even worse. The same logic will make the cost of setting $\alpha_1 = \infty$ and

$\alpha_2 = 0$ a lot worse when $p = 1$. For $p \leq 0.453$, letting the first server to turn off and keeping the second one always on pays off. The minimum cost incurred for this setup being 34.627 when $p = 0.1$. It is also shown that if we are going to route jobs to either of the servers with equal probabilities, then leaving both servers on will produce the minimum cost.

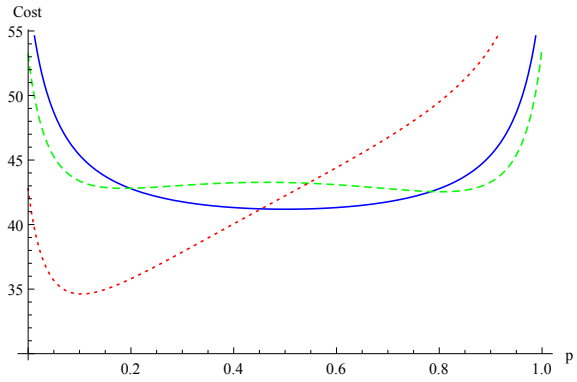
In Figure 11b the cost function of the three different conditions is illustrated for a lighter load. The load is considerably smaller so that even with only one of the servers on, the load on the system will be $\rho = 0.5$. Obviously, directing all the jobs to the second server and turning the first server off will be more efficient. Another non-trivial result is that letting the second server to switch off while it is idle will only increase the cost. This is partly due to the fact that the load is not so small for one server to handle all the load and also partly due to the low setup rate and choice of β.

Figures 11c and 11d show the cost function of the system under a heavy load such that both servers need to be in working state for the system to be stable[4]. The result is intuitive since leaving both servers on will give the minimum cost for a low setup rate while letting both servers to turn off will be optimal when the setup rate is higher.

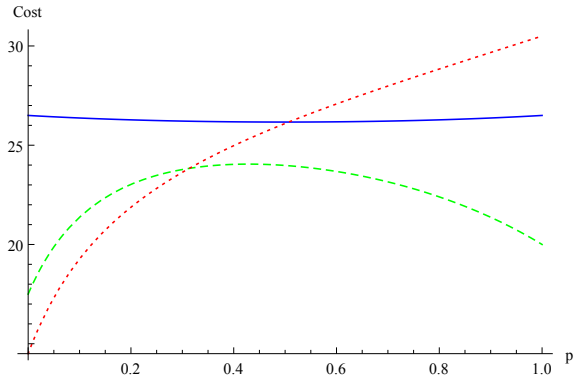
In Figure 11e we are neglecting the energy cost, $\beta = 0$. Leaving both servers on will clearly give the minimum cost. For $p = 0$, all the jobs are directed to the second server, therefore setting $\alpha_2 = 0$ will be optimal regardless of the state of the first server. This is because we are neglecting the energy cost. One can see that letting the second server to turn off will have a higher cost in this case.

Figure 11f illustrates the cost of the system when even more attention is given to the energy cost, $\beta = 100$. Obviously, allowing both servers to turn off when there is no job to serve will minimize our cost in this setup.

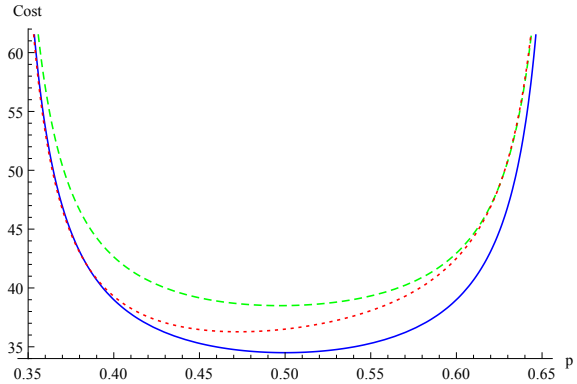
[4]In the corresponding figure for Figure 2(d) in [28] there is a misprint of the value of λ, which otherwise would give a lighter load.



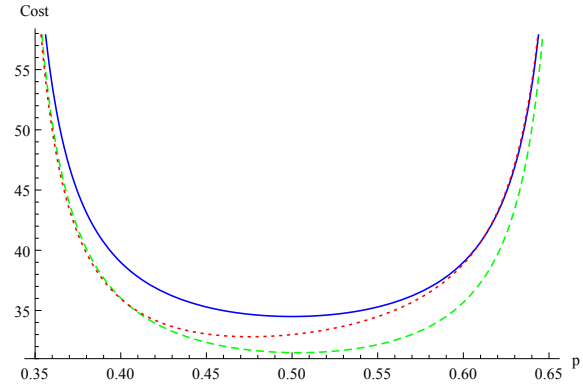
(a) $\mu = 2, \lambda = 1.9, \gamma = 0.2, \beta = 22, k_1 = 5, k_2 = 7$



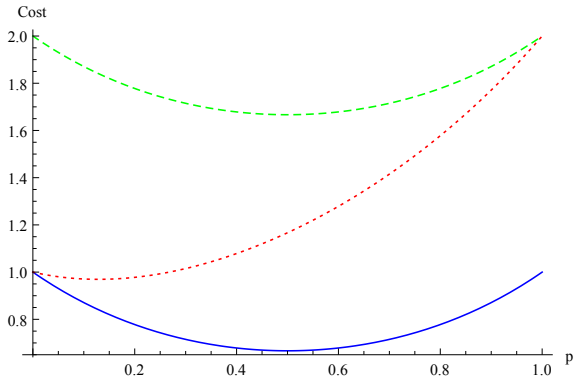
(b) $\mu = 2, \lambda = 1, \gamma = 0.2, \beta = 15, k_1 = 1, k_2 = 4$



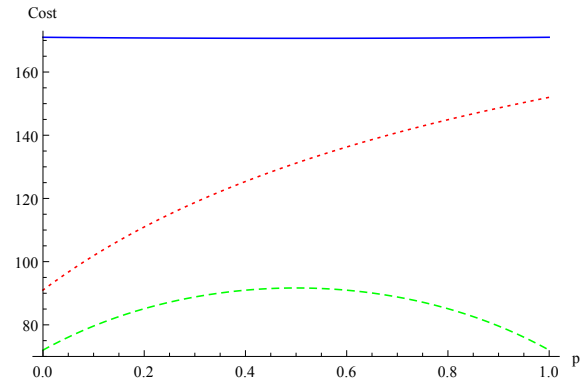
(c) $\mu = 2, \lambda = 3, \gamma = 0.5, \beta = 15, k_1 = 3, k_2 = 2$



(d) $\mu = 2, \lambda = 3, \gamma = 3, \beta = 15, k_1 = 1, k_2 = 2$



(e) $\mu = 2, \lambda = 1, \gamma = 1, \beta = 0, k_1 = 1, k_2 = 1$



(f) $\mu = 2, \lambda = 1, \gamma = 1, \beta = 100, k_1 = 1, k_2 = 1$

Figure 11: Random Routing: Optimization of a two server system with respect to p.

6.2.1 Optimization with respect to k and p

In the preceding discussion, we have seen how the cost of the two-server-system is affected by the choice of control policy, mainly by varying the routing probability p. Now we will focus on the k values and try to find the optimal combination (k_1, k_2) that minimizes the cost of the system in Figure 11a with both servers applying the BUSY/OFF policy. The value set $\{1, 10\}$ is assumed and for each value of k_1, k_2 is iterated from 1 to 10 leading to 100 possible combinations. For each value of k_1, the k_2 value that produces the minimum cost within that iteration is illustrated by the blue points in Figure 12 along with the associated p value. The red points provide the (k_1, k_2) combinations that result in minimum values for the entire iterations. Comparing this result to that of 11a, one can see that the cost improvement achieved by optimization with respect to the threshold values is marginal.

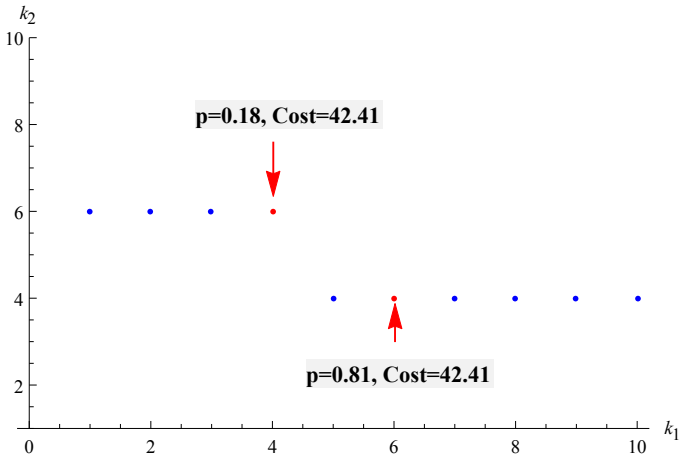


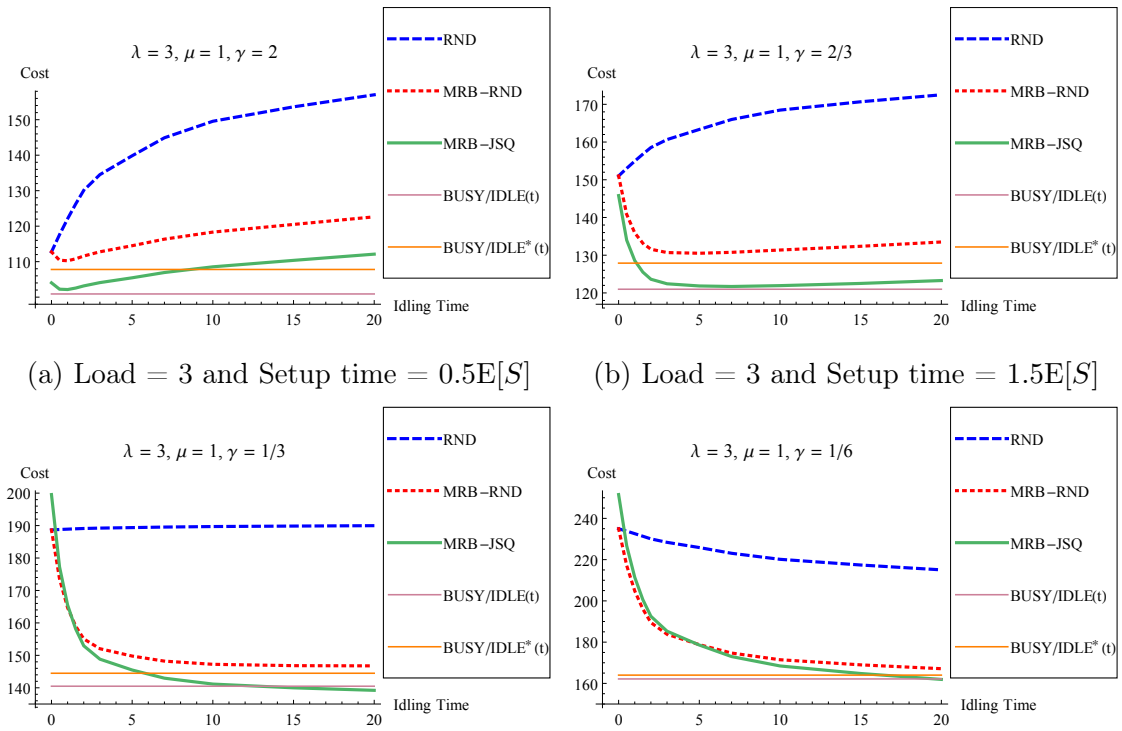
Figure 12: Random routing: Optimization of a two server system with respect to k and p, taking system parameters in Figure 11a.

6.3 Multiserver system

In a practical configuration, a server farm contains multiple servers to which a task assignment policy routes incoming jobs by applying a specific set of rules. To come up with energy-aware control and task assignment policies for this kind of setup, a server farm composed of 10 servers will be studied in this section. The RND, MRB-RND and MRB-JSQ task assignment policies will be examined along with the BUSY/IDLE, BUSY/OFF, DELAYEDOFF and BUSY/IDLE(t) control policies. For instance, a server farm that employs the RND task assignment policy together with the BUSY/IDLE control policy will route incoming jobs randomly while keeping both busy and idle servers turned on all the time.

When the task assignment policy under consideration is different from RND, closed form expressions can not be derived for the mean delay and mean energy metrics using simple stochastic models. Thus, the simulator provided in Appendix

A is used to produce the results in this illustration.



(a) Load = 3 and Setup time = 0.5E[S] (b) Load = 3 and Setup time = 1.5E[S]

(c) Load = 3 and Setup time = 3E[S] (d) Load = 3 and Setup time = 6E[S]

Figure 13: Total cost of the system under the RND, MRB-RND, MRB-JSQ and BUSY/IDLE(t) policies with load = 3. The $BUSY/IDLE(t)$ control policy allows t servers to be idle at a time and applies the JSQ task assignment policy while the $BUSY/IDLE^*(t)$ policy applies the RND task assignment policy.

Power consumption values of $P_{\text{busy}} = P_{\text{setup}} = 240W$ and $P_{\text{idle}} = 150W$ with exponentially distributed service times with mean E[S] = 1 s are used. These values are based on practical measurements conducted in [16]. Both setup and idling time are assumed to be deterministic with idling time reset every time the server goes out of the *idle* state while in all cases the turn-on threshold is $k = 1$. Additionally, jobs arrive to the system according to a Poisson process, with rate λ, while each server has its own queue on which the FIFO discipline is applied. The total system cost is modeled by the weighted sum of mean delay and mean power consumption with $\beta = 0.1$.

For a systematic study of each task assignment and control policy combination, the total cost of the system is computed by varying the load, setup and idling times. The result, as shown in Figures 13 - 15, is plotted as a function of the idling time. In each of these figures, there are two lines with labels BUSY/IDLE(t) and BUSY/IDLE*(t). The difference between these lies only on the task assignment policy used to produce each curve. That is, the BUSY/IDLE(t) allows t servers

to be idle at the same time and applies the JSQ task assignment policy while the BUSY/IDLE*(t) policy uses the RND policy with the same threshold t for idle servers. Clearly, the value of the threshold t impacts the total cost of the system. Hence, a separate simulation was run to determine the optimal value of t that minimizes the system cost under the BUSY/IDLE(t) policy for each combination of load and setup time.

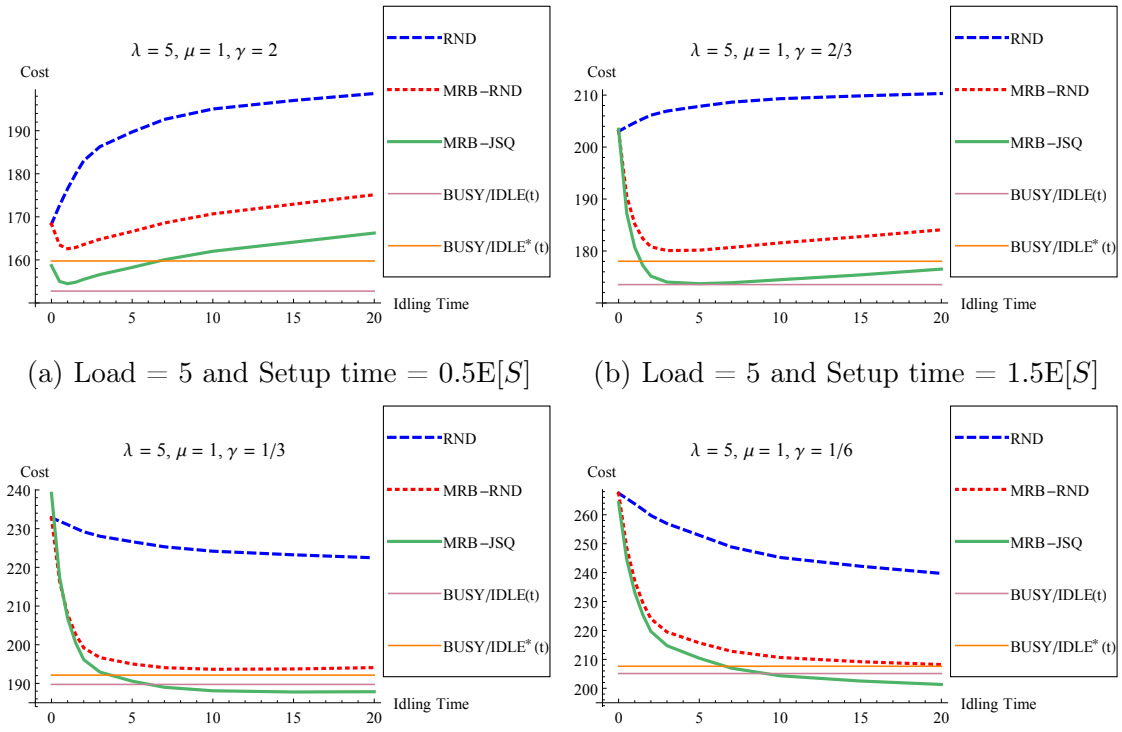
The BUSY/IDLE(t) policy has t BUSY/IDLE and $10 - t$ BUSY/OFF servers, and hence it has only two possible values of idling time. In the figures, it is plotted as a function of idling time only for the sake of comparison.

Simulation results of the RND, MRB-RND and MRB-JSQ task assignment policies and that of the BUSY/IDLE(t) control are labeled explicitly in all of these figures. However, those of BUSY/OFF and DELAYEDOFF policies can also be observed by considering specific coordinates in the X-axis (the Idling Time axis). That is, when Idling Time = 0, we have the BUSY/OFF policy whereas non-zero values of idling time give the DELAYEDOFF policy.

The total system cost for load values of $3, 5$ and 8 are depicted in Figures 13, 14 and 15 respectively. Within each of these figures, setup time is varied from $0.5\text{E}[S]$ to $6\text{E}[S]$. In all cases, the cost of the random task assignment policy is monotonous, indicating that the optimal control policy is either the BUSY/OFF or BUSY/IDLE. This is in line with what was discussed in Section 5.1.

However, in the case of the MRB-JSQ and MRB-RND task assignment policies, the choice of the optimal control policy is not straightforward. For a lighter load, $\{3, 5\}$, and low setup time compared to service time, the cost of using the BUSY/OFF and DELAYEDOFF control policies is slightly lower than that of the BUSY/IDLE, with up to 9% improvement. However, when the setup time becomes higher compared to service time ($1.5\text{E}[S], 3\text{E}[S], 6\text{E}[S]$), the BUSY/OFF policy becomes inefficient for all the load values under consideration. By applying the DELAYEDOFF or BUSY/IDLE control policies instead of BUSY/OFF, the total cost of the system can be reduced by $20 - 30\%$. Moreover, beyond a certain value of idling time, the system cost becomes flat which implies that further cost reduction can not be achieved by letting longer idling times.

Finally, for all load and setup time values, the MRB-JSQ task assignment policy combined with the DELAYEDOFF control or the JSQ task assignment policy combined with the BUSY/IDLE(t) control policy emerge as the two most efficient policies among the others. Although there is no clear winner between these two, it can be seen that the JSQ - BUSY/IDLE(t) combination has a marginal gain in some of the configurations.



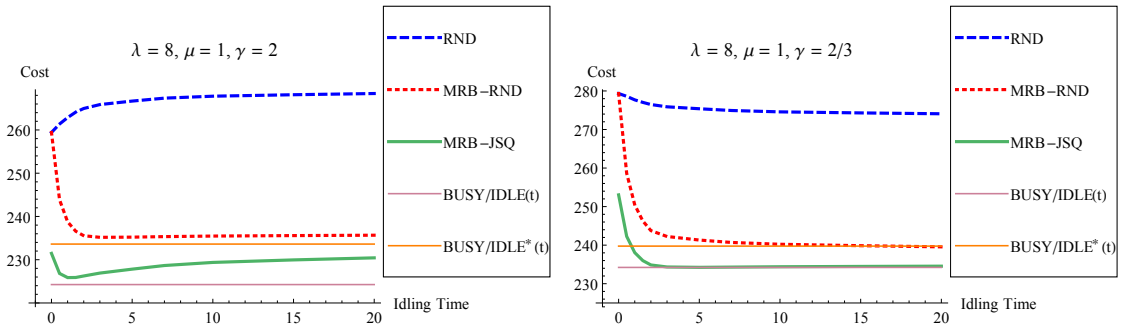
(a) Load = 5 and Setup time = 0.5E[S]

(b) Load = 5 and Setup time = 1.5E[S]

(c) Load = 5 and Setup time = 3E[S]

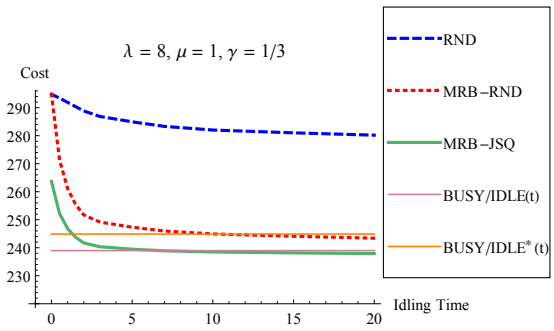
(d) Load = 5 and Setup time = 6E[S]

Figure 14: Total cost of the system under the RND, MRB-RND, MRB-JSQ and BUSY/IDLE(t) policies with load = 5.

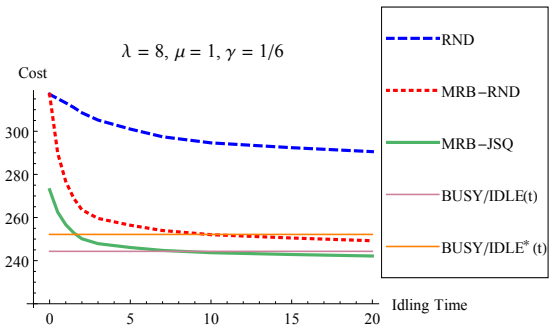


(a) Load = 8 and Setup time = 0.5E[S]

(b) Load = 8 and Setup time = 1.5E[S]



(c) Load = 8 and Setup time = 3E[S]



(d) Load = 8 and Setup time = 6E[S]

Figure 15: Total cost of the system under the RND, MRB-RND, MRB-JSQ and BUSY/IDLE(t) policies with load = 8.

7 Conclusion

Energy consumption of server farms has caused a major concern which required the attention of the ICT community. To contribute to this cause, a queueing theoretic study of server control and task assignment policies was conducted in this thesis.

An attempt to reduce energy consumption often comes with a decline in performance. Thus, any applicable solution should strike the right balance between energy and performance. To ensure this, the weighted sum of delay and energy metrics was used as a representation of the system cost. In our study of a single server system, closed form expressions for the system cost were developed and applied to determine the optimal server control policies.In the multiserver setting, numerical analysis was done to come up with energy-aware task assignment and server control policies.

Turning an idle server off induces a setup cost which has energy and delay components. This introduces the *off, setup, busy* and *idle* energy states to the system. Analysis of the general $\{M/G/1\} \circ \{G/G/k\}$ system, which has a turn-on threshold of k jobs, showed that the optimal control policy for a single server system lies in the set $\{\text{BUSY/IDLE}, \text{BUSY/OFF}\}$. That is, we should either turn the server off immediately after the last job leaves the system or keep it idle until the next job arrives. This result holds irrespective of whether idling time is remembered or reset every time the server comes out of the idle state.

In a multiserver system, the applied task assignment policy impacts the energy efficiency of the system in addition to the control policy. A numerical analysis of the RND, MRB-RND and MRB-JSQ task assignment policies was performed along with the BUSY/IDLE, BUSY/IDLE(t), BUSY/OFF and DELAYEDOFF control policies. In addition to low, medium and high load values, setup time was varied relative to service time requirements and the resulting system was studied. The combination of the MRB-JSQ task assignment policy and the DELAYEDOFF control policy or that of JSQ and BUSY/IDLE(t) are found to reduce up to 30% of the system cost.

7.1 Future work

As already mentioned, we have studied energy efficiency of server farms by applying the weighted sum of mean energy and mean delay as a system cost. However, the value of the weighting factor in this model is purely subjective and can be varied depending on how much one wants to emphasize energy cost. Therefore, other cost models also need to be considered for a complete study of energy efficiency in server farms.

Moreover, the analysis throughout this thesis was based on the assumption that the service discipline is FIFO. Hence, the results may not hold for other service disciplines. Finally, our study of multiserver systems was restricted to the numerical

simulation of selected policies. Hence, the optimality problem remains to be solved for a multiserver setting.

References

[1] U.S. Environmental Protection Agency. Report to congress on server and data center energy efficiency, 2007. [http://www.energystar.gov/ia/partners/prod_development/downloads/EPA_Datacenter_Report_Congress_Final1.pdf Online; accessed 20-January-2014].

[2] Susanne Albers and Hiroshi Fujiwara. Energy-efficient algorithms for flow time minimization. *ACM Trans. Algorithms*, 3(4), November 2007.

[3] Lachlan L.H. Andrew, Minghong Lin, and Adam Wierman. Optimality, fairness, and robustness in speed scaling designs. *SIGMETRICS Perform. Eval. Rev.*, 38(1):37–48, June 2010.

[4] Mauricio Arregoces and Maurizio Portolani. *Data Center Fundamentals*. Cisco Press, 2003.

[5] A. Azevedo, I. Issenin, R. Cornea, R. Gupta, N. Dutt, A. Veidenbaum, and A. Nicolau. Profile-based dynamic voltage scheduling using program checkpoints. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '02, pages 168–, Washington, DC, USA, 2002. IEEE Computer Society.

[6] Nikhil Bansal, Ho-Leung Chan, and Kirk Pruhs. Speed scaling with an arbitrary power function. In *Proceedings of the Twentieth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '09, pages 693–701, Philadelphia, PA, USA, 2009. Society for Industrial and Applied Mathematics.

[7] Nikhil Bansal, Tracy Kimbrel, and Kirk Pruhs. Speed scaling to manage energy and temperature. *J. ACM*, 54(1):3:1–3:39, March 2007.

[8] Nikhil Bansal, Kirk Pruhs, and Cliff Stein. Speed scaling for weighted flow time. In *Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '07, pages 805–813, Philadelphia, PA, USA, 2007. Society for Industrial and Applied Mathematics.

[9] Luiz André Barroso and Urs Hölzle. The case for energy-proportional computing. *Computer*, 40(12):33–37, December 2007.

[10] Stephen Boyd and Lieven Vandenberghe. *Convex Optimization*. Cambridge University Press, New York, NY, USA, 2004.

[11] David P. Bunde. Power-aware scheduling for makespan and flow. In *Proceedings of the Eighteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '06, pages 190–196, New York, NY, USA, 2006. ACM.

[12] Green Mountain Data Center. Green mountain: Cooling system. [http://greenmountain.no/efficiency/cooling-system Online; accessed 20-January-2014].

[13] Cisco. Cisco learning. [http://www.cisco.com/c/en/us/td/docs/solutions/Enterprise/Data_Center/DC_Infra2_5/DCInfra_1.html Online; accessed 20-February-2014].

[14] Microsoft Corporation. Server clusters : Architecture overview for windows server 2003, 2003.

[15] Anshul Gandhi, Sherwin Doroudi, Mor Harchol-Balter, and Alan Scheller-Wolf. Exact analysis of the M/M/K/Setup class of markov chains via recursive renewal reward. *SIGMETRICS Perform. Eval. Rev.*, 41(1):153–166, June 2013.

[16] Anshul Gandhi, Varun Gupta, Mor Harchol-Balter, and Michael A. Kozuch. Optimality analysis of energy-performance trade-off for server farm management. *Perform. Eval.*, 67(11):1155–1171, November 2010.

[17] Anshul Gandhi and Mor Harchol-Balter. M/g/k with staggered setup. *Operations Research Letters*, 41(4):317 – 320, 2013.

[18] Anshul Gandhi, Mor Harchol-Balter, and Ivo Adan. Decomposition results for an M/M/K with staggered setup. *SIGMETRICS Perform. Eval. Rev.*, 38(2):48–50, October 2010.

[19] Anshul Gandhi, Mor Harchol-Balter, and Ivo Adan. Server farms with setup costs. *Perform. Eval.*, 67(11):1123–1138, November 2010.

[20] Jennifer M. George and J. Michael Harrison. Dynamic control of a queue with adjustable service rate. *Operations Research*, 49(5):720–731, 2001.

[21] Mor Harchol-Balter. *Performance Modeling and Design of Computer Systems: Queueing Theory in Action*. Cambridge University Press, 2013.

[22] A. Hordijk and G. Koole. On the optimality of the generalized shortest queue policy. *peis*, **4**:477–487, 1990.

[23] Stefanos Kaxiras and Margaret Martonosi. *Computer Architecture Techniques for Power-Efficiency*. Morgan and Claypool Publishers, 1st edition, 2008.

[24] Leonard Kleinrock. *Queueing Systems*, volume I: Theory. Wiley Interscience, 1975. (Published in Russian, 1979. Published in Japanese, 1979. Published in Hungarian, 1979. Published in Italian 1992.).

[25] Jonathan G. Koomey. Growth in data center electricity use 2005 to 2010. *Analytics Press*, 2011.

[26] H. Levy and L. Kleinrock. A queue with starter and a queue with vacations: Delay analysis by decomposition. *Operations Research*, 34(3):426–436, May-June 1986. (cover article).

[27] Xiahoa Lu, Samuli Aalto, and Pasi Lassila. Performance-energy trade-off in data centers: Impact of switching delay. In *Proceedings of 22nd ITC Specialist Seminar on Energy Efficient and Green Networking (SSEEGN 2013)*, pages 50–55, Christchurch, New Zealand, Nov. 2013.

[28] Vincent Maccio and Douglas Down. On optimal policies for energy-aware servers. MASCOTS, 2013.

[29] J. Jyotiprasad Medhi. *Stochastic models in queueing theory*. Academic Press, Amsterdam, Boston, 2003.

[30] Tian Naishuo and Zhe George Zhang. *Vacation Queueing Models : Theory and Applications*. Springer, New York, 2006.

[31] A. Penttinen, E. Hyytia, and S. Aalto. Energy-aware dispatching in parallel queues with on-off energy consumption. In *Performance Computing and Communications Conference (IPCCC), 2011 IEEE 30th International*, pages 1–8, 2011.

[32] Gregory F. Pfister. *In Search of Clusters (2Nd Ed.)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1998.

[33] Kirk Pruhs, Patchrawat Uthaisombut, and Gerhard Woeginger. Getting the best response for your erg. *ACM Trans. Algorithms*, 4(3):38:1–38:17, July 2008.

[34] S. M. Ross. *Applied Probability Models with Optimization Applications*. Holden-Day, San Francisco, 1970.

[35] Christian Schwartz, Rastin Pries, and Phuoc Tran-Gia. A queuing analysis of an energy-saving mechanism in data centers. In *International Conference on Information Networking (ICOIN)*, Bali, Indonesia, 2 2012.

[36] Howard E. Taylor and Samuel Karlin. *An Introduction to Stochastic Modeling*. Academic Press, 3rd edition, February 1998.

[37] European Union. The european code of conduct for data centres. http://iet.jrc.ec.europa.eu/energyefficiency/ict-codes-conduct/data-centres-energy-efficiency Online; accessed 20-January-2014.

[38] O. S. Unsal and I. Koren. System-level power-aware design techniques in real-time systems. *Proceedings of the IEEE*, 91(7):1055–1069, July 2003.

[39] Justin Vela. The guardian: Helsinki data centre to heat homes, July 2010. [http://www.theguardian.com/environment/2010/jul/20/helsinki-data-centre-heat-homes Online; accessed 20-January-2014].

[40] Richard R. Weber. On the optimal assignment of customers to parallel servers. *Journal for Applied Probability*, 15:406–413, 1978.

[41] P. Welch. On a generalized M/G/1 queueing process in which the first customer of each busy period receives exceptional service. *Operations Research*, pages 736–752, 1964.

[42] Adam Wierman, Lachlan L. H. Andrew, and Ao Tang. Power-aware speed scaling in processor sharing systems: Optimality and robustness. *Perform. Eval.*, 69(12):601–622, December 2012.

[43] W. Winston. Optimality of the shortest line discipline. *Journal for Applied Probability*, **14**:181–189, 1977.

[44] F. Yao, A. Demers, and S. Shenker. A scheduling model for reduced cpu energy. In *Proceedings of the 36th Annual Symposium on Foundations of Computer Science*, FOCS '95, pages 374–, Washington, DC, USA, 1995. IEEE Computer Society.

[45] Lin Yuan and Gang Qu. Analysis of energy reduction on dynamic voltage scaling-enabled systems. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 24(12):1827–1837, 2005.

A Simulation code

We have seen that single server systems with energy-aware control policies can be modeled and analyzed with relative ease. However, server farms, with multiple servers, are complicated to model mathematically. In addition to this, the need for a task assignment policy makes the problem more complicated. For this reason, a C^{++} based simulator is used to study the behavior of the control and task assignment policies introduced in this thesis.

A C^{++} class library called CNCL[5] is used to implement the RND, MRB-RND and MRB-JSQ task assignment policies combined with the BUSY/OFF, BUSY/IDLE and BUSY/IDLE(t) control policies. An example $M/M/1$ queue simulator is extended to implement the aforementioned policies. Each implementation has three main parts as shown in Figure 16.

The entire code for the random task assignment simulator is provided in the following section while only the generator classes of the MRB-RND and MRB-JSQ are given in the subsequent sections to avoid repetition.

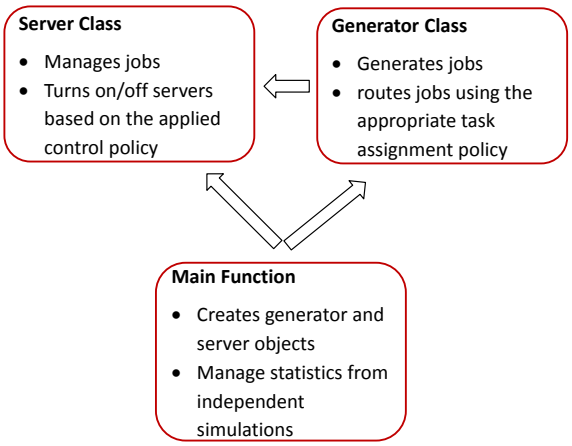


Figure 16: Logical structure of the simulator.

[5]CNCL (Communication Netowrks Class Library) is a collection of C^{++} classes developed in Aachen University of Technology for event driven simulations of Communication Networks.

A.1 Random task assignment

```
#include <string>
#include <iostream>
#include <strstream>

#include <CNCL/QueueFIFO.h>
#include <CNCL/EventScheduler.h>
#include <CNCL/MT.h>
#include <CNCL/NegExp.h>
#include <CNCL/Moments.h>
#include <CNCL/Uniform.h>
#include <CNCL/Job.h>
#include <fstream>


ofstream Rnd_Srv;
int Random_flag = 0;

#define Num_of_Servers 10
#define Beta 0.1    //Weighting Factor for energy
#define SETUP 3
#define P_on 240
#define P_idle 150
#define K 1 // Threshold number of jobs to turn the server ON

// Limit for jobs to be processed
enum { NJOBS=100000 };

// CNEvent types for energy-aware simulation
enum { EV_JOB, EV_TIMER_G, EV_TIMER_S, EV_TIMER_I, EV_TIMER_SET };

class Server : public CNEventHandler
{
private:
  unsigned long ID;      // Event ID
  CNJob *job;           // Served job
  //ofstream Statistics;
  //CNQueueFIFO queue;    // CNQueue
  CNRandom &rnd_b;      // Distribution of service time b
  //CNRandom &rnd_s;   // Distribution of setup time s
  float DET_IDLE;

  int flag;    // used between the SETUP and SERVING states
```

```
  enum { ST_IDLING, ST_SERVING, ST_SETUP, ST_OFF };

public:
  CNQueueFIFO queue;
  virtual void event_handler(const CNEvent *ev);
  void print_results();
  void eval_job(CNJob *job);

  CNMoments t_w, t_b;   // Evaluation tau_w, tau_b
   // mean time spent in each state: off, setup, serving and idle
  CNMoments t_off, t_setup, t_serving, t_idling, t_workcycle;
   // Used to capture the time at which a new workcycle is started
  CNSimTime WorkCycle_Started;
  //Used to capture the time at which a new idling perios is started
  CNSimTime Idling_Started;
  CNSimTime Setup_Started;
  CNSimTime Serving_Started;
  // Just to check completeness/correctness of work-cycle
  CNSimTime Off_Started;
  CNSimTime Setup_Temp;
  // captures intermediate values in the busy period calculation
  CNSimTime Serving_Temp;
  // captures intermediate values in the idle period calculation
  CNSimTime Idling_Temp;
  //CNSimTime Work_Cycle;
  // state of server used in Generator(OFF=0,SETUP=1,SERVING=2,IDLING=3)
  int STATUS;
  int WC_Counter;
  Server(CNRandom &rnd, float IDLE)
    : job(NIL), rnd_b(rnd), DET_IDLE(IDLE), flag(0), t_w("tau_w"),
    t_b("tau_b"), t_off("Turned Off"), t_setup("Setup_Time"),
    t_serving("Serving_Time"), t_idling("Idle_Time"),
    t_workcycle("Work Cycle"), WorkCycle_Started(0),
    Idling_Started(0), Setup_Started(0), Serving_Started(0),
    Off_Started(0), Setup_Temp(0), Serving_Temp(0), Idling_Temp(0),
    STATUS(0), WC_Counter(1)
    {
      state(ST_OFF);
    }
};

class Generator : public CNEventHandler
{
private:
  float prob;
```

```
  CNUniform rnd_p; //routing probability
  CNRandom &rnd_a;  // Distribution of arrival time a

  // Connected queues/servers
  Server (&server_farm)[Num_of_Servers];

  int temp;
  int Index;
  long n;

public:
  virtual void event_handler(const CNEvent *ev);
  Generator(float p, CNUniform rndP, CNRandom &rnd,
  Server (*serv)[Num_of_Servers]) : prob(p), rnd_p(rndP), rnd_a(rnd),
  server_farm(*serv), temp(0), Index(0),
    n(0){}
};


void Generator::event_handler(const CNEvent *)
{
  if(n == NJOBS)
  // Stop simulation
  return;

  // Incoming event -> generate new Job
  temp = int( (Num_of_Servers) * rnd_p.draw() );

  send_now(new CNEvent(EV_JOB, &server_farm[temp], new CNJob));
  // CNRandom delay
  send_delay(new CNEvent(EV_TIMER_G), rnd_a());
  n++;
}


void Server::event_handler(const CNEvent *ev)
{

  switch(state())
  {

   case ST_OFF:
      switch(ev->type())
      {
             case EV_JOB:
```

```
                CNJob *job;
                job = (CNJob *)ev->object();
                job->in = now();
                queue.put(job);
                if(queue.length() == K)
                {
                    t_off.put(now() - Off_Started);
                    Off_Started = 0;
                    send_delay(new CNEvent(EV_TIMER_SET), SETUP);
                    Setup_Started = now();
                    STATUS=1;
                    state(ST_SETUP);
                }
                else
                    state(ST_OFF);
          break;

          case EV_TIMER_I:
             error("illegal event 'IDLING' in state ST_OFF");
          break;

          case EV_TIMER_SET:
             error("illegal event 'SETUP' in state ST_OFF");
          break;

          default:
             error("illegal event in state ST_OFF");
          break;

          }
       break;

   case ST_SETUP:
          switch(ev->type())
          {
              case EV_JOB:
              CNJob *job;
              job = (CNJob *)ev->object();
              job->in = now();
              queue.put(job);
              break;

              case EV_TIMER_SET:   //Setup completed
                  flag = 1;
                  Setup_Temp = now() - Setup_Started;
```

```
                Setup_Started = 0;
                send_now(new CNEvent(EV_TIMER_S));

                Serving_Started = now();
                STATUS = 2;
                state(ST_SERVING);

            break;

            case EV_TIMER_I:
                error("illegal event 'IDLING' in state ST_SETUP");
            break;

            case EV_TIMER_S:
                error("illegal event 'IDLING' in state ST_SETUP");
            break;

            default:
                 error("illegal event in state ST_SETUP");
            break;
        }
        break;

    case ST_SERVING:
            switch(ev->type())
            {
                 case EV_JOB:
                 // Incoming job, put into queue

                     CNJob *job;
                     job = (CNJob *)ev->object();
                     job->in = now();
                     queue.put(job);

                 break;

                 case EV_TIMER_S:

                    // Timer event, service time run down
                    switch(flag)
                    {
                        case 0:
                            job->out = now();
                            // Evaluate job if transient is over
                            if(now() > 1000)
```

```
            {
               eval_job(job);
            }
            delete job;
            job = NIL;
        break;

        case 1:
            flag = 0;
        break;

        default:
            error("mm1: ", "incorrect value for flag");
        break;
    }
    // Get new job from queue
    if(!queue.empty())
    {
        job = (CNJob *)queue.get();
        job->start = now();
        // CNRandom service time
        send_delay(new CNEvent(EV_TIMER_S), rnd_b());
        STATUS = 2;
        state(ST_SERVING);
    }
    else if(queue.empty())
    {
 Serving_Temp = Serving_Temp +
 (now() - Serving_Started);
      Serving_Started = 0;
      ID=send_delay(new CNEvent(EV_TIMER_I), DET_IDLE);
      Idling_Started = now();
      STATUS = 3;
      state(ST_IDLING);
    }
break;

case EV_TIMER_I:
   error("illegal event 'IDLING' in state ST_SERVING");
break;

case EV_TIMER_SET:
   error("illegal event 'SETUP' in state ST_SERVING");
break;
```

```
      default:
         error("illegal event in state ST_SERVING");
      break;
}

break;

case ST_IDLING:
   switch(ev->type())
   {
      case EV_JOB:
         // Incoming job
         delete_event(ID);
         Idling_Temp = Idling_Temp +
         (now() - Idling_Started);
         Idling_Started = 0;
         job = (CNJob *)ev->object();
         job->in    = now();
         job->start = now();

         // CNRandom service time
         send_delay(new CNEvent(EV_TIMER_S), rnd_b());
         Serving_Started = now();
         STATUS = 2;
         state(ST_SERVING);
      break;

      case EV_TIMER_I:
          //Idling time expired
          Idling_Temp = Idling_Temp +
          (now() - Idling_Started);
          t_idling.put(Idling_Temp);
          t_serving.put(Serving_Temp);
          t_setup.put(Setup_Temp);
          t_workcycle.put(now() - WorkCycle_Started);
          Idling_Started = 0;
          Idling_Temp = 0;
          Serving_Temp = 0;
          Setup_Temp = 0;
          WorkCycle_Started = now();
          Off_Started = now();
          STATUS = 0;

          state(ST_OFF);
      break;
```

```
                case EV_TIMER_S:
                    error("illegal event 'SERVING' in state ST_IDLING");
                break;

                case EV_TIMER_SET:
                    error("illegal event 'SETUP' in state ST_IDLING");
                break;

                default:
                    error("illegal event in state ST_IDLING");
                break;
            }
            break;



        default:
             error("mm1: ", "illegal state");
        break;

    }

}
void Server::eval_job(CNJob *job)
{
  t_w.put(job->start - job->in);
  t_b.put(job->out - job->in);
}


void Server::print_results()
{
  cout <<"Mean Delay: "<< t_b.mean() << endl;
  cout <<"OFF State: " <<t_off.mean() <<endl
       <<"SETUP State: "<<t_setup.mean()<<endl
       <<"BUSY State: " <<t_serving.mean()<< endl
       <<"IDLE State: " <<t_idling.mean()<<endl
       <<"Work-cycle: " <<t_workcycle.mean()<<endl
       <<"============================"<<endl;
}


int main()
{
```

```
float mu = 1;
float lambda = 5;
float DET_IDLE;
ofstream RandomResult;

float Mean_Energy = 0;
float Mean_Delay_Sum = 0;

float p = 0.5;
float Idling_array[]={0, 0.5, 1, 1.5, 2, 3, 5, 7, 10, 15, 20};

CNMoments delay("Delay");
CNMoments off_period("off_period");
CNMoments setup_period("setup_period");
CNMoments busy_period("busy_period");
CNMoments idle_period("idle_priod");
CNMoments work_cycle("work_cycle");
CNMoments total_energy_cost("Total_Cost");
CNMoments total_delay("Total_delay");



long arrival_seed[]={ 1192722770, 1641327678,
                      316473, 616180309,
                      1988517235, 279103050,
                      2017655168, 1705001958,
                      403982381, 1729728974 };
long service_seed[]={ 774252441, 256809583,
                      1650965283, 17001471,
                      805103753, 952596788,
                      749036158, 885527052,
                      1622578158, 1279399464 };

long prob_seed[]={ 1077708736, 1187340440,
                   1534916643, 958842836,
                   1227253782, 1617293094,
                   1732296971, 1592394575,
                   373268907, 576387424 };

for(int a = 0; a < 11; a++)
{
DET_IDLE = Idling_array[a];
for(int i = 0; i<10; i++)
{
```

```
        Mean_Energy = 0;
        Mean_Delay_Sum = 0;

        CNRNG   *rng_arr  = new CNMT;
        CNRNG   *rng_ser  = new CNMT;
        //CNRNG   *rng_set = new CNMT;
        CNRNG   *rng_prob = new CNMT;

        rng_arr->seed(arrival_seed[i]);
        rng_ser->seed(service_seed[i]);
        //rng_set->seed(setup_seed[i]);
        rng_prob->seed(prob_seed[i]);

        CNNegExp rnd_a(1/lambda, rng_arr);
        CNNegExp rnd_b(1/mu, rng_ser);
        //CNNegExp rnd_s(SETUP, rng_set);
        CNUniform rndP(0, 1, rng_prob);

        Server server_farm[Num_of_Servers] = { Server(rnd_b, DET_IDLE),
                                               Server(rnd_b, DET_IDLE),
                                               Server(rnd_b, DET_IDLE),
                                               Server(rnd_b, DET_IDLE),
                                               Server(rnd_b, DET_IDLE),
                                               Server(rnd_b, DET_IDLE),
                                               Server(rnd_b, DET_IDLE),
                                               Server(rnd_b, DET_IDLE),
                                               Server(rnd_b, DET_IDLE),
                                               Server(rnd_b, DET_IDLE) };


        Generator      generator(p, rndP, rnd_a, &server_farm);

        CNEventScheduler scheduler;
        scheduler.start(new CNEvent(EV_TIMER_G, &generator, 0.));


        //Statistics calculation per simulation

for(int j = 0; j < Num_of_Servers; j++)
if((server_farm[j].t_workcycle).mean() != 0)
        {
     Mean_Delay_Sum = Mean_Delay_Sum + (server_farm[j].t_b).mean();

             Mean_Energy = Mean_Energy +
                           (((server_farm[j].t_serving).mean()/
```

```
                        (server_farm[j].t_workcycle).mean())*P_on +
                        ((server_farm[j].t_setup).mean()/
                        (server_farm[j].t_workcycle).mean()) * P_on +
                        ((server_farm[j].t_idling).mean()/
                        (server_farm[j].t_workcycle).mean()) * P_idle);
        }

        //Statistics collection for total energy and delay costs


        total_energy_cost.put( Mean_Energy);
        total_delay.put(Mean_Delay_Sum);

    }


   cout<<"                                                    "<<endl;
   cout<<"                                                    "<<endl;

   cout<<"****************************************************"<<endl;
   cout<<"IDLING TIME: "<<DET_IDLE<<" SETUP TIME: "<<SETUP<<endl;
   cout<<"MEAN TOTAL DELAY COST : "<<total_delay.mean()<<endl;
   cout<<"MEAN TOTAL ENERGY COST : "<<total_energy_cost.mean()<<endl;
   cout<<"TOTAL COST: "<<total_delay.mean() +
          Beta * total_energy_cost.mean()<<endl;
   cout<<"****************************************************"<<endl;
   RandomResult.open("RandomResult.txt", ios::app);
   RandomResult<<"{"<<DET_IDLE<<","<<total_delay.mean() +
               Beta * total_energy_cost.mean()<<"},";
   RandomResult.close();
   }
   RandomResult.open("RandomResult.txt", ios::app);
   RandomResult<<"\nLOAD: "<<lambda/mu<<" SETUP: "<<SETUP<<endl;
   RandomResult.close();
   return 0;
}
```

A.2 MRB-RND task assignment policy

```
class Generator : public CNEventHandler
{
private:
  float prob;
  CNUniform rnd_p; //routing probability
  CNRandom &rnd_a;  // Distribution of arrival time a
  // Connected queue/server
  Server (&server_farm)[Num_of_Servers];

  CNSimTime MRB;
  unsigned int temp;
  int Index;
  int state_flag;
  long n;

public:
  virtual void event_handler(const CNEvent *ev);
  Generator(float pr, CNUniform rndP, CNRandom &rnd,
  Server (*serv)[Num_of_Servers]) : prob(pr), rnd_p(rndP),
  rnd_a(rnd), server_farm(*serv), MRB(0), temp(0), Index(0), state_flag(0),
    n(0){}
};


void Generator::event_handler(const CNEvent *)
{
  if(n == NJOBS)
    // Stop simulation
    return;

  // Incoming event -> generate new Job
  else
  {
      for(int j=0; j < Num_of_Servers; j++)
      {
          if(server_farm[j].STATUS == 3)
          {
              MRB = server_farm[j].Idling_Started;
              state_flag = 1;
              break;
          }
      }
```

```
        if(state_flag == 1) // MRB part
        {
           Index = 0;
           for(int k = 0; k < Num_of_Servers; k++)

               if( server_farm[k].STATUS ==3 &&
                   server_farm[k].Idling_Started >= MRB  )
               {
                    Index = k;
                    MRB = server_farm[k].Idling_Started;
               }
           send_now(new CNEvent(EV_JOB, &server_farm[Index], new CNJob));

         }
         else  // Random part
         {
            Index = int( (Num_of_Servers) * rnd_p.draw() );
            send_now(new CNEvent(EV_JOB, &server_farm[Index], new CNJob));
         }
         state_flag = 0;
    }

    // CNRandom delay
    send_delay(new CNEvent(EV_TIMER_G), rnd_a());
    n++;
}
```

A.3 MRB-JSQ task assignment

```
class Generator : public CNEventHandler
{
private:
  float prob;
  CNUniform rnd_p; //routing probability
  CNRandom &rnd_a;  // Distribution of arrival time a
  // Connected queue/server
  Server (&server_farm)[Num_of_Servers];

  CNSimTime MRB;
  unsigned int temp;
  int Index;
  int state_flag;
  long n;

public:
  virtual void event_handler(const CNEvent *ev);
  Generator(float pr, CNUniform rndP, CNRandom &rnd,
  Server (*serv)[Num_of_Servers]) : prob(pr), rnd_p(rndP),
  rnd_a(rnd), server_farm(*serv), MRB(0), temp(0), Index(0), state_flag(0),
    n(0){}
};


void Generator::event_handler(const CNEvent *)
{
  if(n == NJOBS)
    // Stop simulation
    return;

  // Incoming event -> generate new Job
  else
  {
      for(int j=0; j < Num_of_Servers; j++)
      {
          if(server_farm[j].STATUS == 3)
          {
              MRB = server_farm[j].Idling_Started;
              state_flag = 1;
              break;
          }
      }
```

```
        if(state_flag == 1) // MRB part
        {
           Index = 0;
           for(int k = 0; k < Num_of_Servers; k++)

               if( server_farm[k].STATUS ==3 &&
                   server_farm[k].Idling_Started >= MRB  )
               {
                    Index = k;
                    MRB = server_farm[k].Idling_Started;
               }
           send_now(new CNEvent(EV_JOB, &server_farm[Index], new CNJob));

         }
         else if(state_flag == 0)
         {
           temp = (server_farm[0].queue).length();
           Index = 0;
           for(int k = 0; k < Num_of_Servers; k++)

              if((server_farm[k].queue).length() < temp)
              {
                  Index = k;
                  temp = (server_farm[k].queue).length();
              }

            send_now(new CNEvent(EV_JOB, &server_farm[Index], new CNJob));
         }
         state_flag = 0;
    }

    send_delay(new CNEvent(EV_TIMER_G), rnd_a());
    n++;
}
```

A.4 BUSY/IDLE(t) with JSQ

```
int Idle_counter = 0;


enum { NJOBS=100000 };  // Limit for jobs to be processed

//event types
enum { EV_JOB, EV_TIMER_G, EV_TIMER_S, EV_TIMER_I, EV_TIMER_SET };

class Server : public CNEventHandler
{
private:
  unsigned long ID;      // Event ID
  CNJob *job;           // Served job
  //ofstream Statistics;
  //CNQueueFIFO queue;    // CNQueue
  CNRandom &rnd_b;      // Distribution of service time b
  //CNRandom &rnd_s;   // Distribution of setup time s
  int IDLE; // the threshold t, passed from main function (1 to 10)

  int flag;    // used between the SETUP and SERVING states
  enum { ST_IDLING, ST_SERVING, ST_SETUP, ST_OFF };

public:
  CNQueueFIFO queue;
  virtual void event_handler(const CNEvent *ev);
  void print_results();
  void eval_job(CNJob *job);

  CNMoments t_w, t_b;   // Evaluation tau_w, tau_b
   // mean time spent in each state: off, setup, serving and idle
  CNMoments t_off, t_setup, t_serving, t_idling, t_workcycle;
   // Used to capture the time at which a new workcycle is started
  CNSimTime WorkCycle_Started;
  //Used to capture the time at which a new idling perios is started
  CNSimTime Idling_Started;
  CNSimTime Setup_Started;
  CNSimTime Serving_Started;
  // Just to check completeness/correctness of work-cycle
  CNSimTime Off_Started;
  CNSimTime Setup_Temp;
  // captures intermediate values in the busy period calculation
  CNSimTime Serving_Temp;
```

```
  // captures intermediate values in the idle period calculation
  CNSimTime Idling_Temp;
  //CNSimTime Work_Cycle;
  //state of server used in Generator(OFF=0,SETUP=1,SERVING=2,
  //IDLING=3)
  int STATUS;
  int WC_Counter;
  Server(CNRandom &rnd, int IDLE_n)
    : job(NIL), rnd_b(rnd), IDLE(IDLE_n), flag(0), t_w("tau_w"),
    t_b("tau_b"), t_off("Turned Off"), t_setup("Setup_Time"),
    t_serving("Serving_Time"), t_idling("Idle_Time"),
    t_workcycle("Work Cycle"), WorkCycle_Started(0),
    Idling_Started(0), Setup_Started(0), Serving_Started(0),
    Off_Started(0), Setup_Temp(0), Serving_Temp(0), Idling_Temp(0),
    STATUS(0), WC_Counter(1)
    {
      state(ST_OFF);
    }
};

class Generator : public CNEventHandler
{
private:
  float prob;
  CNUniform rnd_p; //routing probability
  CNRandom &rnd_a;  // Distribution of arrival time a
  // Connected queue/server
  Server (&server_farm)[Num_of_Servers];

  CNSimTime MRB;
  unsigned int temp;
  int Index;
  int state_flag;
  long n;

public:
  virtual void event_handler(const CNEvent *ev);
  Generator(float pr, CNUniform rndP, CNRandom &rnd,
  Server (*serv)[Num_of_Servers]) : prob(pr), rnd_p(rndP),
  rnd_a(rnd), server_farm(*serv), MRB(0), temp(0), Index(0),
  state_flag(0),
  n(0){}
};

void Generator::event_handler(const CNEvent *)
```

```
{
  if(n == NJOBS)
  {
    for(int i=0; i < Num_of_Servers; i++)
    {
      server_farm[i].t_workcycle.put(now() -
      server_farm[i].WorkCycle_Started);
      server_farm[i].t_idling.put( server_farm[i].Idling_Temp);
      server_farm[i].t_serving.put( server_farm[i].Serving_Temp);
    }

    // Stop simulation
    return;
  }

  else
  {
      for(int j=0; j < Num_of_Servers; j++)
  {
      if(server_farm[j].STATUS == 3)
  {
      //cout<<server_farm[j].STATUS<<endl;
  Index = j;
  state_flag = 1;
  break;
  }
  }
      if(state_flag == 1)
  {
     send_now(new CNEvent(EV_JOB, &server_farm[Index], new CNJob));

      }
       else if(state_flag == 0)
      {
         temp = (server_farm[0].queue).length();
         Index = 0;
         for(int k = 0; k < Num_of_Servers; k++)
              if((server_farm[k].queue).length() < temp)
              {
                   Index = k;
                   temp = (server_farm[k].queue).length();
              }

     send_now(new CNEvent(EV_JOB, &server_farm[Index], new CNJob));
```

```
      }
      state_flag = 0;
  }

  send_delay(new CNEvent(EV_TIMER_G), rnd_a());
  n++;
}

void Server::event_handler(const CNEvent *ev)
{

  switch(state())
  {
    case ST_OFF:

      switch(ev->type())
      {
        case EV_JOB:
           CNJob *job;
           job = (CNJob *)ev->object();
           job->in = now();
           queue.put(job);
           if(queue.length() == K)
           {
              t_off.put(now() - Off_Started);
              Off_Started = 0;
              send_delay(new CNEvent(EV_TIMER_SET), SETUP);
              Setup_Started = now();
              STATUS=1;
              state(ST_SETUP);
           }

           else
              state(ST_OFF);
        break;

        case EV_TIMER_I:
           error("illegal event 'IDLING' in state ST_OFF");
        break;

        case EV_TIMER_SET:
           error("illegal event 'SETUP' in state ST_OFF");
        break;

        default:
```

```
        error("illegal event in state ST_OFF");
      break;

  }
break;

case ST_SETUP:

   switch(ev->type())
   {
      case EV_JOB:
        CNJob *job;
        job = (CNJob *)ev->object();
        job->in = now();
        queue.put(job);
      break;

      case EV_TIMER_SET:   //Setup completed
        flag = 1;
        Setup_Temp = now() - Setup_Started;
        Setup_Started = 0;
        send_now(new CNEvent(EV_TIMER_S));
        t_setup.put(Setup_Temp);

        Serving_Started = now();
        STATUS = 2;
        state(ST_SERVING);
      break;

      case EV_TIMER_I:
        error("illegal event 'IDLING' in state ST_SETUP");
      break;

      case EV_TIMER_S:
        error("illegal event 'IDLING' in state ST_SETUP");
      break;

      default:
        error("illegal event in state ST_SETUP");
      break;
   }

break;

case ST_SERVING:
```

```
switch(ev->type())
{
   case EV_JOB:
   // Incoming job, put into queue
      CNJob *job;
      job = (CNJob *)ev->object();
      job->in = now();
      queue.put(job);

   break;

   case EV_TIMER_S:

    // Timer event, service time run down
       switch(flag)
       {
          case 0:
              job->out = now();
              // Evaluate job if transient is over
              if(now() > 1000)
              {
                 eval_job(job);
              }
              delete job;
              job = NIL;

          break;

          case 1:
              flag = 0;
          break;

          default:
               error("incorrect value for flag");
          break;
       }

       // Get new job from queue
       if(!queue.empty())
       {
          job = (CNJob *)queue.get();
          job->start = now();
          // CNRandom service time
          send_delay(new CNEvent(EV_TIMER_S), rnd_b());
```

```
      STATUS = 2;
      state(ST_SERVING);
   }

   else if(queue.empty())
   {
      Serving_Temp = Serving_Temp +
      (now() - Serving_Started);
      Serving_Started = 0;
      if(Idle_counter < IDLE)
      {
          Idle_counter++;
          Idling_Started = now();
          STATUS = 3;
          state(ST_IDLING);
      }

      else
      {
          t_idling.put(Idling_Temp);
          t_serving.put(Serving_Temp);
          Idling_Temp = 0;
          Serving_Temp = 0;
          t_workcycle.put(now() - WorkCycle_Started);
          Idling_Started = 0;
          Idling_Temp = 0;
          Serving_Temp = 0;
          Setup_Temp = 0;
          WorkCycle_Started = now();
          Off_Started = now();
          STATUS = 0;
          state(ST_OFF);
      }
   }
break;

case EV_TIMER_I:
   error("illegal event 'IDLING' in state ST_SERVING");
break;

case EV_TIMER_SET:
   error("illegal event 'SETUP' in state ST_SERVING");
break;

default:
```

```
            error("illegal event in state ST_SERVING");
        break;
    }

    break;

    case ST_IDLING:

        switch(ev->type())
        {
            case EV_JOB:
                // Incoming job
                Idling_Temp = Idling_Temp + (now() - Idling_Started);
                Idling_Started = 0;
                Idle_counter--;
                job = (CNJob *)ev->object();
                job->in    = now();
                job->start = now();

                // CNRandom service time
                send_delay(new CNEvent(EV_TIMER_S), rnd_b());
                Serving_Started = now();
                STATUS = 2;
                state(ST_SERVING);
            break;

            case EV_TIMER_S:
                error("illegal event 'SERVING' in state ST_IDLING");
            break;

            case EV_TIMER_SET:
                error("illegal event 'SETUP' in state ST_IDLING");
            break;

            default:
                error("illegal event in state ST_IDLING");
            break;
        }
    break;

    default:
        error("illegal state");
    break;
  }
}
```