Guidelines for the Design of Flexibility in Queueing Systems: Model, Measures and Analysis

by

Suryanarayanan Gurumurthi

B.E., Mechanical Engineering (1997)
Birla Institute of Technology, India

M.S.I.E., Industrial Engineering (2001)
University of Minnesota

Submitted to the MIT Sloan School of Management
in Partial Fulfillment of the Requirements for the Degree of
Master of Science

at the

Massachusetts Institute of Technology

September 2004

Signature of Author ....[illegible]..
MIT Sloan School of Management
August 31st 2004

Certified by ....[illegible]..
Prof. Stephen C. Graves
Abraham J. Siegel Professor of Management Science & Engineering Systems
MIT Sloan School of Management

Accepted by ....[illegible]..
Prof. Birger Wernerfelt
J. C Penney Professor of Management Science
MIT Sloan School of Management

GUIDELINES FOR THE DESIGN OF FLEXIBILITY IN QUEUEING SYSTEMS: MODEL, MEASURES, AND ANALYSIS

by
Suryanarayanan Gurumurthi

Submitted to the MIT Sloan School of Management
on August 31st 2004, in partial fulfillment of the requirements for the degree of
Master of Science

ABSTRACT

An analytical and computational framework is presented that has been developed for the performance analysis of arbitrary queueing networks with multiple heterogeneous servers and multiple customer classes, where customers have the flexibility of being processed by more than one server and servers possess the capability of processing more than one customer class. Jobs of a given class may arrive according to an independent Poisson process to a facility consisting of multiple heterogeneous servers. The service time for the processing of any given job class at any given server is assumed to be exponentially distributed with a mean that could vary by job class as well by server. Significantly, we do not impose any restriction on the set of job classes that can be processed by any given server. Assuming finite work-in-process capacity in terms of the number of jobs already in the system, we allow for multiple stages of processing in the queueing system.

In order to motivate the research, we first identify the different forms of flexibility in such queueing systems that are relevant to managers given their importance as design factors and control policies for higher performance. Next, we present an analytical framework whose goal is to capture for performance analysis, the relative impact of the different forms of flexibility so identified. Third, we demonstrate the usefulness of the modeling framework through a simple but illuminative numerical analysis of single-stage queuing systems that in turn shows the significance of these flexibility mechanisms to the performance measures of interest to system managers.

In terms of insights from the modeling efforts, we first show that when evaluated within this framework, control policies such as job-routing and job-selection rules have a relatively limited impact on throughput and overall utilization measures when compared to strategic flexibility design parameters such as the assignment of long-term job responsibilities to servers. However, we show that this influence on performance is significant enough that various flexibility design alternatives are better compared after taking into account the control policy that will be used to operate the system. Furthermore, and motivated by the recent interest in revenue management techniques for operational systems, we show that such operational control policies can have disproportionate influence on revenue and cost measures of performance; this fact further underscores the importance of having such models, measures, and analytical tools to examine various system design alternatives for improving performance.

Acknowledgements

I am very grateful to the Operations Management area faculty members for their valuable time and assistance. In particular, I thank Prof. Stephen Graves for his time over the past two years, and for his willingness to help me address my academic and personal concerns. I am very grateful for his generosity in offering me a teaching assistantship during Spring 2004 that not only gave me the chance to participate in his course, but also the opportunity to continue my studies at MIT.

I am also thankful to Prof. Thomas Roemer for his time and for the discussions that enabled me to perform an internship after my first year here. I am similarly grateful to Prof. Gabriel Bitran for his kind gesture in offering me a teaching assistantship for Summer 2004, and therefore the opportunity to participate in his course. I thank Prof. Saif Benjaafar of the University of Minnesota for his guidance when I was a student there, on some of the ideas that are presented here in this thesis. I also thank my fellow students, from whom I have learnt a great deal. I am grateful to Sharon Cayley for being patient with, and very knowledgeable about the administrative details of my studies.

Table of Contents

Acknowledgements 3

Table of Contents 4

1 Introduction and Literature Review 5

2 Flexibility Mechanisms for Queueing Systems 10
2.1 Strategic Flexibility Mechanisms 11
2.2 Tactical Flexibility Mechanisms 12
2.3 Operational Flexibility Mechanisms 14
2.4 Guidelines for the Design of Flexibility Mechanisms 17

3 A Markov Modeling Framework for Queueing Systems 19
3.1 The State Space 22
3.2 Performance Evaluation 25
3.3 Performance Measures 44

4 Numerical Examples and Insights on Single-Stage Systems 46

5 Conclusions and Future Work 62

6 References 63

Appendix A1 : Computer Code for Single Stage Analysis 65
Appendix A2 : Computer Code for Multi-Stage Analysis 70

1 Introduction and Literature Review

In this thesis we consider the representation, modeling and analysis of flexiblility in queueing systems. We study systems with heterogeneous servers and multiple customer or job classes where each job class has the flexibility of being processed by more than one server and servers in turn possess the capability of processing more than one job class. Customer or job classes can vary in demand rate and routing flexibility. Servers can vary in service rates and service flexibility. The dynamic assignment of a job to servers is determined by a server selection rule, that can be job class-specific, and the selection of the next customer to serve is determined by a queue selection rule, that can be server-specific. Further, a job upon completion of service can either leave the system, or can return for processing as a different job class with different processing requirements than before. An example of such a queueing system is shown in Figure 1 that consists of four job classes $P_1, P_2, P_3,$ and P_4, and five servers $R_1, \ldots, R_5$ with service times that are exponentially distributed with means $1/\mu_1, \ldots, 1/\mu_5$ respectively. Job classes P_1 and P_2 arrive according to a Poisson process with mean rates λ_1 and λ_2 respectively. Each job class has a pre-specified set of servers that are capable of processing jobs of that class and in the figure these sets are defined by the arcs connecting the queue representing jobs of a particular class to the servers. Jobs that are processed at servers $R_2, R_4,$ and R_5 leave the system immediately. On the other hand jobs that are processed at servers R_1 and R_3 return to the system as job classes P_3 and P_4 respectively. Jobs of any class that do not find an available server to begin processing wait in their designated queues with capacities defined as $b_1, \ldots, b_5$. Finally, we may assume that arriving jobs are balked from the system if their designated buffer or queue is already at capacity; whereas if jobs that wish to return for processing as a different class find the buffer or queue for that class at capacity, they maintain their current identity and request repeat processing.

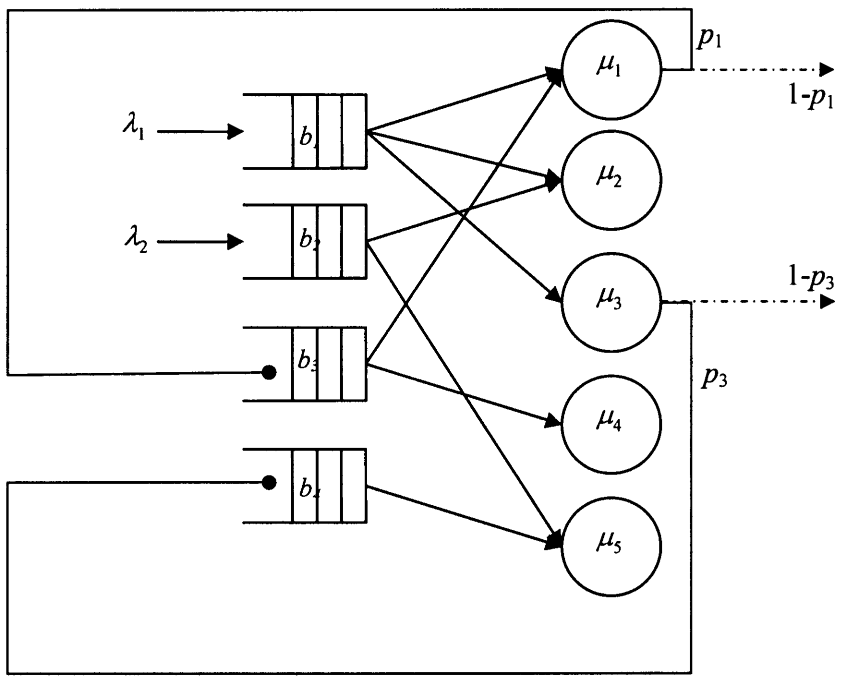


Figure 1 – An example of a flexible queueing system

Queueing systems, similar to the one just described are found in manufacturing systems (Buzacott and Shanthikumar, 1992), telecommunication networks (Ross, 1995), and service operations (Hall, 1991). It is interesting to note that one source of complexity as well as convenience in such systems is the *process flexibility* that is defined for each job class. In manufacturing systems, there is often flexibility in routing demand for different product or job classes to one or more functionally equivalent pieces of equipment, each with different processing characteristics such as speed, cost or quality of processing. In fact, such process flexibility is also observed in manufacturing supply chains where different plants or facilities are tooled for different sets of products in keeping with strategic supply chain performance measures and objectives (Jordan and Graves, 1995). One can observe qualitatively similar considerations in the management of service operations, where for example call centers are staffed by operators with varying skills who are capable of handling some or all of the call types (Koole and Mandelbaum, 2002) (Whitt, 2002). The layout and design of telecommunication networks similarly involves decisions of flexibility, but in a different sense, where the objective is to retain greater *routing flexibility* for managing requests for data transfer between any two nodes in the network using multiple link paths between the nodes (Ross, 1991).

In this research, we provide a modeling framework for the analysis of general queueing systems or networks with an arbitrary number of server and job types and arbitrary process flexibility. We consider a varied set of control policies that includes strict priority schemes for job routing to the servers and for queue selection, and for demonstration purposes a dynamic policy in the form of the longest queue first policy. In fact we see the analysis of and search for effective dynamic control policies as a promising line of research that could be motivated by this work. The queue capacity may be specific to a job class or in the form of a global bound on the total number of customers in the system. To our knowledge, this work is the first to provide such a state representation and the accompanying compact description of the dynamics of such queueing systems with general customer and server flexibility and with heterogeneous servers.

It is to be noted at the outset that our models are applicable only to systems with finite queue or buffer capacity, where this has obvious implications for the size of state space to represent the operation of queueing system. While our modeling representation and framework might result in computational hurdles that inhibit the ready application of these models to real world or industrial settings, the motivation for this research is different. For this work, we are motivated by a need to highlight the critical performance measures and objectives that are of potential interest to system managers, along with the need to outline the key design parameters and control policies or levers that allow system managers to improve along such performance measures.

To illustrate the usefulness of our models, we carry out a numerical study of single-stage queueing systems that are a special case of the general multi-stage systems that we describe here. Specifically we examine the inter-relationships between throughput as a performance measure for such systems with finite queue capacity, the process flexibility of the system as defined by the assignment of customer and job classes to servers, and such system parameters as heterogeneity among the servers and the customer classes and the loading levels. As has been shown previously, we show that higher flexibility does not always improve throughput (Gurumurthi and Benjaafar, 2004)

(Hopp, Tekin, and van Oyen, 2001). In this research, we go further to try and show that effective control policies for an arbitrary queueing system could be devised based on the process flexibility of the system, the available capacity to the various job classes based on their assigned servers, and the loading levels for the servers based on the specific job classes that are assigned to them. More significantly, we try to show indirectly that the problem of determining appropriate levels of process flexibility for the queueing system requires an understanding of the impact of control policies that will be used for the system. In other words flexibility configurations that work well for one type of control policy can lead to inferior performance for the same queueing system when operating under a different control policy. Therefore, control policies have a certain measure of influence on the performance of such queueing systems and given that their impact on performance is not yet well understood, we need to explicitly account for the influence of control policies in order to determine the desirability of certain process flexibility configurations over others.

This leads to a discussion of the reasons behind the importance of process flexibility decisions to system managers.

1. There are both strategic (long term) as well as short term implications from decisions concerning flexibility. The long term implications arise out of the fact that often times designing additional process flexibility is an expensive, time-taking, and potentially disruptive process that requires system managers to approach the task as an investment decision for the firm (Fine and Freund, 1993). The short term implications, which are also highlighted by this current work, include the need for altering control policies to suit the new process flexibility configurations, as well as the system performance that results from the fit between the control policies and the flexibility configuration.
2. Measuring performance accurately for queueing systems with arbitrary process flexibility configurations operating under commonly used control policies, is not in general a task that is easy to accomplish for system managers. Apart from the computational or analytical challenges that the literature in queueing theory attempts to address, there is also the issue of developing operational models to

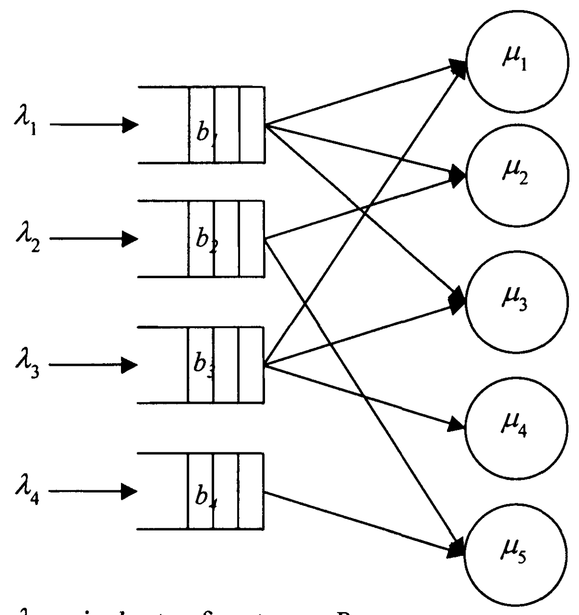
resemble the real-life queueing system, at a suitable level of abstraction in order to measure performance.

3. Given that measuring performance is a not an easy task, and given that there could be costs associated with increased flexibility, determining efficient process flexibility configurations (that requires in turn an optimization approach) is also therefore a difficult task for system managers. All the same, it might be important for system managers to notice that decisions involving flexibility of servers and the design of control policies are an aspect of their work that requires careful attention and consideration.
4. Finally, the problem of determining efficient process flexibility configurations is compounded by the fact that at different levels of abstraction in the model of the system, the results may not be consistent, and may indeed present contradictions. For example, at a level of abstraction where control policies are not considered as a factor, an optimal matching between overall system supply (capacity) with demand from different customers may point to the feasibility of a particular process flexibility configuration. However, when we consider explicitly the influence of control policies, and when we evaluate the different flexibility alternatives within the subset of control policies, we might arrive at different conclusions.

The remainder of this thesis is organized as follows. In section 2, we provide a brief discussion of the various forms of flexibility that could be of interest to system managers given our view of them as strategic or tactical design factors and operational control mechanisms or policies that have a direct bearing on system performance. In Section 3, we present our model and present some of the basic performance measures that are captured by our model. In Section 4, we discuss briefly numerical results and several insights from a computational analysis of single-stage queueing systems that are a special case of the multi-stage model presented in section 3. In section 5, we summarize our results and offer some concluding comments, and in Section 6 we present the the list of references used for this work. Finally, Appendices A1 and A2 contain up-to-date versions of computer code that has been written in order to perform the numerical analysis.

2 Flexibility Mechanisms for Queueing Systems

In this section, we outline the various forms of flexibility that are available as mechanisms for performance improvement to system managers in a variety of contexts including manufacturing and service operations. We classify these flexibility mechanisms as being strategic, tactical or operational in their timing and implications from a planning and execution viewpoint. However, given the broad nature of the discussion without reliance on a specific operational context, there could be some cross-over in terms of how flexibility mechanisms for a particular context, say for example health-care operations involving medical equipment, fit into such taxonomy. Consider for a base system, the single stage queueing system described in Figure 2.



λ_i: arrival rate of customer P_i
μ_i: service rate of server R_i
b_i: buffer size for customer P_i

Figure 2 – An example of a single-stage flexible queueing system

2.1 Strategic Flexibility Mechanisms

If in the system shown in Figure 2, the demand arising from any job class is considered as exogenous and the arrival process as independent of the system, we can observe that strategic process flexibility is the result of two considerations that are part of the same decision process. Firstly, we have the problem of allocating the demand from the various job classes to the servers; since however in the example shown, the queues are organized by job class, and more specifically not by server, the problem of demand allocation in this framework is the same as the problem of cross training or tooling of the servers for the various job classes. The resulting configuration of job classes that are assigned to one or more servers is what we refer to as process flexibility in the system. From a planning standpoint, these decisions are often strategic in nature, since investments made in cross-training and tooling may be of a long-term nature in their payoffs to system managers. Figure 3 illustrates the concept of strategic process flexibility in the single-stage system under consideration in this section.

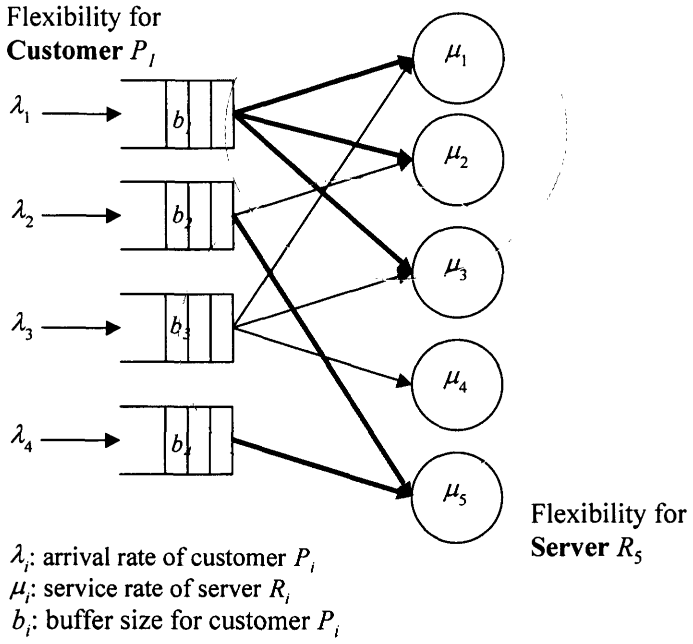
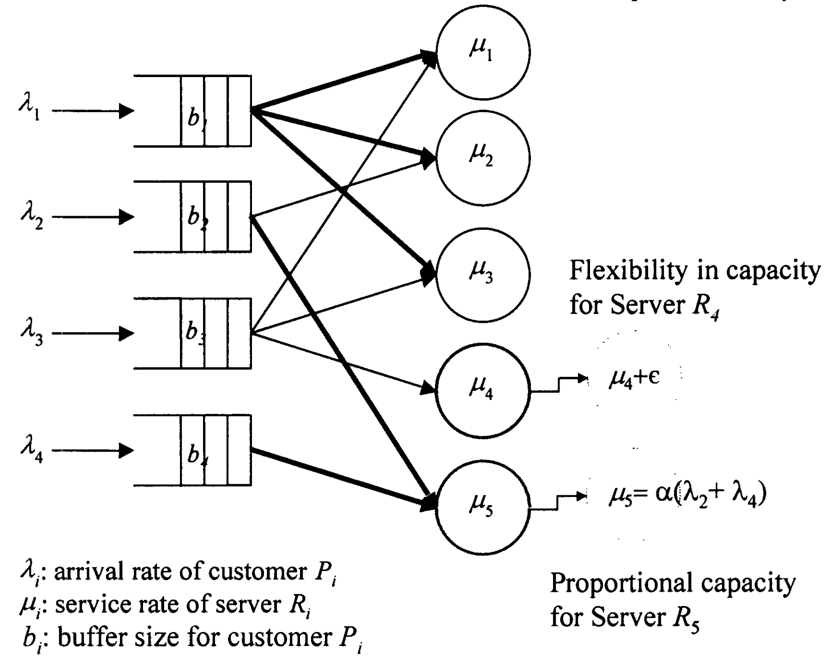

Flexibility for **Customer** P_I
λ_1
λ_2
λ_3
λ_4
b_1
b_2
b_3
b_4
μ_1
μ_2
μ_3
μ_4
μ_5
Flexibility for **Server** R_5
λ_i: arrival rate of customer P_i
μ_i: service rate of server R_i
b_i: buffer size for customer P_i

Figure 3 – Strategic flexibility: demand allocation and cross-training (tooling)

Secondly we can also extend the scope of strategic flexibility mechanisms to include the determination of whether additional servers are required in the system. For strategic design, the literature can be grouped around two central questions: (1) how many servers should we have, and (2) how much flexibility should each server have, and therefore how much routing flexibility should we provide to each customer class. Issues pertaining to questions 1 and 2 are also generally referred to as capacity allocation. For a review of important applications that involve such strategic decisions, we refer to Kleinrock (1976) and Buzacott and Shanthikumar (1992). For a similar review that specializes on call center operations, we refer to Gans et al. (2003) and Whitt (2002). In supply chain settings, Jordan and Graves (1995) discuss strategic flexibility within the framework of supply chain decisions that have long term implications, such as cross-tooling of plants for various product lines.

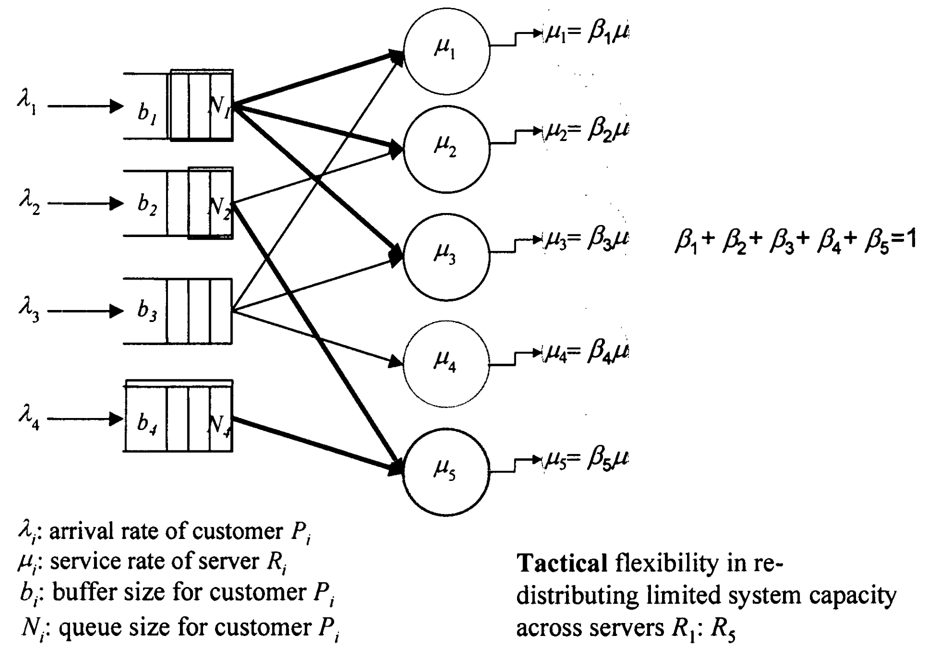
2.2 Tactical Flexibility Mechanisms

If in the system shown in Figure 2, the capacity that is allocated to each server were a decision variable, and if the demand for the various job classes and the process flexibility were fixed, then the capacity allocation decision can be viewed as a tactical flexibility mechanism that is available to system managers. In the literature, one typically finds that these questions are posed together with questions on how many servers one should have for meeting the exogenous demand arising from various job classes. We view the capacity allocation decision as presenting three different flexibility mechanisms to system managers: (*i*) the allocation of capacity proportional to demand assigned to the servers (*ii*) the incremental allocation of additional capacity to a server, or the augmentation of service capacity, and (*iii*) the fractional allocation of fixed system capacity through a re-distribution of the system capacity amongst the servers. Figures 4 and 5 illustrate these capacity allocation schemes for the base example we consider in this section. From a system design view-point the critical parameters that define tactical flexibility mechanisms are denoted by α in the case of proportional allocation, β in the case of fractional allocation, and ε in the case of incremental allocation of capacity to the servers. These decisions are termed as tactical, if only for the reason that they are presented in our work as being conditional on the demand allocation and on the process flexibility decisions; in practice, we find that tactical flexibility involves decisions that can be reversed and / or altered in the near term, without disruption to the system.



λ_1
λ_2
λ_3
λ_4
b_1
b_2
b_3
b_4
μ_1
μ_2
μ_3
μ_4
μ_5
Flexibility in capacity for Server R_4
$\mu_4+\epsilon$
$\mu_5=\alpha(\lambda_2+\lambda_4)$
λ_i: arrival rate of customer P_i
μ_i: service rate of server R_i
b_i: buffer size for customer P_i
Proportional capacity for Server R_5

Figure 4 – Tactical flexibility: capacity allocation, proportional and incremental



λ_1
λ_2
λ_3
λ_4
b_1
b_2
b_3
b_4
N_1
N_2
N_4
μ_1
μ_2
μ_3
μ_4
μ_5
$\mu_1=\beta_1\mu$
$\mu_2=\beta_2\mu$
$\mu_3=\beta_3\mu$
$\mu_4=\beta_4\mu$
$\mu_5=\beta_5\mu$
$\beta_1+\beta_2+\beta_3+\beta_4+\beta_5=1$
λ_i: arrival rate of customer P_i
μ_i: service rate of server R_i
b_i: buffer size for customer P_i
N_i: queue size for customer P_i
Tactical flexibility in re-distributing limited system capacity across servers R_1: R_5

Figure 5 – Tactical flexibility: fractional allocation of capacity

Another form of tactical flexibility that is available to system managers is the physical or logical reorganization of the queues or buffers in the system that serves the purpose of risk pooling in the case of finite buffer capacities. Instead of maintaining separate buffers of finite capacity for each job class, where possible the jobs are held while waiting in single buffers of the same overall capacity as was available previously. Figure 6 illustrates this mechanism; while the concept is simple, in reality this may involve a physical reorganization of flows in the operations facilities, and as such we have classified this as a tactical measure.

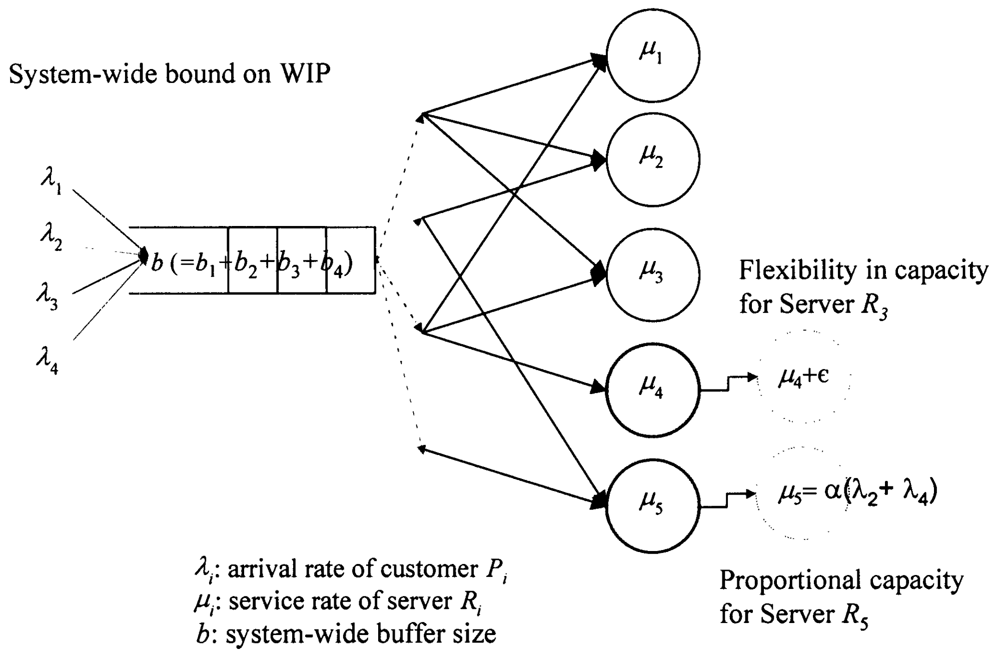

System-wide bound on WIP
λ_1
λ_2
λ_3
λ_4
$b\ (=b_1+b_2+b_3+b_4)$
μ_1
μ_2
μ_3
μ_4
μ_5
Flexibility in capacity for Server R_3
$\mu_4+\epsilon$
$\mu_5=\alpha(\lambda_2+\lambda_4)$
Proportional capacity for Server R_5
λ_i: arrival rate of customer P_i
μ_i: service rate of server R_i
b: system-wide buffer size

Figure 6 – Tactical flexibility: pooling of buffers

2.3 Operational Flexibility Mechanisms

Our definition of operational flexibility mechanisms is based on the dynamic view of the queueing system. Mechanisms that are a response to the state of the queueing system at a given point in time are classified as being operational in their nature. As such they represent the response to current conditions in the operational facility such as the number of jobs that are in a given buffer, or whether a particular server is busy, idle, or if there is an interruption in service for various reasons. The literature has also viewed such response mechanisms as being tactical in nature, but once again, we emphasize that apart from semantics, our understanding and representation of such mechanisms is similar in nature to that found in the literature. One prevalent flexibility mechanism in response to workload levels at the various buffers is the allocation of capacity to servers that is proportional to the workload in the queue; this concept is illustrated in Figure 7. An excellent example of literature that illustrates this concept is to be found in Graves (1986)

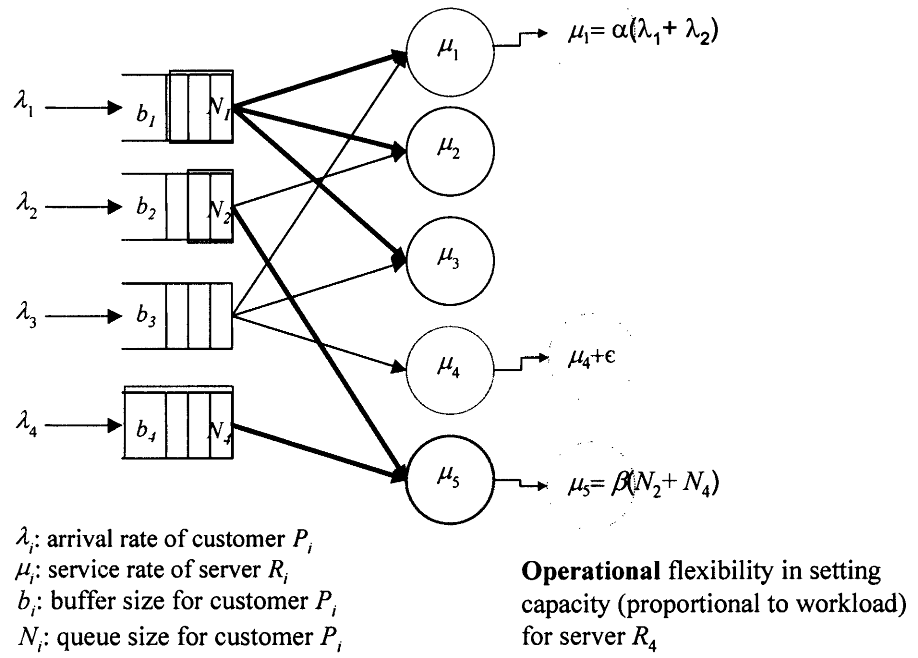
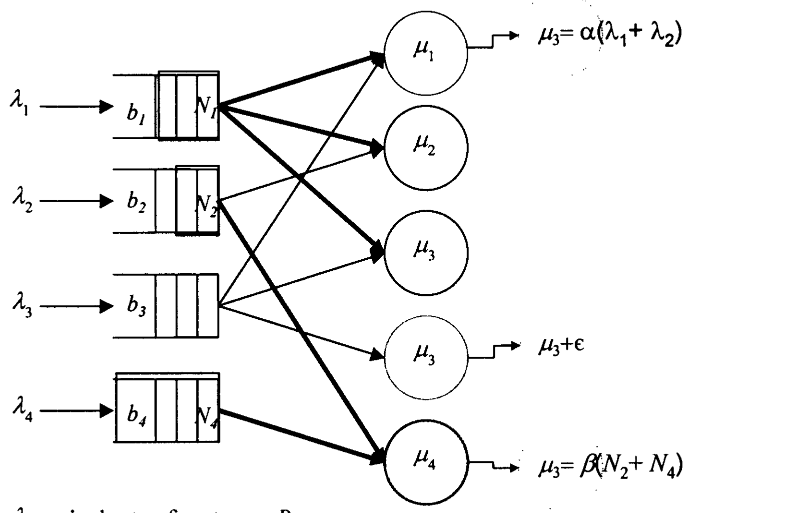

λ_1
λ_2
λ_3
λ_4
b_1
b_2
b_3
b_4
N_1
N_2
N_4
μ_1
μ_2
μ_3
μ_4
μ_5
$\mu_1 = \alpha(\lambda_1 + \lambda_2)$
$\mu_4 + \epsilon$
$\mu_5 = \beta(N_2 + N_4)$
λ_i: arrival rate of customer P_i
μ_i: service rate of server R_i
b_i: buffer size for customer P_i
N_i: queue size for customer P_i
Operational flexibility in setting capacity (proportional to workload) for server R_4

Figure 7 – Operational flexibility: capacity allocation proportional to workload

Finally, an important class of operational flexibility mechanisms is available to system managers in the form of dynamic job sequencing and job routing policies in such queueing systems. In other words, based on the current state of the system defined in terms of state of the servers, and in terms of the workload in each buffer; system managers can and indeed avail of the flexibility in routing an incoming job to a server from the pool of available servers; or in selecting from the set of jobs in queue from various classes, one particular job to process at a server that has become available. In this research, we refer to the combination of job selection and job routing policies as a *control* *policy* for the queueing system. A visualization of control policies as operational flexibility mechanisms is provided in figures 8a and 8b.

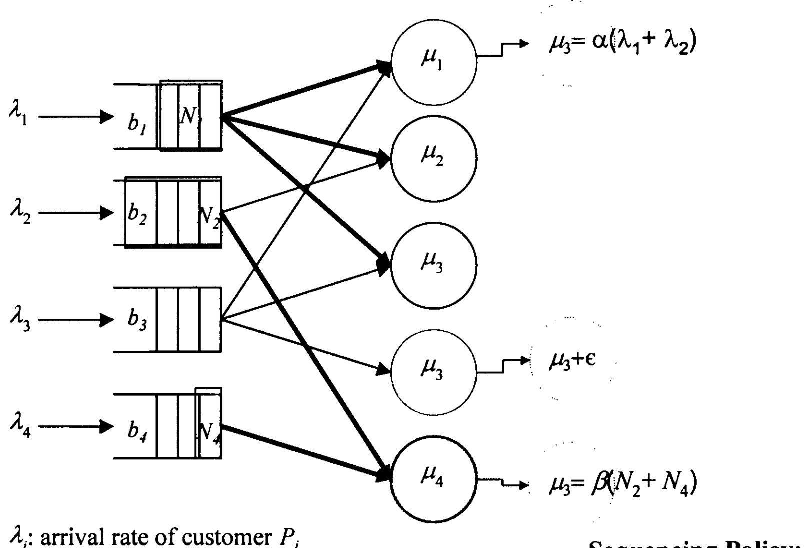


λ_i: arrival rate of customer P_i
μ_i: service rate of server R_i
b_i: buffer size for customer P_i
N_i: queue size for customer P_i

Sequencing Policy:

$\boldsymbol{P_4 > P_1 > P_2}$

Figure 8a – Operational flexibility: dynamic sequencing policy (1)



λ_i: arrival rate of customer P_i
μ_i: service rate of server R_i
b_i: buffer size for customer P_i
N_i: queue size for customer P_i

Sequencing Policy:

$\boldsymbol{P_2 > P_1 > P_4}$

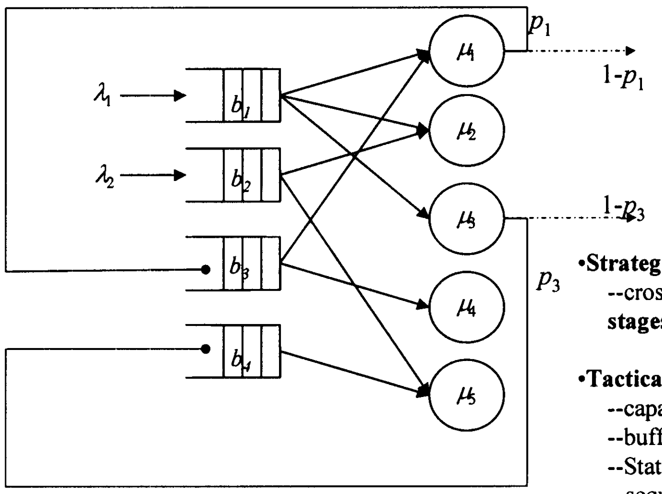
Figure 8b – Operational flexibility: dynamic sequencing policy (2)

One of the goals of this research is to demonstrate the significance of operational control policies on the performance of the queueing system; indeed as we demonstrate in the numerical analysis section operational flexibility can have very much the same levels of influence on performance as strategic or tactical flexibility mechanisms. Further, our research on this subject over the past few years seems to point to the need for the design of flexibility in the strategic, tactical and operational sense to be addressed in an integrated fashion. In other words, the choice of control policy may influence the design of cross-training programs, and conversely, a good choice of control policy appears to be conditional on the process flexibility configuration and the tactical capacity management mechanisms in place.

2.4 Guidelines for the Design of Flexibility Mechanisms

The goal of this research is to arrive at a set of guidelines that will (*i*) help system managers understand the relative and specific importance of flexibility mechanisms just described, to performance measures of relevance to their individual operational settings, (*ii*) demonstrate that there could be multiple performance criteria, metrics or objectives that could be applied to these systems that could result in different design choices, and (*iii*) develop insight on the trade-offs that eventually occur in the selection of appropriate flexibility mechanisms from the set of alternatives in front of managers, when we apply the different metrics to evaluate the set of design alternatives. It is important to note here that the performance and behavior of such arbitrary queueing systems is not in general easy to characterize from the point of view of system managers. We also recognize the combinatorial nature of the search space of design alternatives, and hence we are motivated to first develop additional insight into the behavior of these systems in general, rather than delving first into algorithms for determining efficient process flexibility configurations, or optimal control policies. At the same time, an algorithmic approach to the design of such queueing systems could be a fruitful line of research, in our opinion, and we refer again to this issue in section 5.

These considerations have resulted in the modeling framework that we present in the subsequent section. This modeling framework is capable of capturing explicitly the effect of many of the flexibility mechanisms that we describe here on the performance of such queueing systems. Where some mechanisms have not been explicitly shown or discussed, it will become apparent to the reader that minor extensions of our modeling framework can address some of those shortcomings. Figure 9 summarizes the flexibility mechanisms we have previously discussed in a single stage setting, for the operation of a multi-stage queueing system. Figure 9 can also be used as a reference to understand the motivation for and the details involved in developing a modeling framework for the performance analysis of such systems.



- **Strategic Flexibility**
  - --cross-training, or cross-tooling **across stages**, and also across products
- **Tactical Flexibility**
  - --capacity allocation schemes
  - --buffer design for various stages
  - --Static priority schemes for sequencing and scheduling
- **Operational Flexibility**
  - --dynamic capacity allocation schemes
  - --dynamic sequencing and scheduling

λ_i: arrival rate of customer P_i
μ_i: service rate of server R_i
b_i: buffer size for **stage** P_i

Figure 9 – Flexibility mechanisms for multi-stage queueing systems

3 A Markov Modeling Framework for Queueing Systems

Consider a queueing system, X, consisting of m servers and n job classes. Jobs of class i (i = 1, ..., n) arrive to the system according to an independent Poisson process with rate λ_i. Processing times for job class i at server j (j = 1, ..., m) are exponentially distributed and *i.i.d.* with mean $1/\mu_{ij}$. Each server is capable of processing one or more job classes and each job class can be processed by one or more servers. Let $M = \{R_1, R_2, \dots R_m\}$ be the set of servers in the system and $P = \{P_1, P_2, \dots P_n\}$ be the set of job classes.

The initial feasible job-server assignments are denoted by an $n \times m$ routing matrix $A = [a_{ij}]$, where

$$a_{ij} = \begin{cases} 1, & \text{if part } P_i \text{ can be processed by server } R_j; \\ 0, & \text{otherwise.} \end{cases} \tag{1}$$

We define a set of servers Q_i associated with each job class P_i, such that this job class can be processed by any of the servers in Q_i:

$$Q_i = \{R_{i(1)}, R_{i(2)}, \dots, R_{i(m_i)}\},$$

where $i(k)$ denotes the index of the k^{th} server assigned to job class P_i. We let $|Q_i| = \sum_{i=1}^{m} a_{ij}$ denote the cardinality of the set Q_i. Similarly, we define T_j to be the set of job classes that can be processed by server R_j such that:

$$T_j = \{P_{j(1)}, P_{j(2)}, \dots, P_{j(n_j)}\},$$

where $j(k)$ denotes the index of the k^{th} job class assigned to server R_j and $|T_j| = \sum_{i=1}^{n} a_{ij}$ is the cardinality of set T_j.

The operation of the job-shop like queuing network is described as follows. Jobs of class i (i = 1, ..., n), upon arrival to the system, seek a server from set Q_i for service. If all of the servers from the set Q_i are busy, then the job of class i is placed in its assigned queue. However if there are one or more servers from the set Q_i that are available, the job is then serviced by one of the servers. Upon completion of service, this job is then transformed with probability t_{ik} (k=1,..., n) into a job class k ($\neq i$), and with probability $1 - \sum_{k=1\ .n;k\neq i} t_{ik}$, leaves the system. The resulting $n \times n$ matrix of transition probabilities between job classes is denoted by $\boldsymbol{\Phi}$. In this framework, while we do not discount the possibility of valid transitions of a job to its own class (in an attempt to model rework), we allow a return for processing within the same class if at the time of transition between job classes, the buffer or WIP for the target job class k is already at capacity and therefore if the inter-class transition were unsuccessful. In such an event, the conditional probability for a job returning to its class given that a target job class buffer B_k is at capacity is also t_{ik}. However using only the matrix $\boldsymbol{\Phi}$, it is also possible in the general case to characterize the distribution of the number of stages traversed before a job of class i returns to this class, assuming only a sequence of *successful* class transitions. Therefore, we can define the queueing network as being closed reentrant, if the Markov Chain defined by transition probabilities $\boldsymbol{\Phi}$ is recurrent. If the Markov Chain defined by $\boldsymbol{\Phi}$ is transient with non-zero (< 1) probability of recurrence to class i, then we will term the network as being open reentrant. Finally, if jobs of class i never return to the same class, then we define the system as being a serial or tandem queuing network.

In addition to specifying feasible job-server assignments, the analysis of the queueing system requires the specification of a control policy. The control policy is applied at each decision epoch. Decision epochs are triggered by either the arrival of a job or the completion of service by a server. When a job arrives and finds multiple idle servers, the control policy specifies which server is selected. Similarly, when a server completes service and finds jobs of more than one class in the queue, the control policy specifies which job is selected next for service. Hence a control policy is defined by a server selection rule and a job selection rule.

In this framework, we consider static server selection rules, where server preferences can be specified in terms of a priority scheme for each job class. For each job class and for each server, we associate a priority $\alpha(P_i, R_j) \in \{1, 2, \dots m\}$, which, for notational compactness, we shall heretofore denote as α_{ij}, where priority is higher for lower values of α_{ij}. If there is competition between two or more idle servers for a job of class P_i, the job is assigned to the server with the lower value of α_{ij}. Special cases of the priority scheme include the strict priority (SP) rule where $\alpha_{ij} \neq \alpha_{ik}$ for all values of i, j and k (such that $j \neq k$; $a_{ij}=a_{ik}=1$) and the random routing (RR) rule where $\alpha_{ij} = \alpha_{ik}$ for all values of $i, j,$ and k (such that $a_{ij}=a_{ik}=1$). In all cases, ties are broken arbitrarily.

For job selection, we consider a dynamic rule under which a server, upon becoming available, always selects a job from the class with the longest queue from the set of feasible job classes. Among jobs from the same class, jobs are served on a first in first out (FIFO) basis. We term this rule the longest queue first (LQF) rule. We also consider static job selection rules, where jobs are selected based on a priority scheme. Specifically, for each job class and for each server, we associate a priority $\gamma(P_i, R_j) \in \{1, 2, \ldots n\}$, or more simply γ_{ij}. Upon becoming idle, a server R_j selects a job from the class with the lowest value of γ_{ij}. Within each class, jobs are again ordered on a first in first out (FIFO) basis. Special cases of priority schemes include the *strict priority* (SP) rule where $\gamma_{ij} \neq \gamma_{kj}$ for all values of i, j and k (such that $j \neq k$; $a_{ij}=a_{kj}=1$) and the *random service* (RS) rule where $\gamma_{ij} = \gamma_{kj}$ for all values of $i, j,$ and k (such that $a_{ij}=a_{kj}=1$).

Although static, fixed priority rules allow us to represent a rich set of control policies, including those that take into account differences in processing rate and in flexibility among servers, and demand rates and routing flexibility among jobs. For example, in a single-stage queueing system, jobs may assign priorities to servers based on their processing speed (e.g., always select the fastest available server). Alternatively, jobs may assign servers priorities based on their flexibility (e.g., always select the least flexible available server). Similarly, servers may associate priorities to jobs based on their demand rate or their routing flexibility. For instance, jobs are assigned priorities based on their arrival rate (e.g., always select the job with the highest arrival rate) or alternatively based on their flexibility (e.g., select the job with the fewest feasible number of servers). In the more general multi-stage queueing network, we can assign priorities to the jobs based on the expected amount of work remaining to be performed on the job. For example, a job that is in queue at the final stage of processing could be assigned greater priority than a job class that represents an intermediate stage in the system. In other situations, we could assign greater priority to a job class that represents the stage bottleneck in the system.

In this thesis, we are concerned with the analysis of systems with finite queue capacity. We can define queue capacity in one of two ways. We may specify a global bound b on the maximum number of jobs in the network, regardless of class, that can be allowed in the system. A job is admitted as long as the number of jobs in the network is less than or equal to b; conversely arriving jobs, regardless of their class, are balked when the number in system equals b. Alternatively, we can define a maximum number denoted by b_i for each job class, where $b_i \geq 1$ for i=1,..., n; jobs of class i, when seen either as new arrivals to the system or as inter-class transitions, are only admitted as long as the number of class i jobs already in the network is less than or equal to b_i.

3.1 The State Space

The state of the flexible queueing network can be described completely by specifying (i) the number of jobs in queue for each job class, (ii) the state of every server in the set M, and (iii) for a multi-stage network, if a server is busy, the identity of the job class being processed by the server at any instant in time. Since we do not model server failures, there are therefore $|T_j|+1$ possible states for server j. The state of the system can be described using a vector $\mathbf{N} \equiv (n_1, n_2, ..., n_{n+m})$, where n_i is the number of jobs of class i for $1 \leq i \leq n$ and n_i is the state of server i for $n+1 \leq i \leq n+m$. We denote the state space generated by such a representation as S_1. Although this state space representation could be used, it requires evaluating a large number of states even for small values of n and m. Therefore as a principle, we seek to minimize the number of distinct states that are used to describe the operation of the queueing network. The state representation and the subsequent description of model dynamics, shares similarities with that of Sheikhzadeh et al. (1998), who first demonstrated this approach for specific single-stage queueing systems, and that of Gurumurthi and Benjaafar (2004) who generalized this approach to depict arbitrarily defined single-stage queueing systems. Our state representation attempts to model general queueing networks with potentially multiple stages of processing. This requires marginally greater effort in the description of the state space, and hence for clarity of presentation, we represent the state space of the queuing network X using the state space of a queuing network $\overline{\mathrm{X}}$ that exhibits identical behavior and has the same operational characteristics as the original network, but that offers simplified representation and implementation.

The structural transformation from a given network X to its auxiliary network $\overline{\mathrm{X}}$ is obtained primarily by having, for every server R_j in X, $|T_j|$ servers in $\overline{\mathrm{X}}$ where the service of the l^{th} of these $|T_j|$ servers is restricted to the l^{th} job class in the set T_j with processing rate $\mu_{T_j\{l\},j}$. We use notation κ_j for this group of servers in $\overline{\mathrm{X}}$ to signify that the group was formed from server R_j in X; this additional notation is used later in the operational analysis. The k^{th} server in κ_j is associated with a singleton set $\overline{T}_{ij(k)} = \{T_{ij}(k)\}$; therefore in order to simplify the notation, we simply denote the job class associated with any server $\overline{R}_j$ as $\overline{T}_j$, and update a corresponding routing matrix $\overline{A}$ as $\overline{a}_{\overline{T}_j,j} = 1$ Once the servers have been defined thus, we can create the set $\overline{M} = \{\overline{R}_1, ..., \overline{R}_{\overline{m}}\}$, where $\overline{m} = \sum_{j=1}^{m} |T_j|$, and maintain the relations between servers in $\overline{\mathrm{X}}$ by defining a set K_r for each server r as follows: $\mathrm{K}_r \ni \overline{R}_k$ if $\{\overline{R}_k, \overline{R}_r\} \in \kappa_j$ for any $j \in M, k \in \overline{M}$. Similarly, for every job class P_i in X, we have a distinct job class $\overline{P}_i$ in $\overline{\mathrm{X}}$ with arrival rate λ_i, but with a set of allowable servers $\overline{Q}_i$ that consists of all servers $\overline{R}_r$, such that $\overline{P}_i = \overline{T}_r$ (in other terms $\overline{a}_{i,r} = 1$). The creation of set $\overline{P} = \{\overline{P}_1, ..., \overline{P}_n\}$ completes the transformation of network X into its auxiliary $\overline{\mathrm{X}}$.

Next we manipulate the ordering of the sets $\overline{M}$ and $\overline{P}$ so that every state variable refers to either: (*i*) the number in queue for a job class *i* (q_i), (*ii*) the state of a server *r* (s_r = 0, 1), or (*iii*) the sum of the two, $q_i + s_r$, for a pair job *i*-server *r*. Such an ordering can be achieved in the following fashion. The queueing network $\overline{\mathrm{X}}$ is viewed as a bipartite undirected graph, *G*(*V*, *E*), where *V* is the set of vertices and *E* is the set of edges. The vertices of the graph $\overline{\mathrm{X}}$ can be partitioned into two disjoint subsets, a set V_1 that corresponds to the set of job classes, and a set V_2 that corresponds to the set of servers. The edges of the graph connect the vertices in the set of job classes to the vertices in the set of servers, where an edge e_{ir} between job class i and server r exists only if $\bar{a}_{ir} = 1$. On the other hand, note that the graph is disconnected since the sub-graph $\bar{X}_i$ corresponding to each job class and its allowable servers is now disjoint from $\bar{X} \setminus \bar{X}_i$, where $\bar{X} \setminus \bar{X}_i$ is the set of all vertices and edges not in $\bar{X}_i$. We cannot however analyze the operation of $\bar{X}_i$ independently of $\bar{X} \setminus \bar{X}_i$, given the relation set κ that determines whether any two servers in $\bar{X}$ can be in service simultaneously. Before we present our state space representation scheme, we need the following two definitions and lemma.

Definition 1: *A subgraph of $G(V, E)$ in which every vertex has a degree of at most one (i.e., every vertex has at most one edge) is called a matching. The problem of finding such a sub-graph is also sometimes called matching.*

Definition 2: *A maximum matching (or a matching of maximum cardinality) of graph $G(V, E)$, is a matching $G_x(V, E_x)$ such that $|E_x| \geq |E_y|$ for any other matching $G_y(V, E_y)$, where $|E_x|$ and $|E_y|$ refer to the cardinality of the set of edges E_x and E_y respectively.*

Lemma 1: *Consider an undirected bipartite graph $G(V, E)$ whose vertices can be partitioned into two disjoint sets V_1 with n vertices and V_2 with $\bar{m}$ vertices. Then, there exists a graph $G_x(V, E_x)$ that has the following properties:*

1. *$G_x(V, E_x)$ is a sub-graph of $G(V, E)$,*
2. *$G_x(V, E_x)$ is a maximum matching of $G(V, E)$, and*
3. *$G_x(V, E_x)$ has e_x edges, where $1 \leq e_x \leq \min(\bar{m}, n)$ and $\bar{m} + n - 2e_x$ unmatched vertices.*
4. *There is at most one vertex in any group defined by relation κ_j (j= 1,..., $\bar{m}$) that has degree 1.*

A proof of *Lemma* 1 is left as a simple exercise to the reader. Readily available algorithms for the implied constrained bipartite matching problem include the maximum flow algorithms discussed in Ahuja et al. (1993). The constrained maximum matching yields a sub-graph with e_x jobs and e_x servers with each job connected to one server by an edge. Without loss of generality, we rename this set of jobs and servers so that a job that has been renamed $\bar{P}_i$ is connected to a server that has been renamed $\bar{R}_i$. Then we associate with this job-server pair a state variable n_i, where $n_i = s_i + q_i$. The maximum matching also yields l unmatched vertices ($l = \bar{m} + n - 2e_x$). If $n > e_x$, we rename these job classes $\overline{P}_{e_x+1},...,\overline{P}_n$ and associate with each a state variable n_i where n_i = q_i. Similarly, if $\overline{m} > e_x$, the unmatched servers are renamed $\overline{P}_{e_x+1},...,\overline{P}_{\overline{m}}$, and we associate with each a state variable $n_i = s_i$. The set of allowable servers $\overline{Q}_i$ for each class i of jobs, and the network flow matrix $\boldsymbol{\Phi}$, are also updated to reflect the renaming of job classes and servers. This process results in a state vector **N** = (n_1, n_1, ..., n_q), where q = $l = \overline{m} + n - e_x$ and

$$n_i = \begin{cases} q_i + s_i & \text{if } 1 \le i \le e_x, \\ q_i & \text{if } e_x \le i \le n \text{ and } n > e_x, \text{ and} \\ s_i & \text{if } n \le i \le \overline{m} \text{ and } \overline{m} > e_x \end{cases} \tag{2}$$

We denote the resulting state space of $\overline{\mathrm{X}}$ as S_2. The process of generating the state vector **N** is illustrated in Figure 2 for an example system with 3 job classes and 3 servers. Our representation scheme reduces the number of state variables from $(\overline{m} + n)$ to $(\overline{m} + n - e_x)$ and therefore leads to a reduction in the number of states that are needed to describe $\overline{\mathrm{X}}$. At this time, we do not know of any other representation that allows us to describe the operation of $\overline{\mathrm{X}}$ with a fewer number of distinct states.

3.2 Performance Evaluation

In this section, we develop models for the performance evaluation of flexible queueing networks. Our first task is to determine the probability of occurrence of each system state for the different control policies under consideration. From these probabilities, we show how to obtain various performance measures of interest. We model our system as a continuous time Markov chain (CTMC) with state vectors **N** = (n_1, n_2, ..., n_q), and n_i is as defined in section 1.1. Our Markov chain experiences system transitions from its current state through either a single job arrival or a single job departure. The unique limiting probabilities of system states can be obtained from the balance equations of the Markov chain by equating the rate at which the system enters a state with the rate at which it leaves it (Ross, 1995). This relationship results in a set of linear equations that can be solved using a general-purpose linear equation solver.

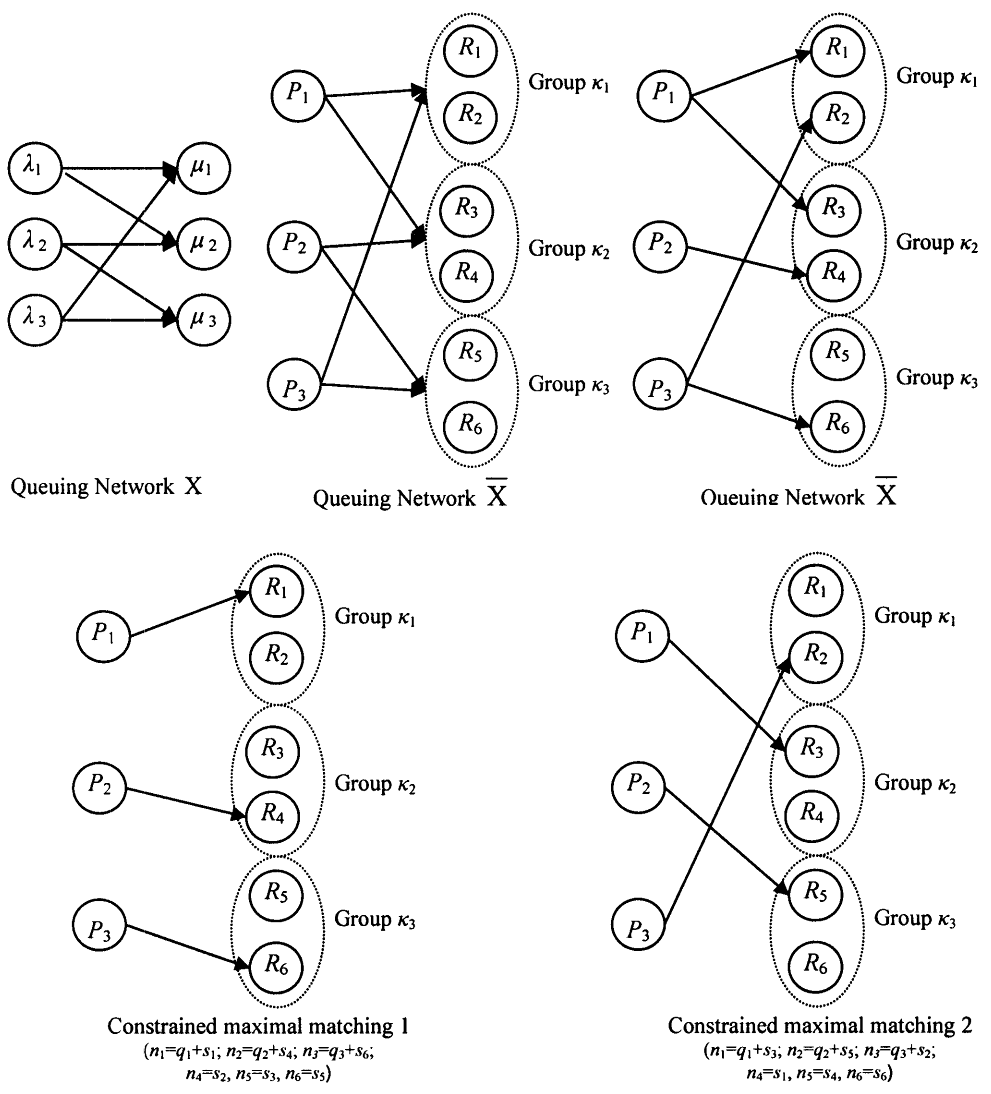

λ_1
λ_2
λ_3
μ_1
μ_2
μ_3
Queuing Network X
P_1
P_2
P_3
R_1
R_2
R_3
R_4
R_5
R_6
Group κ_1
Group κ_2
Group κ_3
Queuing Network $\overline{\text{X}}$
Queuing Network $\overline{\text{X}}$
Constrained maximal matching 1
$(n_1=q_1+s_1;\ n_2=q_2+s_4;\ n_3=q_3+s_6;\ n_4=s_2,\ n_5=s_3,\ n_6=s_5)$
Constrained maximal matching 2
$(n_1=q_1+s_3;\ n_2=q_2+s_5;\ n_3=q_3+s_2;\ n_4=s_1,\ n_5=s_4,\ n_6=s_6)$

Figure 10 – An example to illustrate constrained maximal matching

Extending the approach in Sheikhzadeh et al (1998), we define for each state vector $\mathbf{N} = (n_1, n_2, ..., n_q)$ three sets of states, type *a*, type *d*, and type *a-d* depending respectively on whether a new service request (for e.g. an arrival of a job into a queue), a service completion (e.g. a departure of a job from a class), or a combination of a service completion and a service request causes the system to move to state **N**. We denote these sets of states as N^a , N^d, and $N^{a\text{-}d}$ respectively. Elements of N^a are state vectors $\mathbf{N_i^a}$ such that $n_i^a = n_i - 1$ and all other state variables having the same value as in **N**, while elements of N^d are state vectors $\mathbf{N_i^d}$ such that $n_i^d = n_i+1$ with all other state variables having the same value as in **N**. Elements of $N^{a\text{-}d}$ however, are state vectors $\mathbf{N_{ij}^{a\text{-}d}}$ such that $n_i^a = n_i - 1$, exactly one other state variable has $n_j^a = n_j+1$ with all other state variables having the same value as in **N**; except when $i=j$, in which case $\mathbf{N_{ij}^a}$ potentially includes state **N** itself. Thus, states $\mathbf{N_i^a}$ ($\mathbf{N_i^d}$) witness new arrivals to (net departures from) the queueing network. On the other hand, the states $\mathbf{N_{ij}^{a\text{-}d}}$ witness a transfer between job classes, or even potentially a one-step recurrence to the same class.

We define $\delta(x)$ as a function that returns 1 if $x \geq 1$, and 0 otherwise. We also define:

1. $$v_i = \begin{cases} 1, & \text{if } 1 \leq i \leq n, \text{ and} \\ 0, & \text{otherwise,} \end{cases},$$

2. $$\omega_i = \begin{cases} 1, & \text{if } (1 \leq i \leq n) \text{ or } (n < i \leq \overline{m} \text{ and } \overline{m} > e_x) \\ 0, & \text{otherwise, and} \end{cases}$$

3. $$u_i = \begin{cases} b_i - \left(\sum_{t=1}^{n} \delta(\overline{a}_{ti}) \left(n_t + \sum_{r=1}^{q} \delta(\overline{a}_{tr}) n_r \right) \right), & \text{if } w_i = 1 \text{ and } b_i < b, \\ b_i - \left(n_t + \sum_{r=1}^{q} \delta(\overline{a}_{tr}) n_r \right), & \text{if } w_i = 0 \text{ and } b_i < b, \\ b_i - \left(\sum_{r=1}^{q} n_r \right), & \text{if } w_i = 0 \text{ and } b_i = b, \\ 0, & \text{otherwise.} \end{cases}$$

The variable v_i is used to indicate if a state variable n_i includes the queue size of a job class i. Similarly, the variable ω_i is used to indicate if a state variable n_i includes the state of a server i. Finally, for each job class a variable u_i is used to determine the slack for the state variable n_i, and represents the remaining WIP capacity in the system for the associated job class. Recall that we had defined the parameter b_i in §3.1, so the condition $b_i < b$ can be used to evaluate whether we are using global network bounds or the alternative class-wise bounds on WIP.

Whenever the system is in state **N**, it is straightforward to show that it leaves this state as a result of a service completion with rate $\sum_{i=1}^{q} \omega_i \delta(n_i) \mu_i$ and as a result of an external arrival with rate $\sum_{i=1}^{q} \delta(u_i) v_i \lambda_i$. If we use r_i^a, r_i^d, and $r_{ij}^{a\text{-}d}$ to denote the rates at which the system enters state **N** from $\mathbf{N_i^a}$, $\mathbf{N_i^d}$, and $\mathbf{N_{ij}^{a\text{-}d}}$ respectively; and if we use notation

$p(\mathbf{N})$, $p(\mathbf{N_i^a})$, $p(\mathbf{N_i^d})$ and $p(\mathbf{N_{ij}^{a\text{-}d}})$ to denote the steady state probabilities of those states, we can then write the Markov chain balance equation as follows:

$$\left(\sum_{i=1}^{q} \omega_i \delta(n_i)\mu_i + \sum_{i=1}^{q} \delta(u_i)v_i\lambda_i\right)p(\mathbf{N}) = \sum_{\mathbf{N_i^a}\in N^a} r_i^{\,a} p(\mathbf{N_i^a}) + \sum_{\mathbf{N_i^d}\in N^d} r_i^{\,d} p(\mathbf{N_i^d}) + \sum_{\mathbf{N_{ij}^{a\text{-}d}}\in N^{a-d}} r_{ij}^{\,a-d} p(\mathbf{N_{ij}^{a\text{-}d}}), \forall\, \mathbf{N} \in S_2. \tag{3}$$

The set of linear equations in (3) along with the normalizing equation $\sum_{S_2} p(\mathbf{N}) = 1$ form a set of $|S_2|$ simultaneous equations, which can be solved to determine the unique steady state probabilities, $p(\mathbf{N}), \mathbf{N} \in S_2$. The uniqueness of the steady state probabilities follows from the (reasonable) assumption that the matrix $\boldsymbol{\Phi}$ is well-defined. When the matrix $\boldsymbol{\Phi}$ is well-defined, then the CTMC defined on state space S_2 is (*i*) irreducible and positive recurrent, since any two states in S_2 can communicate with each other through a sequence of one-step transitions (*ii*) aperiodic, since a CTMC cannot be periodic, (*iii*) has a finite number of states (by construction), and (*iv*) ergodic, since property (*iii*) assures us of the existence of an unique vector of steady state probabilities. However, in order to solve for $p(\mathbf{N})$, we need to first define the sets of entering states $\{\mathbf{N_i^a}, \mathbf{N_i^d}, \mathbf{N_{ij}^{a\text{-}d}}\}$ and subsequently determine the associated rates $\{r_i^a, r_i^d, r_{ij}^{a-d}\}$. There are two types of constraints that define whether a state can be included in one of the sets N^a, N^d and N^{a-d}. The first constraint deals with feasibility. Depending on the routing matrix, the network flow matrix $\boldsymbol{\Phi}$, and the control policies, there are certain states that can never occur. The second restriction stems from the requirement that it should be possible to go from a member of N^a, N^d or N^{a-d} to $\mathbf{N}$ by a one-step transition as a result of a single service request, a single service completion, or a combination of both.

3.2.1 The sets $\mathrm{N_i^a}$ and $\mathrm{N_i^d}$

The system can move into state $\mathbf{N}$ from $\mathbf{N_i^a}$, only if $n_i^a = n_i - 1$ and all other state variables have exactly the same values as in $\mathbf{N}$. That is,

$$n_i^a = n_i - 1 \text{ and } n_k^a = n_k \forall k \neq i. \tag{4}$$

The state $\mathbf{N_i^a}$ exists only when one of the following mutually exclusive conditions holds:

$$n_i^a \geq \left(1 - \delta(\omega_i \sum_{k \in \mathrm{K}_i, k \neq i} n_k)\right), u_i^a \geq 1 \text{ and } \delta(\sum_{k \in K_l} n_k^a) = 1, \forall l \in \overline{Q}_i, \nu_i = 1, \lambda_i > 0; \tag{5}$$

$$\delta(\sum_{k \in \mathrm{K}_i} n_k^a) = 0, \text{and } n_{\overline{T}_k}^a \leq \omega_{\overline{T}_k}\left(1 - \delta(\sum_{l \in \mathrm{K}_{\overline{T}_k}, l \neq \overline{T}_k} n_l^a)\right), \forall k \in \mathrm{K}_i \in \omega_i = 1; \lambda_i > 0. \tag{6}$$

Condition (5) follows from the fact that we do not allow a queue to form while a feasible server is idle. It states that for the queue represented by n_i^a to increase by 1, any server r that is directly capable of processing jobs of class i must be busy, or at the least must be out of contention for service, given that the network transformation from X to $\overline{\mathrm{X}}$ disqualifies server r from service if any one of its related servers from set K_r is busy. Condition (6) applies to a state variable n_i^a (and state $\mathbf{N_i^a}$) that represents an idle server in position to accept a service request in the form of a new arrival to the network to move into state **N**. Such a state $\mathbf{N_i^a}$ is possible only if the server represented by variable n_i is idle and is still in contention for service, which event in turn occurs only if there are no jobs in the queue that can be processed on any server related to i (including server i itself) and if all of these related servers are idle. Note that the job class i must allow new arrivals from outside the network (i.e. $\lambda_i > 0$) for conditions (5) and (6) to hold, since otherwise this job class represents an intermediate process stage, and therefore by definition state $\mathbf{N_i^a}$ cannot exist.

Similarly, the network can move into state **N** from a state $\mathbf{N_i^d}$ only if $n_i^d = n_i + 1$ and all other state variables maintain the same values as in **N**. That is:

$$n_i^d = n_i + 1 \textit{ and } n_k^d = n_k, \forall k \neq i. \tag{7}$$

The state $\mathbf{N_i^d}$ exists only when one of the following mutually exclusive conditions holds:

$$n_i^d = 1, \delta(\sum_{k \in K_i, k \neq i} n_k{}^d) = 0, n_{\overline{T}_k}^d \leq \omega_{\overline{T}_k}\left(1 - \delta(\sum_{l \in \mathrm{K}_{\overline{T}_k}, l \neq \overline{T}_k} n_l^a)\right), \forall k \in \mathrm{K}_i, \omega_i = 1; \text{and} \sum_{j=1}^{n} t_{\overline{T}_i, j} < 1 \tag{8}$$

$$n_i^d \geq 1 + \omega_i\left(1 - \delta(\sum_{k \in \mathrm{K}_i, k \neq i} n_k{}^d)\right), \text{ and } \delta(\sum_{k \in K_l} n_k{}^d) = 1, \forall l \in \overline{Q}_i, \nu_i = 1; \text{and} \sum_{j=1}^{n} t_{i,j} < 1 \tag{9}$$

Condition (8) follows from the fact that a departure would cause n_i^d to decrease from 1 to 0 if there are no other jobs present in queue that can be serviced at server *i*. Note however, that a departure from a state variable n_i^d is possible, only if the server associated with this variable were the only busy server in set K_i, and this is exactly what constraint $\delta(\sum_{k \in K_i, k \neq i} n_k) = 0$ implies. Condition (9), on the other hand, concerns a state variable n_i^d (and state $\mathbf{N_i^d}$) representing a queue that could experience a unit depletion through a service completion at one of the servers associated with that queue; this service completion results in a job departing from the network as opposed to a job transition to another class. Such a state $\mathbf{N_i^d}$ is therefore possible only if the queue represented by variable n_i is non-empty, which event in turn occurs only if there are no servers in the network that are idle or if even idle, in contention (given the constraint that only one server in any set K_r be busy at any instant) to service a job in the queue of class *i*. Note again that in order for $\mathbf{N_i^d}$ to exist, there must be a positive probability of the departure of the job of class *i* from the network as a result of this one-step transition, and this is what is dictated by the constraint $\sum_{j=1}^{n} t_{i,j} < 1$.

3.2.2 The set $N_{ij}^{a\text{-}d}$

The system can move into state **N** from $\mathbf{N_{ij}^{a\text{-}d}}$, only if $n_i^a = n_i - 1$, $n_j^a = n_j + 1$ and all other state variables have exactly the same values as in **N**, except when $i=j$ when the system experiences a self-transition. That is,

$$\begin{aligned} &n_i^{a-d} = n_i - 1,\ n_j^{a-d} = n_j + 1,\ \textit{and}\ n_k^{a-d} = n_k \forall k \neq i, k \neq j; i \neq j, \text{ or} \\ &n_k^{a-d} = n_k\ \forall k \text{ if } i = j. \end{aligned} \tag{10}$$

Considering the $i \neq j$ case first, the state $\mathbf{N_{ij}^{a\text{-}d}}$ exists only when one of the following mutually exclusive conditions holds:

$$\begin{aligned}
&n_i^{a-d} \geq \left(1-\delta(\omega_i \sum_{k\in K_i, k\neq i} n_k^{a-d})\right),\ \delta(\sum_{k\in K_l} n_k^{a-d}) = \omega_l,\ \forall l \in \overline{Q}_i,\ v_i = 1;\\
&n_j^{a-d} = 1, \delta(\sum_{k\in K_j, k\neq j} n_k^{a-d}) = 0, n_{\overline{T}_k}^{a-d} \leq \omega_{\overline{T}_k}\left(1-\delta(\sum_{l\in K_{\overline{T}_k}, l\neq \overline{T}_k} n_l^{a-d})\right), \forall k\in K_j,\ \omega_j = 1; \text{and}\\
&(t_{\overline{T}_j,i} > 0, \text{and}\, u_i^{a-d}\geq 1) \text{ or } (u_h^{a-d} = 0, t_{\overline{T}_j,h} > 0, \text{and}\, \overline{T}_j = i; h\neq i; h\in\{1,2,\ldots,n\})
\end{aligned} \tag{11}$$

$$\begin{aligned}
&n_i^{a-d} \geq \left(1-\delta(\omega_i \sum_{k\in K_i, k\neq i} n_k^{a-d})\right),\ u_i^{a-d}\geq 1,\ \delta(\sum_{k\in K_l} n_k^{a-d}) = \omega_l,\ \forall l \in \overline{Q}_i,\ v_i = 1;\\
&n_j^{a-d} \geq 1+\omega_j\left(1-\delta(\sum_{k\in K_i, k\neq j} n_k^{a-d})\right),\ \text{and}\ \delta(\sum_{k\in K_r} n_k^{a-d}) = 1,\ \forall r \in \overline{Q}_j,\ v_j = 1; \text{and}\\
&(t_{\overline{T}_k,i} > 0; k\in K_r; r\in \overline{Q}_j; u_i^{a-d}\geq 1) \text{or}\\
&(u_h^{a-d} = 0, t_{T_k,h} > 0, \text{and}\, \overline{T}_k = i; k\in K_r; r\in\overline{Q}_j; h\neq i; h\in\{1,2,\ldots,n\})
\end{aligned} \tag{12}$$

$$\begin{aligned}
&\delta(\sum_{k\in K_i} n_k^{a-d}) = 0, \text{and}\, n_{\overline{T}_k}^{a-d} \leq \omega_{\overline{T}_k}\left(1-\delta(\sum_{l\in K_{\overline{T}_k}, l\neq \overline{T}_k} n_l^{a-d})\right), \forall k\in K_i,\ \omega_i = 1;\\
&n_j^{a-d} = 1, \delta(\sum_{k\in K_j, k\neq j} n_k^{a-d}) = 0, n_{\overline{T}_k}^{a-d} \leq \omega_{\overline{T}_k}\left(1-\delta(\sum_{l\in K_{\overline{T}_k}, l\neq \overline{T}_k} n_l^{a-d})\right), \forall k\in K_j,\ \omega_j = 1;\\
&(t_{\overline{T}_j,\overline{T}_i} > 0, \text{and}\, u_i^{a-d}\geq 1) \text{ or}\\
&(u_h^{a-d} = 0, t_{\overline{T}_j,h} > 0, \text{and}\, \overline{T}_j = \overline{T}_i; h\neq \overline{T}_i; h\in\{1,2,\ldots,n\})
\end{aligned} \tag{13}$$

$$\begin{aligned}
&\delta(\sum_{k\in K_i} n_k^{a-d}) = 0,\ u_i^{a-d}\geq 1, \text{and}\, n_{\overline{T}_k}^{a-d} \leq \omega_{\overline{T}_k}\left(1-\delta(\sum_{l\in K_{\overline{T}_k}, l\neq \overline{T}_k} n_l^{a-d})\right), \forall k\in K_i,\ \omega_i = 1;\\
&n_j^{a-d} \geq 1+\omega_j\left(1-\delta(\sum_{k\in K_i, k\neq j} n_k^{a-d})\right),\ \text{and}\ \delta(\sum_{k\in K_r} n_k^{a-d}) = 1,\ \forall r \in \overline{Q}_j,\ v_j = 1;\\
&(t_{\overline{T}_k,\overline{T}_i} > 0; k\in K_r; r\in \overline{Q}_j; u_i^{a-d}\geq 1) \text{or}\\
&(u_h^{a-d} = 0, t_{T_k,h} > 0, \text{and}\, \overline{T}_k = \overline{T}_i; k\in K_r; r\in\overline{Q}_j; h\neq \overline{T}_i; h\in\{1,2,\ldots,n\})
\end{aligned} \tag{14}$$

Conditions (11)-(14) have been derived by enforcing as a pair the essence of conditions {5, 8}, {5,9}, {6,8} and {6,9} respectively, with the only additional constraint being that in order for $\mathbf{N_{ij}^{a\text{-}d}}$ to exist, there must be a positive probability of the departure of the job of class j to class i, as defined by the network flow matrix $\boldsymbol{\Phi}$. Note also that in all of these conditions we account for the fact that certain inter-class transitions may not be allowed because of the bounds specified on the WIP for each job class. Condition (11) follows from the fact that in order for a net transfer of one job to occur from the busy server represented by n_j^{a-d} to the queue represented by n_i^{a-d}, any server r that is directly capable of processing jobs of class i must be busy, or at the least must be out of contention for service. At the same time there cannot be a job of any other class present in queue that can be serviced at server j or any of its related servers in set K_j. Note again that a decrement in state variable n_j^{a-d} is possible, only if the server associated with this variable were the only busy one in set K_j, and this is exactly what constraint $\delta(\sum_{k \in K_j, k \neq j} n_k^{a-d}) = 0$ implies. Condition (12) states that in order for a net transfer of one job to occur from a non-empty queue represented by n_j^{a-d} (one of whose associated servers completes service at the moment of the transfer) to the queue represented by n_i^{a-d}, any server r that is directly capable of processing jobs of class i must be busy, or at the least must be out of contention for service, while at the same time there are no servers in the network that are idle or if idle, in contention to service a job in the queue of class j. Condition (13) concerns the idle server that is represented by variable n_i^{a-d} and that accepts a service request generated by the service completion at one of the servers represented by n_j^{a-d}. Such a job transfer is possible only if the server represented by variable n_i^{a-d} is idle and is still in contention for service which event in turn occurs only if there are no jobs in the queue that can be processed on server i or any of its related servers; further a service completion at server j can decrement n_j^{a-d} only if there no other jobs present in queue that can be serviced at server j or its related servers, and if this server were the only busy one in set K_i. Finally condition (14) states that in order for a net transfer of one job to occur from a non-empty queue represented by n_j^{a-d} (one of whose associated servers completes service at the event of the transfer) to the idle server represented by n_i^{a-d}, there can be no job in the queue that can be serviced by server i ; all of the related servers of server i must also be idle. Further there can be no server in the network that is idle (or if idle, even in contention) to service a job in the queue of class j.

Considering the $i = j$ case next, a transition from state $\mathbf{N}$ to itself as a result of the combination of a service completion by the server represented by n_i and a service request at the queue or server represented by n_i can occur, if the following conditions hold:

$$n_i = 1, \delta(\sum_{k \in K_r, k \neq i} n_k) = 0,\ n_{\overline{T}_k} \leq \omega_{\overline{T}_k}\left(1 - \delta(\sum_{l \in \mathrm{K}_{\overline{T}_k}, l \neq \overline{T}_k} n_l)\right), \forall k \in \mathrm{K}_i,\ \omega_i = 1; \text{and } t_{i,\overline{T}_i} > 0 \text{ or} \tag{15}$$
$$(u_h^{a-d} = 0, t_{\overline{T}_i,h} > 0, \text{and}\,; h \neq \overline{T}_i; h \in \{1,2,...,n\})$$

$$n_i^d \geq 1 + \omega_i\left(1 - \delta(\sum_{k \in \mathrm{K}_i, k \neq i} n_k)\right), \text{ and } \delta(\sum_{k \in K_r} n_k) = 1, \forall r \in \overline{Q}_i, \omega_i = v_i = 1; t_{r,i} > 0 \text{ or} \tag{16}$$
$$(u_h^{a-d} = 0, t_{T_k,h} > 0, \text{and } \overline{T}_k = i; k \in K_r; r \in \overline{Q}_j; h \neq i; h \in \{1,2,...,n\})$$

In stating these conditions we also account for the fact that certain inter-class transitions may not be allowed because of the bounds specified on the WIP for each job class.

3.2.3 Transition rates r_i^a, r_i^d, and $r_i^{a\text{-}d}$.

In this section, we show how the transition rates for the control policies we consider can be determined. Recall that we define control policies in terms of a server and job selection rule combination.

The SP-LQF Policy

Under the SP-LQF policy, servers are selected based on a strict priority scheme and jobs are selected from the class with the longest queue. When condition (5) holds, a transition from $\mathbf{N_i^a}$ to $\mathbf{N}$ clearly occurs with rate:

$$r_i^a = \lambda_i. \tag{17}$$

On the other hand, when condition (6) holds, the transition rate depends on the routing priorities. First note that for n_i^a to increase from zero to one, we need an arrival from a job class $\overline{T}_i$. Although necessary, this condition is not sufficient since an arrival of a job of class $\overline{T}_i$ may not be routed to server i unless server i has the highest priority among those available to process job $\overline{T}_i$. This means that the following condition

$$\alpha_{\overline{T}_i,i} < \alpha_{\overline{T}_i,j} \ \forall j \in \overline{Q}_{\overline{T}_i}; \text{s.t. } \delta(\sum_{k \in K_j} n_k^a) = 0$$

must be satisfied. Since the arrival rate of jobs of class $\overline{T}_i$ is $\lambda_{\overline{T}_i}$, the transition rate r_i^a is given by:

$$r_i^a = \lambda_{\overline{T}_i} \gamma(\alpha_{\overline{T}_i,i}),$$

where

$$\gamma(\alpha_{\overline{T}_i,i}) = \begin{cases} 1, \text{if} \alpha_{\overline{T}_i,i}\left(1 - \delta(\sum_{k \in K_i} n_k^a)\right) \le \alpha_{\overline{T}_i,t}, \forall t \in \overline{Q}_{\overline{T}_i}; \text{and} \\ 0, \text{otherwise.} \end{cases} \tag{18}$$

Putting it all together, we have:

$$r_i^a = \begin{cases} \lambda_{\overline{T}_i}, \text{if condition (5) holds;} \\ \lambda_{\overline{T}_i} \gamma(\alpha_{\overline{T}_i,i}), \text{if condition (6) holds.} \end{cases}$$

Similarly, we can derive the transition rates $\mathbf{N_i^d}$ to $\mathbf{N}$. If condition (8) holds, then we clearly have $r_i^d = \mu_i \left(1 - \sum_{r=1}^{n} t_{i,r}\right)$. When condition (9) holds, the transition rate depends on the relative size of the queues. There can be a transition from $\mathbf{N_i^d}$ to $\mathbf{N}$ if the queue for job class i has one of the longest queues for any of the servers in the set $\overline{Q}_i$. In other words, for a server r in the set $\overline{Q}_i$ to select queue i, queue i must be one of the longest queues in the set $\{\overline{T}_j : j \in \mathrm{K}_r\}$, the set of feasible job classes for server r or its related servers K_r. We denote by B_r the number of jobs that have the longest queue in the set $\{\overline{T}_j : j \in \mathrm{K}_r\}$. When $B_r > 1$, queue i is selected by server r with probability $1/B_r$. Thus, the transition rate can be written as:

$$r_i^d = \begin{cases} \mu_i \left(1 - \sum_{r=1}^{n} t_{i,r}\right), \text{if condition (8) holds;} \\ \sum_{r \in Q_i} \frac{\mu_r \left(1 - \sum_{z=1}^{n} t_{r,z}\right) \varepsilon_r(i)}{B_r}, \text{if condition (9) holds.} \end{cases} \tag{19}$$

where B_r is the number of job classes that have the longest queue in the set $\{\overline{T}_j : j \in K_r\}$; and $\varepsilon_r(i) = 1$ if the queue of job class i is one of the B_r longest queues in the set $\{\overline{T}_j : j \in K_r\}$, and 0 otherwise.

Finally, we can derive the transition rates $\mathbf{N_i^{a\text{-}d}}$ to $\mathbf{N}$. If condition (11) holds, then we clearly have $r_{ij}^{a-d} = \mu_j t_{j,i}$. If condition (12) holds, then the transition rate depends on the relative size of the queues. There can be a transition from $\mathbf{N_i^{a\text{-}d}}$ to $\mathbf{N}$ if the queue for job class j has one of the longest queues for any of the servers in the set $\overline{Q}_j$. In other words, for a server r in the set $\overline{Q}_j$ to select queue j, queue j must be one of the longest queues in the set $\{\overline{T}_f : f \in K_r\}$, the set of feasible job classes for server r or its related servers K_r. We denote by B_r the number of jobs that have the longest queue in the set $\{\overline{T}_f : f \in K_r\}$. When $B_r > 1$, queue i is selected by server r with probability $1/B_r$. Hence the transition rate $r_{ij}^{a-d} = \sum_{r \in Q_j} \frac{\mu_r t_{r,i} \varepsilon_r(j)}{B_r}$, where B_r is the number of job classes that have the longest queue in the set $\{\overline{T}_f : f \in K_r\}$; and $\varepsilon_r(j) = 1$ if the queue of job class i is one of the B_r longest queues in the set $\{\overline{T}_f : f \in K_r\}$, and 0 otherwise. On the other hand, when condition (13) holds, the transition rate depends on the routing priorities. First note that for n_i^{a-d} to increase from zero to one, we need a completion of from job class $\overline{T}_j$ at server j that is transformed into a job of class $\overline{T}_i$. Although necessary, this condition is not sufficient since a transfer of a job to class $\overline{T}_i$ may not be routed to server i unless server i has the highest priority among those available to process job $\overline{T}_i$. This means that the following condition

$$\alpha_{\overline{T}_i,i} < \alpha_{\overline{T}_i,t} \ \forall t \in \overline{Q}_{\overline{T}_i}; \text{s.t. } \delta\Big(\sum_{k \in K_t} n_k^{a-d}\Big) = 0$$

must be satisfied. Since the transfer rate of jobs of class $\overline{T}_j$ to class $\overline{T}_i$ is $\mu_j t_{j,i}$, the transition rate r_{ij}^{a-d} is given by:

$$r_i^{a-d} = \mu_j t_{j,i} \gamma(\alpha_{\overline{T}_i,i}),$$

where

$$\gamma(\alpha_{\overline{T}_i,i}) = \begin{cases} 1, \text{if} \alpha_{\overline{T}_i,i}\left(1 - \delta(\sum_{k\in K_t} n_k^a)\right) \le \alpha_{\overline{T}_i,t}, \forall t \in \overline{Q}_{\overline{T}_i}; \text{and} \\ 0, \text{otherwise.} \end{cases}$$

When condition (14) holds, then the transition rate depends on the relative size of the queues as well as the routing priorities in the network. There can be a transition from $\mathbf{N}_i^{a-d}$ to $\mathbf{N}$ if the queue for job class j has one of the longest queues for any of the servers in the set $\overline{Q}_j$. In other words, for a server r in the set $\overline{Q}_j$ to select queue j, queue j must be one of the longest queues in the set $\{\overline{T}_f : f \in \mathrm{K}_r\}$, the set of feasible job classes for server r or its related servers K_r. We again denote by B_r the number of jobs that have the longest queue in the set $\{\overline{T}_f : f \in \mathrm{K}_r\}$ and set $\varepsilon_r(j) = 1$ if the queue of job class i is one of the B_r longest queues in the set $\{\overline{T}_f : f \in \mathrm{K}_r\}$, and 0 otherwise. Hence we conclude that the transfer to class $\overline{T}_i$ occurs at the rate $\sum_{r\in\overline{Q}_j} \frac{\mu_r t_{r,i} \varepsilon_r(j)}{B_r}$. However, the transfer of a job at this rate to class $\overline{T}_i$ may not result in the job being routed to server i unless server i has the highest priority among those available to process job $\overline{T}_i$. This means that the following condition

$$\alpha_{\overline{T}_i,i} < \alpha_{\overline{T}_i,t} \ \forall t \in \overline{Q}_{\overline{T}_i}; \text{s.t. } \delta(\sum_{k\in K_t} n_k^{a-d}) = 0$$

must be satisfied. Hence the transition rate r_{ij}^{a-d} for condition (14) is given by:

$$r_i^{a-d} = \gamma(\alpha_{\overline{T}_i,i}) \sum_{r\in\overline{Q}_j} \frac{\mu_r t_{r,i} \varepsilon_r(j)}{B_r},$$

where

$$\gamma(\alpha_{\overline{T}_i,i}) = \begin{cases} 1, \text{if} \alpha_{\overline{T}_i,i}\left(1 - \delta(\sum_{k\in K_t} n_k^a)\right) \le \alpha_{\overline{T}_i,t}, \forall t \in \overline{Q}_{\overline{T}_i}; \text{and} \\ 0, \text{otherwise.} \end{cases} \quad (20)$$

Thus, the transition rate can be written as:

$$r_{ij}^{a-d} = \begin{cases} = \mu_j t_{j,i}, \text{if condition (11) holds;} \\ = \sum_{r \in Q_j} \frac{\mu_r t_{r,i} \varepsilon_r(j)}{B_r}, \text{if condition (12) holds;} \\ = \mu_j t_{j,i} \gamma(\alpha_{j,i}), \text{if condition (13) holds;} \\ = \gamma(\alpha_{\overline{T}_i,i}) \sum_{r \in Q_j} \frac{\mu_r t_{r,j} \varepsilon_r(j)}{B_r}, \text{if condition (14) holds.} \end{cases} \tag{21}$$

We still have to determine the rates at which the network experiences a self-transition, when in state **N**, as a result of a combination of a service completion and a service request. When condition (15) holds in the case of a valid return of a job to the same class, the rate at which a service completion occurs is simply μ_i, and therefore the rate at which the job is transferred to class $\overline{T}_i$ is $\mu_i t_{i,\overline{T}_i}$. However for n_i to increase again from zero to one, we need this job that has returned to class $\overline{T}_i$ to be routed to server *i*, so as to effect a self-transition; this event occurs when
server *i* has the highest priority among those available to process job $\overline{T}_i$. This means that the condition $\alpha_{\overline{T}_i,i} < \alpha_{\overline{T}_i,t}\ \forall t \in \overline{Q}_{\overline{T}_i}$; s.t. $\delta(\sum_{k \in K_t} n_k^{a-d}) = 0$ must be satisfied. Since the transfer rate of jobs of class $\overline{T}_j$ to class $\overline{T}_i$ is $\mu_j t_{j,i}$, the transition rate r_{ij}^{a-d} is given by:

$$r_i = \mu_i t_{i,\overline{T}_i} \gamma(\alpha_{\overline{T}_i,i}), \tag{22}$$

where $\gamma(\alpha_{\overline{T}_i,i})$ is given by Equation (20). When we further account for inter-class transitions that are invalidated as a result of WIP capacity bounds, we add the transition rates derived for conditions (11) and (13) for the same control policies to compute the effective transition rate when condition (15) applies.

When condition (16) holds in the case of a valid return of a job to the same class, the transition rate depends on the relative size of the queues. There can be a transition from $\mathbf{N}_i^{a-d}$ to **N** if the queue for job class *j* has one of the longest queues for any of the servers in the set $\overline{Q}_j$. In other words, for a server *r* in the set $\overline{Q}_j$ to select queue *j*, queue *j* must be one of the longest queues in the set $\{\overline{T}_f : f \in \mathrm{K}_r\}$, the set of feasible job classes for server r or its related servers K_r. We again denote by B_r the number of jobs that have the longest queue in the set $\{\overline{T}_f : f \in \mathrm{K}_r\}$ and set $\varepsilon_r(j) = 1$ if the queue of job class i is one of the B_r longest queues in the set $\{\overline{T}_f : f \in \mathrm{K}_r\}$, and 0 otherwise. Hence we conclude that the transfer to class $\overline{T}_i$, and hence the self-transition through a combination of a service completion and service request occurs at the rate $r_i = \sum_{r \in \overline{Q}_j} \frac{\mu_r t_{r,i} \varepsilon_r(j)}{B_r}$. When we further account for inter-class transitions that are invalidated as a result of WIP capacity bounds, we add the transition rates derived for conditions (12) and (14) for the same control policies to compute the effective transition rate for when condition (16) applies.

The RR-LQF Policy

The sets of entering states are the same for this queue selection policy as in the SP-LQF control policy. Only the transition rates differ. When condition (5) holds, a transition from $\mathbf{N_i^a}$ to $\mathbf{N}$ clearly occurs with rate:

$$r_i^a = \lambda_i. \tag{23}$$

On the other hand, when condition (6) holds, the transition rate depends on the routing priorities. First note that for n_i^a to increase from zero to one, we need an arrival from a job class $\overline{T}_i$. Although necessary, this condition is not sufficient since an arrival of a job of class $\overline{T}_i$ may be routed with equal probability to any of the servers that are idle and are in contention to service the job. The rate r_i^a is therefore given by:

$$r_i^a = \frac{\lambda_{\overline{T}_i}}{\sum_{t \in \overline{Q}_{\overline{T}_i}} \left(1 - \delta\left(\sum_{k \in \mathrm{K}_t} n_k^a\right) \right)} \tag{24}$$

Putting it all together, we have:

$$r_i^a = \begin{cases} \lambda_{\overline{T}_i}, \text{if condition (5) holds;} \\ \dfrac{\lambda_{\overline{T}_i}}{\sum\limits_{l \in Q_k} \left(1 - \delta(\sum\limits_{k \in K_l} n_k^a)\right)}, \text{if condition (6) holds.} \end{cases} \tag{25}$$

The transition rates $\mathbf{N_i^d}$ to $\mathbf{N}$ are the same as described for the SP-LQF control policy, in Equation (19). The transition rates $\mathbf{N_i^{a-d}}$ to $\mathbf{N}$ however have to be derived separately for this control policy. If condition (11) holds, then we clearly have $r_{ij}^{a-d} = \mu_j t_{j,i}$. If condition (12) holds, then following the LQF queue selection policy, we have the same transition rate as described for the SP-LQF policy: $r_{ij}^{a-d} = \sum\limits_{r \in \overline{Q}_j} \dfrac{\mu_r t_{r,i} \varepsilon_r(j)}{B_r}$, where B_r is the number of job classes that have the longest queue in the set $\{\overline{T}_f : f \in K_r\}$; and $\varepsilon_r(j) = 1$ if the queue of job class i is one of the B_r longest queues in the set $\{\overline{T}_f : f \in K_r\}$, and 0 otherwise. When condition (13) holds, the transition rate depends on the random routing principle. Note that for n_i^{a-d} to increase from zero to one, we need a completion of service for job class $\overline{T}_j$ at server j that is transformed into a job of class $\overline{T}_i$. Although necessary, this condition is not sufficient since a transfer of a job to class $\overline{T}_i$ may only be routed to server i with probability $\dfrac{1}{\sum\limits_{l \in Q_{\overline{T}_i}} \left(1 - \delta(\sum\limits_{k \in K_l} n_k^{a-d})\right)}$, where the denominator represents the number of servers that are idle and in contention to service the job of class $\overline{T}_i$ at the moment of transfer. Hence the transition rate when condition (13) holds is given by $r_{i,j}^a = \dfrac{\mu_j t_{j,\overline{T}_i}}{\sum\limits_{l \in Q_{\overline{T}_i}} \left(1 - \delta(\sum\limits_{k \in K_l} n_k^{a-d})\right)}$. When condition (14) holds, then the transition rate depends on the relative size of the queues as well as the random routing principle. Proceeding in the same fashion as in the SP-LQF control policy, we obtain that $r_{i,j}^{a-d} = \frac{1}{\sum_{\iota\in \bar{Q}_{\bar{T}_i}} \left(1-\delta(\sum_{k\in K_\iota} n_k^{a-d})\right)} \sum_{r\in \bar{Q}_j} \frac{\mu_r t_{r,i}\varepsilon_r(j)}{B_r}$.

We can similarly determine the rates at which the network experiences a self-transition, when in state **N**, as a result of a combination of a service completion and a service request. When condition (15) holds, the rate at which a service completion occurs is simply μ_i, and therefore the rate at which the job is transferred to class $\bar{T}_i$ is $\mu_i t_{i,\bar{T}_i}$. However for n_i to increase again from zero to one, we need this job that has returned to class $\bar{T}_i$ to be routed to server *i*, so as to effect a self-transition; this event occurs with probability that is the inverse of the number of servers idle and in contention to service a job of class $\bar{T}_i$. Hence again the transition rate is given by: $r_i = \frac{\mu_i t_{i,\bar{T}_i}}{\sum_{\iota\in Q_{\bar{T}_i}} \left(1-\delta(\sum_{k\in K_\iota} n_k)\right)}$.

Finally, when condition (16) applies, the rate at which a service completion occurs, depleting the queue for class *i*, is written using the LQF policy as $\sum_{r\in Q_j} \frac{\mu_r t_{r,i}\varepsilon_r(j)}{B_r}$, where B_r is the number of job classes that have the longest queue in the set $\{\bar{T}_f : f \in K_r\}$; and $\varepsilon_r(j)=1$ if the queue of job class *i* is one of the B_r longest queues in the set $\{\bar{T}_f : f \in K_r\}$, and 0 otherwise. Since the queue *i* gets replenished by this new service request at the same rate, we have the required transition rate: $r_i = \sum_{r\in Q_j} \frac{\mu_r t_{r,i}\varepsilon_r(j)}{B_r}$.

The RR-SP Policy

The rate r_i^a for the RR-SP policy is the same as in the RR-LQF policy. The rate r_i^d is given as follows; when condition (8) holds, a transition from $\mathbf{N_i^d}$ to **N** clearly occurs with rate: $r_i^d = \mu_i\left(1-\sum_{r=1}^{n} t_{i,r}\right)$. However, when condition (9) applies, the transition rate depends on the strict priority scheme. First note that for n_i^d to decrease by one, we need a departure from a server that belongs to the set $\overline{Q}_i$. Although necessary, this condition is not sufficient since a service completion at server *r* from the set $\overline{Q}_i$ may not decrease n_i^d by one, unless the job class *i* has the highest priority among all the job classes in queue that can be routed to the servers in the set K_r. This means that the following condition must be satisfied:

$$\gamma_{i,r} < \gamma_{\overline{T}_t,t} \ \forall r \in \overline{Q}_i; \forall t \in \mathrm{K}_r \text{ s.t. } n_{\overline{T}_t}^d \geq 2 - \delta\Big(\sum_{k \in \mathrm{K}_t, k \neq t} n^d{}_k\Big); v_{T_t} = 1;$$

Since the departure rate from server *r* is $\mu_r\left(1 - \sum_{z=1}^{n} t_{r,z}\right)$r, the transition rate r_i^d is given by:

$$r_i^d = \sum_{r \in \overline{Q}_i}\left(\mu_r \theta(\gamma_{i,r})\left(1 - \sum_{z=1}^{n} t_{r,z}\right)\right)$$

where

$$\theta(\gamma_{ir}) = \begin{cases} 1, \text{if } \gamma_{i,r}\left(1 - \delta\left(2 - \delta\Big(\sum_{k \in \mathrm{K}_t, k \neq t} n^d{}_k\Big)\right)\right) \leq \gamma_{\overline{T}_t,t}, & \forall r \in \overline{Q}_i; \forall t \in \mathrm{K}_r; v_{T_t} = 1; \\ 0, \quad \text{otherwise.} \end{cases} \tag{26}$$

The transition rates $\mathbf{N_i^{a-d}}$ to $\mathbf{N}$ are also derived as follows. If condition (11) holds, then we clearly have $r_{ij}^{a-d} = \mu_j t_{j,i}$. If condition (12) holds, then following the SP queue selection policy, we have the same rate of depletion of a job at queue *i* as was derived for the expression r_i^d. Hence we have a net transition rate: $r_{i,j}^{a-d} = \sum_{r \in \overline{Q}_j}\left(\mu_r \theta(\gamma_{i,r}) t_{r,i}\right)$, where $\theta(\gamma_{i,r})$ is defined in Equation (26). When condition (13) holds, the transition rate depends on the random routing principle. Note that for n_i^{a-d} to increase from zero to one, we need a completion of service for job class $\overline{T}_j$ at server *j* that is then transformed into a service request of class $\overline{T}_i$. Although necessary, this condition is not sufficient since a transfer of a job to class $\overline{T}_i$ may only be routed to server i with probability $\dfrac{1}{\sum_{\iota\in Q_{\overline{T}_i}}\left(1-\delta(\sum_{k\in K_\iota} n_k^{a-d})\right)}$,

where the denominator represents the number of servers that are idle and in contention to service the job of class $\overline{T}_i$ at the moment of transfer. Hence the transition rate when condition (13) holds is given by $r_{i,j}^{a} = \dfrac{\mu_j t_{j,\overline{T}_i}}{\sum_{\iota\in Q_k}\left(1-\delta(\sum_{k\in K_\iota} n_k^{a-d})\right)}$. When condition (14) holds, then the transition rate $r_{i,j}^{a-d}$ depends on the SP queue selection policies as well as on the random routing principle. Proceeding in similar fashion as in the SP-LQF and the RR-LQF policies, we obtain that $r_{i,j}^{a-d} = \dfrac{\sum_{r\in Q_j}\left(\mu_r\theta(\gamma_{i,r})t_{r,i}\right)}{\sum_{\iota\in \overline{Q}_i}\left(1-\delta(\sum_{k\in K_\iota} n_k^{a-d})\right)}$.

We then determine the rates at which the network experiences a self-transition, when in state **N**, as a result of a combination of a service completion and a service request. When condition (15) holds, the rate at which a service completion and a service request both occur at server i, is given by $r_i = \dfrac{\mu_i t_{i,\overline{T}_i}}{\sum_{\iota\in Q_{\overline{T}_i}}\left(1-\delta(\sum_{k\in K_\iota} n_k)\right)}$. This is derived in the same manner as for the RR-SP policy. Further, when we account for inter-class transitions that are invalidated as a result of WIP capacity bounds, we add the transition rates derived for conditions (11) and (13) for the same control policies to compute the effective transition rate. However, when condition (16) applies for the case of a valid return of a job to its same class after completion of service, the rate at which a service completion occurs, depleting the queue for class i, is written using the SP policy as $\sum_{r\in Q_i}\left(\mu_r\theta(\gamma_{i,r})\right)$. Since the queue i gets replenished by this new service request at the same rate, we have the required transition rate: $\sum_{r\in Q_i}\left(\mu_r\theta(\gamma_{i,r})t_{r,i}\right)$. Again, when we account for inter-class transitions that are invalidated as a result of WIP capacity bounds, we add the transition rates derived for conditions (12) and (14) for the same control policies to compute the effective transition rate.

The SP-SP Policy

The rate r_i^a for the SP-SP is the same as in the SP-LQF policy, while the rate r_i^d is the same as in the RR-SP policy. The transition rates $\mathbf{N_i^{a\text{-}d}}$ to $\mathbf{N}$ are derived as follows. If condition (11) holds, then we clearly have $r_{ij}^{a-d} = \mu_j t_{j,i}$. If condition (12) holds, then following the SP queue selection policy, we have the same rate of depletion of a job at queue i as was derived for the expression. Hence we have a net transition rate: $r_{i,j}^{a-d} = \sum_{r \in \overline{Q}_j} (\mu_r \theta(\gamma_{i,r}) t_{r,i})$, where $\theta(\gamma_{i,r})$ is defined in Equation (26). When condition (13) holds, the transition rate depends on the SP routing policy. Note that for n_i^{a-d} to increase from zero to one, we need a completion of service for job class $\overline{T}_j$ at server j that is transformed into a job of class $\overline{T}_i$. Although necessary, this condition is not sufficient since a transfer of a job to class $\overline{T}_i$ may not be routed to server i unless server i has the highest priority among those available to process job $\overline{T}_i$. This means that the following condition

$$\alpha_{\overline{T}_i,i} < \alpha_{\overline{T}_i,t} \ \forall t \in \overline{Q}_{\overline{T}_i}; \text{s.t. } \delta(\sum_{k \in K_t} n_k^{a-d}) = 0$$

must be satisfied. Since the transfer rate of jobs of class $\overline{T}_j$ to class $\overline{T}_i$ is $\mu_j t_{j,i}$, the transition rate r_{ij}^{a-d} is given by:

$$r_i^{a-d} = \mu_j t_{j,i} \gamma(\alpha_{\overline{T}_i,i}),$$

where $\gamma(\alpha_{\overline{T}_i,i})$ is as defined in Equation (20). When condition (14) holds, then the transition rate $r_{i,j}^{a-d}$ depends on the SP queue selection policies as well as on the SP routing policies. Proceeding in similar fashion as in the derivation in previously shown policies, we obtain that $r_{i,j}^{a-d} = \gamma(\alpha_{\overline{T}_i,i}) \sum_{r \in \overline{Q}_j} (\mu_r \theta(\gamma_{i,r}) t_{r,\overline{T}_i})$. We then turn to the task of determining the rates at which the network experiences a self-transition, when in state $\mathbf{N}$, as a result of a combination of a service completion and a service request. When condition (15) holds, the rate at which a service completion and a service request both occur at server *i*, is given by $r_i = \mu_i t_{i,i} \gamma(\alpha_{\overline{T}_i,i})$. When we account for inter-class transitions that are invalidated as a result of WIP capacity bounds, we add the transition rates derived for conditions (11) and (13) for the same control policies to compute the effective transition rate. Finally, when condition (16) applies, the rate at which a service completion occurs, depleting the queue for class *i*, is written using the SP policy as $\sum_{r \in Q_i} (\mu_r \theta(\gamma_{i,r}))$. Since the queue *i* gets replenished by this new service request at the same rate, we have the required transition rate: $\sum_{r \in \overline{Q}_i} (\mu_r \theta(\gamma_{i,r}) t_{r,i})$. Once again, when we account for inter-class transitions that are invalidated as a result of WIP capacity bounds, we add the transition rates derived for conditions (12) and (14) for the same control policies to compute the effective transition rate.

The transition rates for the remaining policies, namely SP-RS and RR-RS, can be determined in a similar fashion. The details are omitted.

3.3 Performance Measures

From $p(\mathbf{N})$, we can obtain the marginal probability $p(n_i)$ associated with the state variable n_i. Our primary measure of performance is throughput for each job class *i* which can be obtained as follows:

$$\tau_i^P = \lambda_i (1 - p(\sum_{j \in \overline{Q}_i} n_j = b_i)) \tag{27}$$

from which, we can then obtain the overall network throughput as:

$$\tau_s = \sum_{i=1}^{n} \tau_i^P. \tag{28}$$

We can also derive expressions for throughput due to each server *j* as:

$$\tau_j^R = \mu_j (1 - p(\sum_{r \in K_j} n_r = 0)). \tag{29}$$

Several other measures of expected performance can be obtained as well, including the expected queue size for each job class, average utilization of each server, and expected total WIP in the network. Significantly, using this framework, it is possible to obtain the variance of the throughput of job class, or each server in the network. Finally, variances of WIP levels in the network for each job class can also be obtained using the evaluation scheme described in the previous section.

4 Numerical Examples and Insights on Single-Stage Systems

4.1 Evaluation of flexibility mechanisms on throughput measures

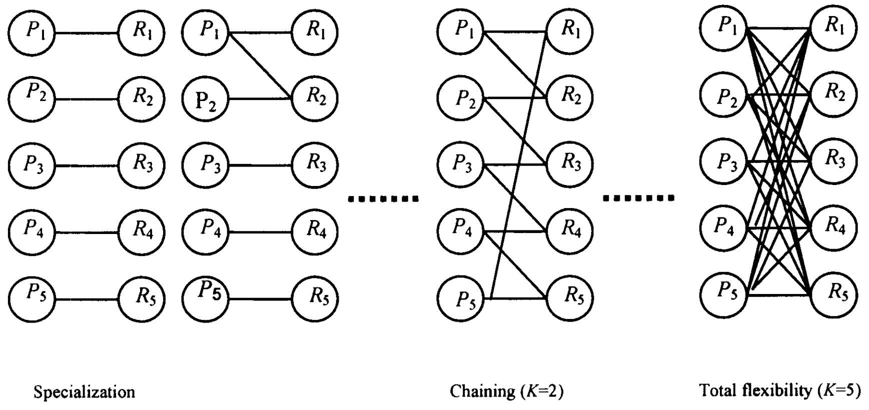


Figure 11 – Effect of Increasing Flexibility in a Queueing System

To illustrate the application of our models with a small example, consider a basic queueing system of the type shown in Figure 11. Consider also for a moment, that it is only throughput that is a performance measure of interest to us. Let λ_i denote the arrival rate of each customer (i = 1, ..., n), where the arrivals are assumed to follow the Poisson process. Similarly, let μ_j denote the processing rate of each server (j = 1, ..., m), where the processing time for each server is an exponentially distributed random variable with mean $1/\mu_j$. Finally, let a_{ij} denote the binary variable (0/1) that indicates whether customer i can be processed at server j. In addition to specifying feasible customer-server assignments, the analysis of the example flexible queueing system requires the specification of a control policy. The control policy is applied at each decision epoch. Decision epochs are triggered by either the arrival of a customer or the completion of service by a server. When a customer arrives and finds multiple idle servers, the control policy specifies which server is selected. Similarly, when a server completes service and finds more than one customer in the queue, the control policy specifies which customer is selected next for service. Hence a control policy is defined by a server selection rule and a customer selection rule.

Consider for the control of the example queueing system, the following four policies. Under the first policy, policy C_1, servers and customer classes are assigned priorities based on their flexibility. The server with the least flexibility is assigned highest priority and the same for customers. Flexibility is measured by the number of customers a server can process or the number of servers to which a customer can be assigned. Under the second policy, policy C_2, servers are assigned priorities based on the ratio of the sum of arrival rates of all customers that can be assigned to a server to the server's processing rate. Each server j ($j = 1, \ldots, m$) is assigned a priority based on the ratio $\sum_{i=1}^{n} a_{ij}\lambda_i / \mu_j$ with lower ratios corresponding to higher priorities. Similarly, customers are assigned priorities based on the ratio of a customer's arrival rate to the sum of the processing rates of servers to which the customer can be assigned. That is, each customer i ($i = 1, \ldots, n$) is assigned a priority based on the ratio $\lambda_i / \sum_{j=1}^{m} a_{ij}\mu_j$, with higher ratios corresponding to higher priorities. Hence, policy C_2 gives priority to servers with the least potential load and to customers with the least potential available capacity. Under the third policy, policy C_3, servers (customers) are assigned priorities based on their service (arrival) rates with higher rates corresponding to higher priorities. The fourth policy, C_4, customers choose a server at random from the pool of available servers, and similarly servers choose for processing a customer from the queue in random fashion, but in keeping with the process flexibilities.

Recall now that we consider a system with five customer classes and five servers. We consider 21 flexibility configuration scenarios. We start with a dedicated scenario in which each customer can be routed to only one server and each server can process only one customer (scenario 1). In scenarios 2-6, we increase flexibility by adding one link at a time between customers and servers until we reach a chained configuration (i.e., in scenario 2 customer 1 can be assigned to servers 1 or 2, in scenario 3, customer 1 can be assigned to servers 1 or 2 and customer 2 can be assigned to servers 2 or 3, etc). In scenarios 7-21, we further increase the flexibility one customer at a time and one link at a time starting with customer 1 until each customer has full flexibility (e.g., in scenario 7, customer 1 can be assigned to either servers 1, 2 or 3, in scenario 8, it can be assigned to either servers 1, 2, 3 or 4, etc). Scenario 21 corresponds to a system with full flexibility where any customer can be routed to any server and any server can process any customer.

We consider five levels of system loading, $L_1 = 0.6$, $L_2 = 0.9$, $L_3 = 1.2$, $L_4 = 1.5$, and $L_5 = 1.8$, where $L_i = \sum_{i=1}^{n} \lambda_i / \sum_{j=1}^{m} \mu_j$ for $i = 1, \ldots, 5$. We also consider four levels of demand and service rate heterogeneity, H_1, H_2, H_3, and H_4. For level H_1, $\mu_1 = 0.1$ and $\mu_j = \mu_{j-1} + 0.45$ for $j = 2,\ldots, 5$, and $\lambda_5 = 0.1 \times L_1$, and $\lambda_i = \lambda_{i+1} + 0.45 \times L_i$ for $i = 2, \ldots, 5$. For heterogeneity level H_2, we assign equal processing rates and services rates to all the customers and all the servers. For level H_3, we invert the assignments in level H_1, by setting $\mu_5 = 0.1$ and $\mu_j = \mu_{j+1} + 0.45; j = 1 \ldots 4$, and by letting $\lambda_1 = 0.1 \times L_1$; $\lambda_i = \lambda_{i-1} + 0.45 \times L_i$ for $i = 2, \ldots, 5$. For level H_4, the demand rates are the same as in H_1 and the service rates are the same as in H_3.

The effect of the different control policies for the different flexibility configurations and heterogeneity levels is illustrated in Figures 12-14 (the results are shown for systems with buffer or queue capacity levels $b_i = 3$ for $i = 1,\ldots, 5$, but are qualitatively similar for other buffer values). As we can see, from Figures 12 and 13, policy C_2 dominates the other policies suggesting that an optimal policy would take into account both the flexibility of servers and customers as well as the capacity available to the customers relative to their demand rates. The results also suggest that when there is significant heterogeneity in demand and service rates, assigning priorities based on these rates could be more helpful than assigning priorities based on flexibility (see Figure 12). Figure 13 illustrates how higher flexibility under a sub-optimal policy, such as C_3 or C_4, could lead to lower throughput (this effect is observed even when there is symmetry in demand and service rates).

Figures 12 and 13 also illustrate how the effect of increasing flexibility could be different depending on the heterogeneity in demand and service rates. In particular, flexibility is useful when it is associated with either the fastest servers or the customers with the greatest demand (this explains the observed jumps in throughput with one step increases in flexibility). The results also show that while there could be value to choosing a good control policy, well designed process flexibility configurations can have more influence on performance than control policies alone. In other terms, an increase in flexibility, if it is carefully designed and effected, can lead to significantly larger improvements in throughput than can be achieved by improving control policies alone.

Finally, we note that the effect of flexibility configuration is highly conditional on the heterogeneity in demand and service rates. For the same number of links between customers and servers (and for the same total system capacity and demand), throughput can vary widely depending on how capacity (demand rates) is distributed among the servers (customers). This is illustrated in Figure 4 where the percentage difference in throughput between systems with heterogeneity type H_1 and systems with heterogeneity type H_3 is shown for different levels of flexibility and different levels of loading. We observe that an increase in flexibility can switch the ordering of the two systems in either direction, so that an increase in flexibility can determine the usefulness of a particular distribution or allocation of capacity and demand rates, over another allocation scheme. This provides further support for the assertion that when possible, the design of flexibility mechanisms at the strategic, tactical and operational levels must be conducted in an integrated fashion.

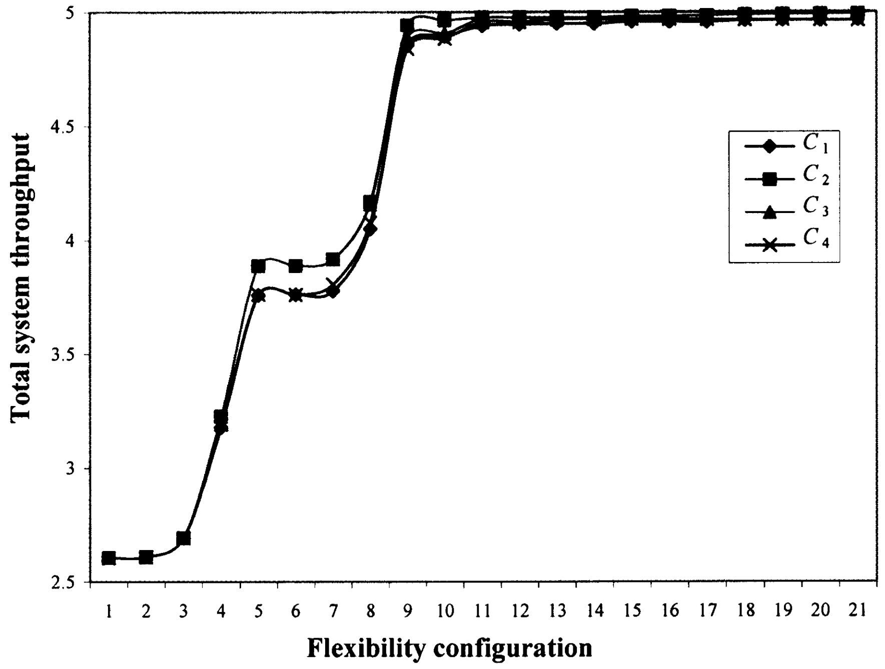


Figure 12 – The joint effect of control policies and flexibility (scenario H_1-L_5)

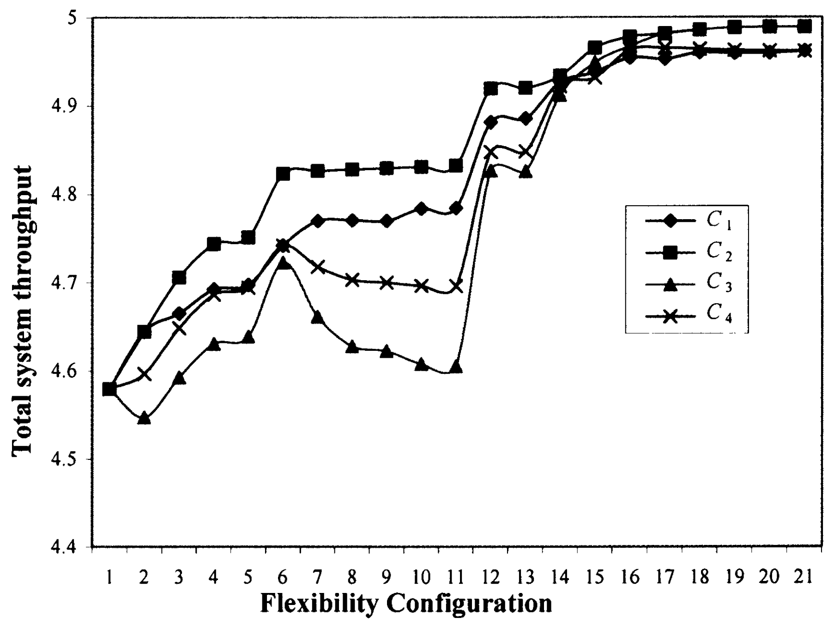


Figure 13 – The interaction between control policies and flexibility configuration (scenario H_4- L_5)

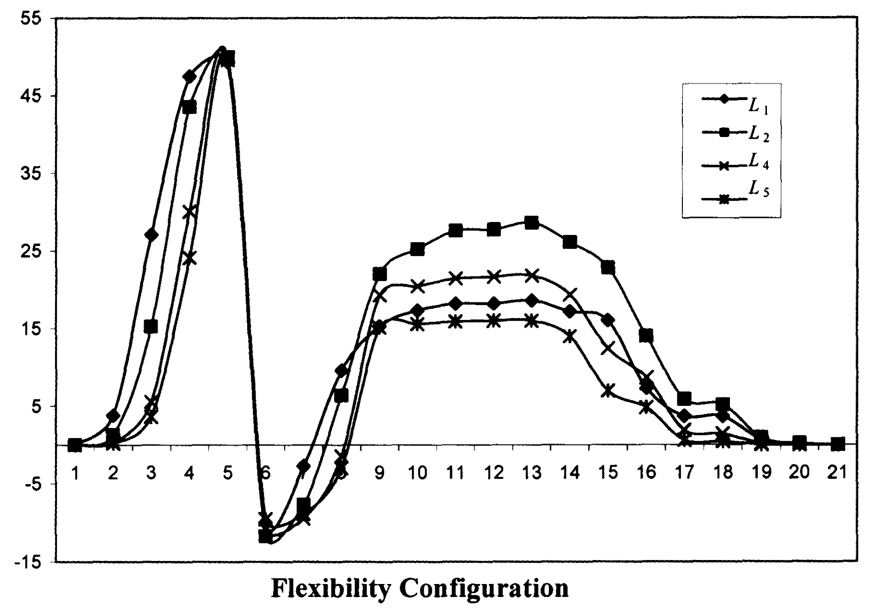


Figure 14 – The effect of flexibility on the percentage difference in throughput between systems H_1 and H_3 (policy C_2)

4.2 Evaluation of flexibility mechanisms on revenue-based measures

In this sub-section, we illustrate the importance of operational control policies to system managers who measure the performance of their system not only by throughput or utilization criteria, but also by revenue and cost considerations. In particular, we examine for varying assumptions on the service cost structures and revenue profiles for the various job classes, the performance of commonly used cost and revenue based control policies as compared to control policies of the type shown in section 4.1 that do not explicitly capture the revenue or cost considerations. The setting for the numerical analysis is still the same system as shown in Figure 11. The only difference is that we allow for heterogeneity in not only the arrival rates for job classes, and service rates for the different servers, but also in the unit costs of processing at the different servers and the unit revenues resulting from the arrival of jobs of different classes into the system. Table 1 shows the cost and revenue profiles that we consider for servers and the job classes.

Code	Revenue Profile of Customers	Cost Profile of Servers
V_1	Uniform	Linear in Service Rate
V_2	Uniform	Quadratic in Service Rate
V_3	Linear in inverse of arrival rate	Linear in Rate
V_4	Linear in inverse of arrival rate	Quadratic in Service Rate
V_5	Quadratic in inverse of arrival rate	Linear in Rate
V_6	Quadratic in inverse of arrival rate	Quadratic in Service Rate
V_7	Uniform	Linear in Flexibility
V_8	Uniform	Linear in Flexibility and in Service Rate
V_9	Linear in inverse of arrival rate	Linear in Flexibility
V_{10}	Quadratic in inverse of arrival rate	Linear in Flexibility
V_{11}	Linear in inverse of arrival rate	Linear in Flexibility and in Service Rate
V_{12}	Quadratic in inverse of arrival rate	Linear in Flexibility and in Service Rate

Table 1 – Revenue and cost profiles considered for the numerical analysis

Essentially, we repeat the numerical experiment shown in section 4.1 but with some changes in parameter values, and for a broader set of control policies. We consider only 11 flexibility configuration scenarios. We start with the chaining process flexibility scenario in which each customer can be routed to its two adjacent servers to obtain the third configuration (denoted as K=2) shown in Figure 11. In scenarios 2-11, we increase flexibility by adding one link at a time between customers and servers so that we progress towards higher orders of process flexibility (e.g., in scenario 2, customer 1 can be assigned to either servers 1, 2 or 3, in scenario 3, it can be assigned to either servers 1, 2, 3 or 4, etc). We consider four levels of system loading, $L_1 = 0.5$, $L_2 = 0.75$, $L_3 = 1.0$, and L_5=1.25 where $L_i = \sum_{i=1}^{n} \lambda_i / \sum_{j=1}^{m} \mu_j$ for i = 1, …, 5. We again consider four levels of demand and service rate heterogeneity, H_1, H_2, H_3, and H_4. For level H_1, $\mu_1 = 0.1$ and $\mu_j = \mu_{j-1} + 0.45$ for j = 2,…, 5, and $\lambda_5 = 0.1 \times L_1$, and $\lambda_i = \lambda_{i+1} + 0.45 \times L_i$ for i = 2, …, 5. For heterogeneity level H_2, we assign equal processing rates and services rates to all the customers and all the servers. For level H_3, we invert the assignments in level H_1, by setting $\mu_5 = 0.1$ and $\mu_j = \mu_{j+1} + 0.45; j = 1...4$, and by letting $\lambda_1 = 0.1 \times L_1$; $\lambda_i = \lambda_{i-1} + 0.45 \times L_i$ for i = 2, …, 5. For level H_4, the demand rates are the same as in H_1 and the service rates are the same as in H_3. Finally, we consider the 12 levels of revenue and cost related heterogeneity among customers and servers respectively that are shown in Table 1.

Consider now for the control of the example queueing system described, the following 15 control policies. Under the first policy, policy C_1, servers are assigned priorities based on the ratio of the sum of arrival rates of all customers that can be assigned to a server to the server's processing rate. Each server j ($j = 1, \ldots, m$) is assigned a priority based on the ratio $\sum_{i=1}^{n} a_{ij}\lambda_i / \mu_j$ with lower ratios corresponding to higher priorities. Similarly, customers are assigned priorities based on the ratio of a customer's arrival rate to the sum of the processing rates of servers to which the customer can be assigned. That is, each customer i ($i = 1, \ldots, n$) is assigned a priority based on the ratio $\lambda_i / \sum_{j=1}^{m} a_{ij}\mu_j$, with higher ratios corresponding to higher priorities. Hence, policy C_1 gives priority to servers with the least potential load and to customers with the least potential available capacity. Under the second policy, policy C_2, servers (customers) are assigned priorities based on their service (arrival) rates with higher rates corresponding to higher priorities. In the third policy C_3, customers choose a server at random from the pool of available servers, and similarly servers choose for processing a customer from the queue in random fashion, but in keeping with the process flexibilities. For policies C_4 to C_{15}, we simply assign higher priorities to less expensive resources, and higher priorities to higher revenue customers, where the unit costs and unit revenues are derived according the cost and revenue profile schemes described in Table 1.

Briefly, we describe how we assign unit costs to servers and unit revenues to customers based on the profiles defined in Table 1. Firstly, for uniform rates, we simply assume that the unit costs and revenues for servers and customers alike are equal to 1. Unit costs that are linear in service rates are normalized and set equal to $c_j = \dfrac{\mu_j}{\sum_{j=1}^{m} \mu_j}$,

since we can ensure by doing so that the ratio of unit costs between any two servers is preserved in accordance with the linear relationship of cost with service rates. Using similar logic, unit costs that are quadratic are set equal to $\dfrac{\mu_j^{2}}{\sum_{j=1}^{m} \mu_j^{2}}$ Similarly unit revenues say for customer P_1 when linear in the inverse of arrival rates are normalized and set equal to $r_i = \frac{\lambda_5}{\sum_{i=1}^{n} \lambda_n}$, unit revenues for customer P_2 when linear in the inverse of arrival rates are normalized and set equal to $\frac{\lambda_4}{\sum_{i=1}^{n} \lambda_n}$ and so on in a cyclical fashion. Costs for a server j that are linear in process flexibility are computed as $\frac{\sum_{i=1}^{n} a_{ij}\lambda_i}{\sum_{j=1}^{m}\sum_{i=1}^{n} a_{ij}\lambda_i}$. Finally costs for a server j that are linear in process flexibility as well as in service rate are computed as $\left(\frac{\sum_{i=1}^{n} a_{ij}\lambda_i}{\sum_{j=1}^{m}\sum_{i=1}^{n} a_{ij}\lambda_i}\right)\left(\frac{\mu_j}{\sum_{j=1}^{m}\mu_j}\right)$.

Next, we develop a mechanism for the comparison of revenue and cost-based control policies with policies that are based on throughput considerations alone. Firstly the revenue performance of any system is computed simply as $\psi = \frac{\sum_i^n r_i \tau_i^P}{\sum_i^m c_j \tau_j^R}$. Secondly, the relative revenue performance index of a control policy C_k; k=4…15 is computed as $\mathrm{T} = \min\left\{\frac{\psi_k}{\psi_1}, \frac{\psi_k}{\psi_2}, \frac{\psi_k}{\psi_2}\right\}$. This measure of revenue performance indicates how well a revenue or cost-based control policy has performed for a system, over the *closest* performing control policy that is designed based only on throughput and capacity considerations. Figures 15 to 25 then describe the relative performance of revenue-based control policies for various settings of system parameters. Clearly lower ratios (<1.0) for T imply that revenue or cost based control policies are inefficient for the specific system scenarios under consideration. Higher ratios on the other hand imply that simple control policies based on revenue and cost considerations are relatively effective for the particular system. Based on such definitions and numerical analysis, we make the following brief observations regarding the design flexibility for revenue or cost based measures.

1. Simplistic revenue or cost based control policies can be either beneficial or can result in inferior revenue performance relative to demand or capacity

based control policies for the systems considered. The interesting observation for system managers however is that well designed control policies that are based on throughput and capacity management considerations and that work well for the process flexibility configurations and capacity allocation scheme in place, can in fact lead to superior performance on revenue based measures. This observation is supported by figures 18 through 20, where for a wide range of system parameter settings, the values of T, the relative performance index, were consistently lower than 1.0.

2. The effect of revenue or cost based control policies can change with the loading conditions. In some systems, as the load increases the positive impact of revenue or cost based control policies diminishes rapidly; this is observed in particular in Figure 17. In other systems the positive impact of revenue or cost based measures increases with the loading levels, whereas in some other cases one does not observe any change at all. While this accounts for all possible forms of relationships, a deeper analysis of the results does seem to indicate that revenue based control policies work well in lower loading conditions (or at lower utilization levels) where overall throughput is not influenced by control policies to a significant extent. When the throughput performance measures are invariant under different control policies, a redistribution of utilization levels and customer balking rates through carefully constructed revenue based policies can indeed improve revenue performance. On the other hand, it is also true that at higher loading levels the throughput performance of the system could be impaired by revenue based control policies, especially when the control policy offers a poor fit with the process flexibility configuration. In such situations, overall revenue measures are also inferior with revenue based control policies.

3. Heterogeneity levels in our experiments represent the different ways system managers can distribute or allocate capacity among the servers. Figure 23 demonstrates the relative significance of tactical flexibility mechanisms involving allocation or redistribution of capacity among servers on revenue performance. While the parameter settings have been chosen to emphasize the

differences between different capacity allocation schemes (in Figure 23 for example, one can obtain a two-fold improvement in revenue performance by allocating capacity differently to the servers), it is important to note the independent influence of this design factor on system performance.

4. Process flexibility has come to be understood as a mechanism for increasing overall system capacity through the sharing of demand from multiple job classes across servers, and has been shown to be particularly useful in cases of high variability in demand and service rate. The effect, therefore, of increasing process flexibility on the performance of revenue based control measures, should be in spirit, similar to the effect of decreasing the loading on the system. Where it is possible to discern the impact of revenue based control policies, for example in Figure 21, we see that an increase in process flexibility can lead to lowering of system utilization levels, which in turn offers system managers with the opportunity to deploy revenue based control policies.

5. Overall, the results again confirm the need to design flexibility mechanisms in the form of strategic process flexibility, tactical capacity allocation schemes and operational level control policies in an integrated fashion. The impact of these design factors on the revenue performance of the system are shown to be of some significance; this in turn warrants system managers to employ more careful analysis of decisions involving such design factors, and their impact on revenue performance.

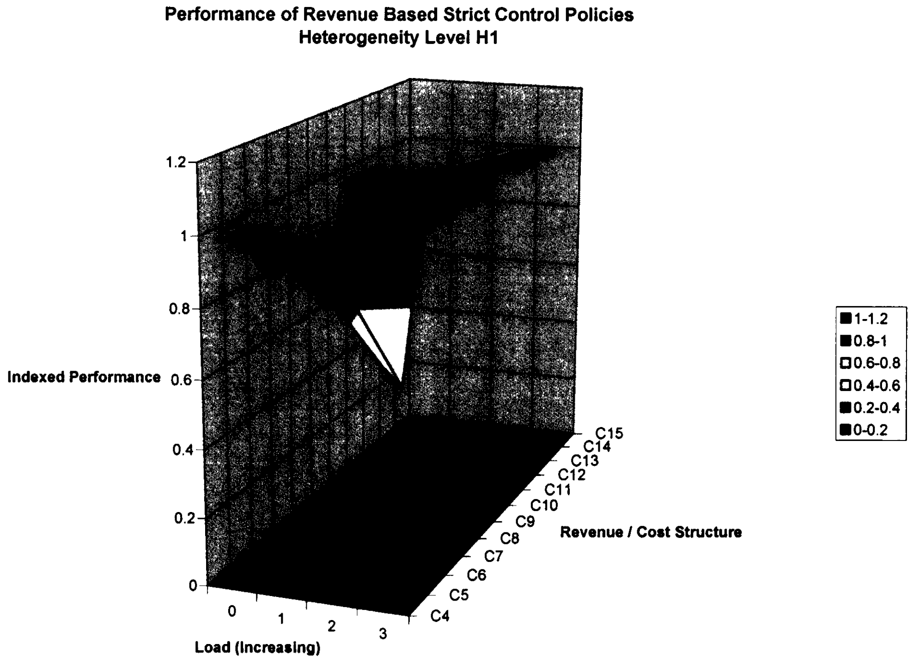


Figure 15 – The performance of revenue / cost based control policies for varying loads

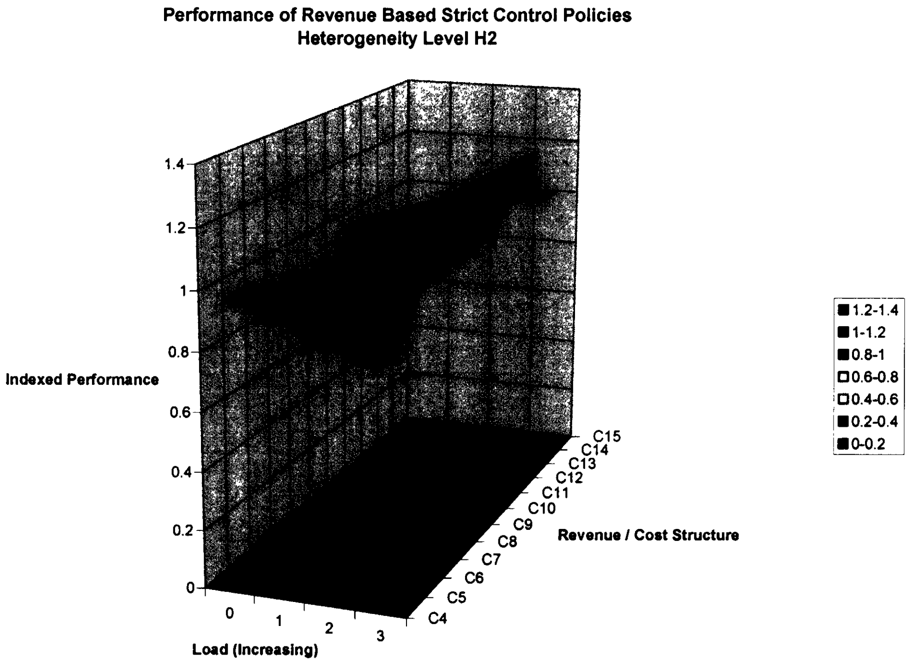


Figure 16 – The performance of revenue / cost based control policies for varying loads

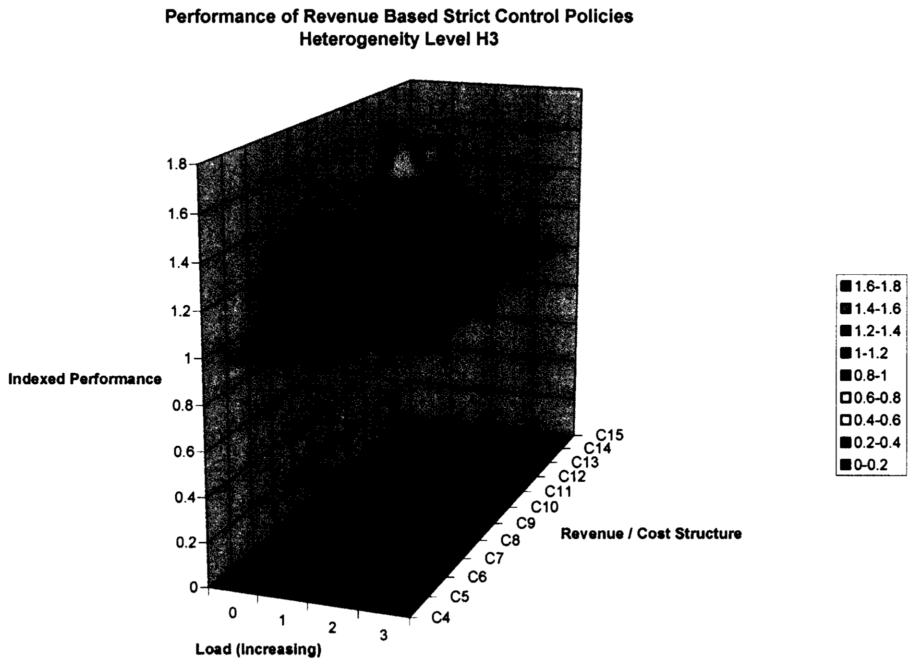


Figure 17 – The performance of revenue / cost based control policies for varying loads

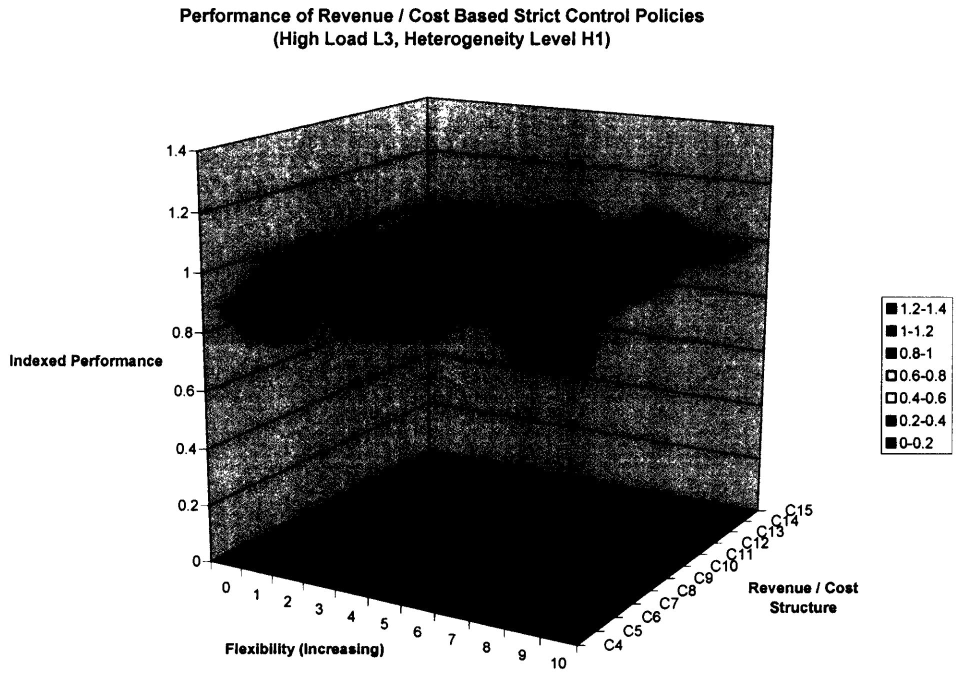


Figure 18 –Performance of revenue based control policies for varying process flexibilities

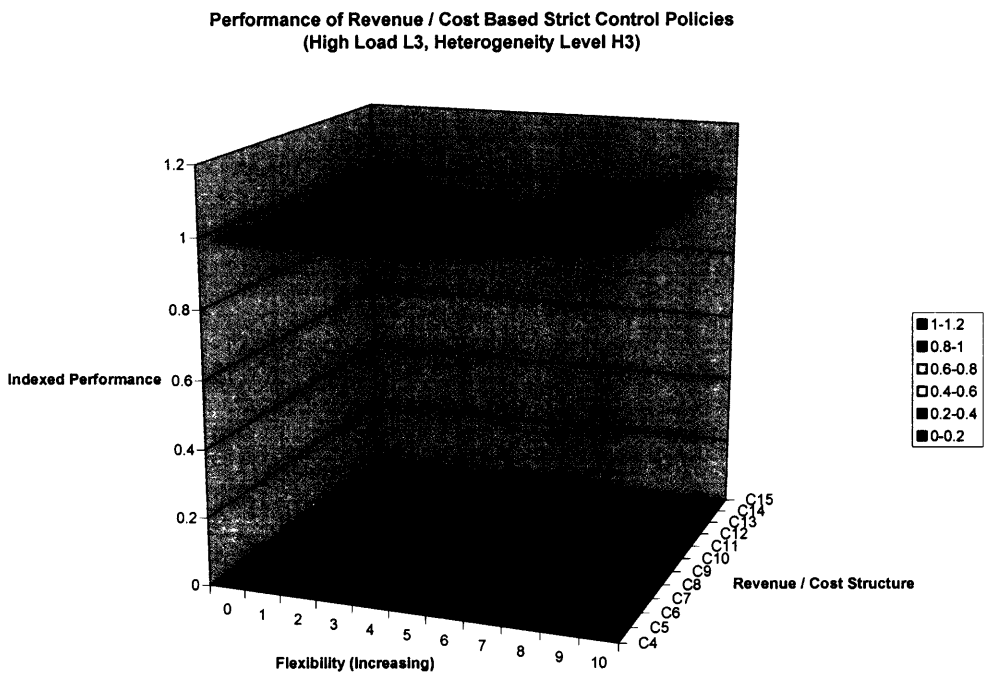


Figure 19 –Performance of revenue based control policies for varying process flexibilities

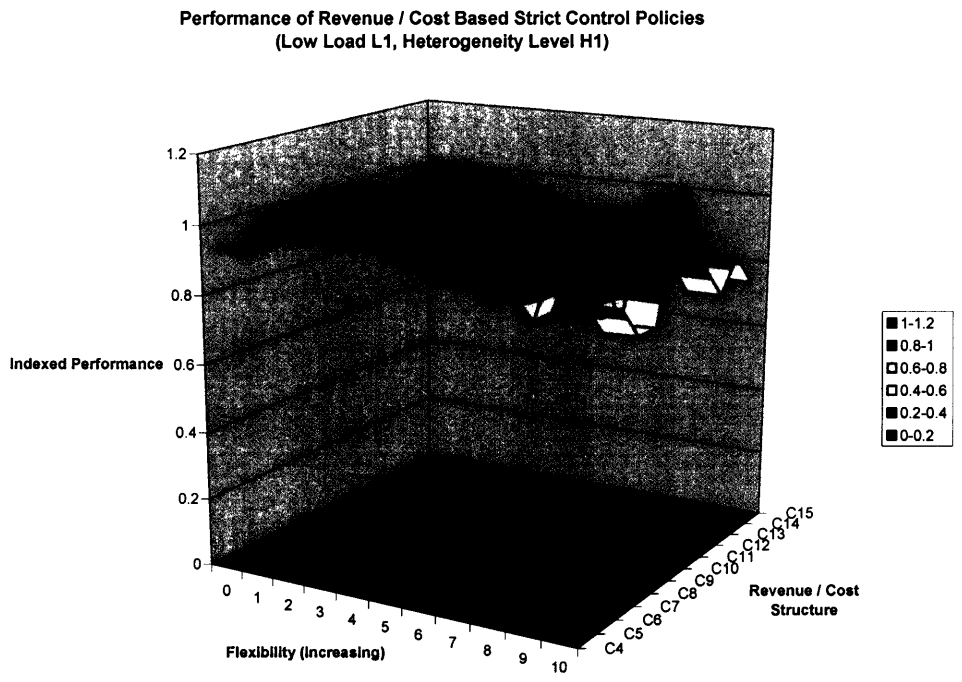


Figure 20 –Performance of revenue based control policies for varying process flexibilities

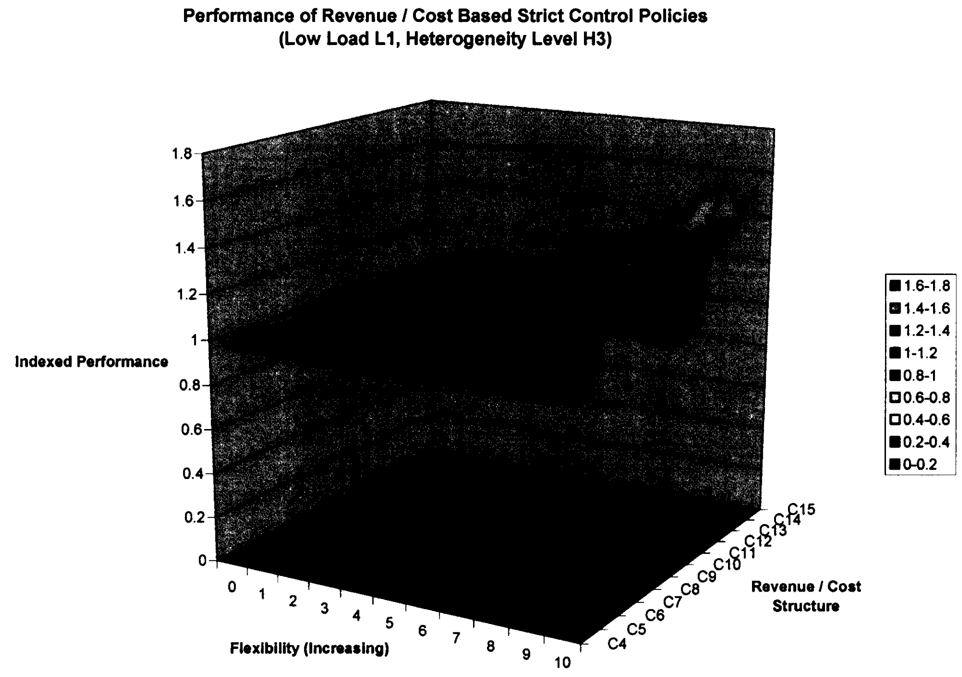


Figure 21 –Performance of revenue based control policies for varying process flexibilities

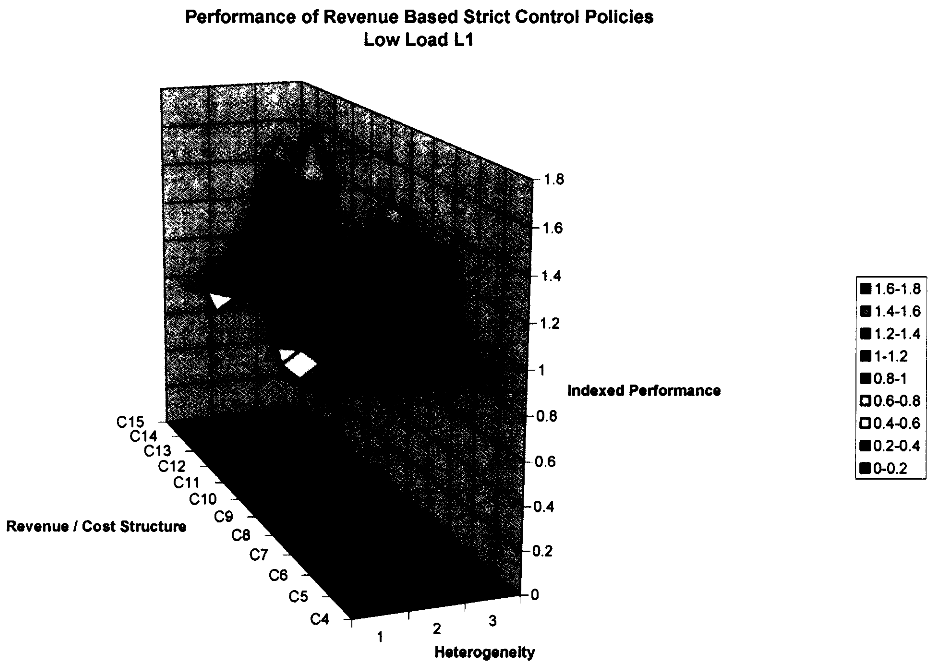


Figure 22 –Performance of revenue based control policies for varying heterogeneity levels (Scenario 11)

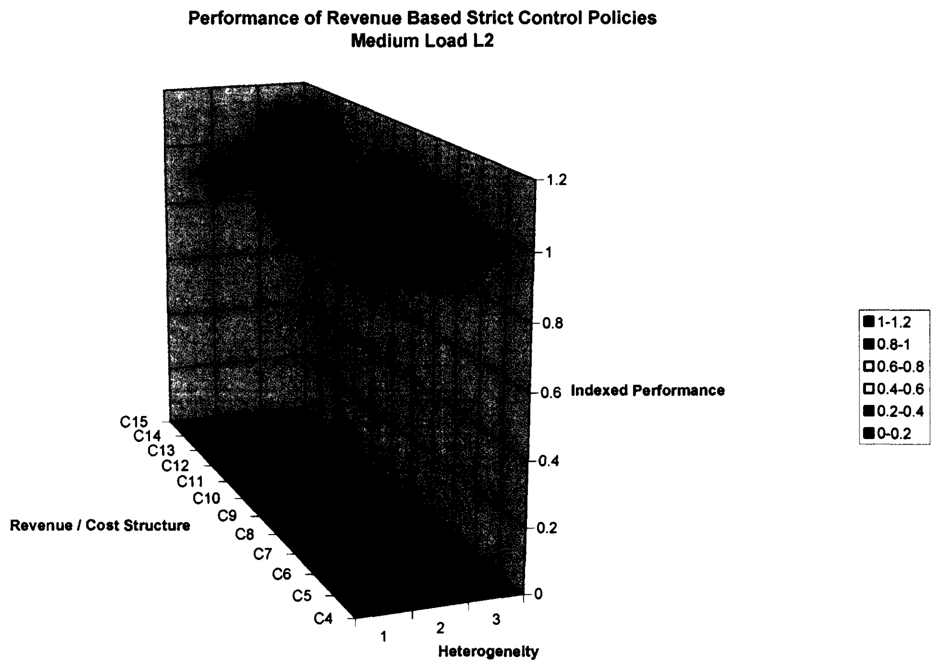


Figure 23 –Performance of revenue based control policies for Scenario 11

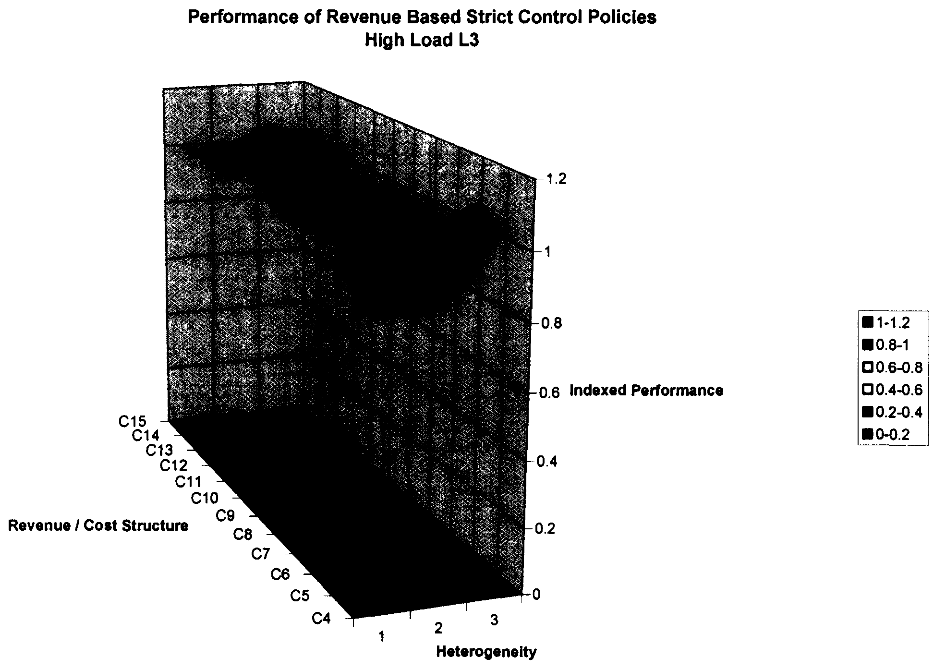


Figure 24 –Performance of revenue based control policies for Scenario 11

5 Conclusions and Future Work

In this thesis, we presented a framework for the representation, modeling and analysis of flexible queueing systems. The analytical model allows for the analysis of general system configurations with an arbitrary number of job classes and servers, arbitrary process flexibilities, heterogeneity in demand and service rates, and a wide range of control policies. The models are generic and can be used to analyze flexible queueing systems in a variety of applications even with the computational challenges that arise out of the level of detail in the analysis. In particular, for the purpose of comparing process flexibility configurations based on throughput measures, and the accompanying decisions on capacity allocation and control policies, these models could prove useful for developing insight through the analysis of relatively smaller scale models. Furthermore, our characterization of the probability distribution of system states and the transition probability between these states offers the opportunity to formulate optimal control problems (e.g., using the framework of a Markov decision process).

From a broader perspective, our study serves to highlight for the benefit of system managers the critical design factors that govern system performance along a number of performance dimensions including revenue and cost based measures. From an analytical perspective, our model can be extended in a variety of ways. This includes relaxing the assumptions of Poisson demand and exponential processing times and allowing service times to vary by customer and server. It would then be useful to examine the impact of demand and service variability on different system configurations and different control policies. In many applications, such as manufacturing, the processing of multiple customers on the same servers is accompanied by losses in efficiencies due to switchover times or costs. It would be worthwhile to extend the models to account for these loss factors. Finally, from an algorithmic perspective, it could prove worthwhile to explore better state representations, and also solution techniques that solve for the performance of such job-shop like queueing systems, and going beyond allow for a computationally efficient evaluation of alternative flexibility mechanisms that can lead to improved performance.

6 References

Ahuja, R. K., T. L. Magnanti and J. B. Orlin, *Networks Flows*, Prentice Hall, 1993.

Aksin, O. Z. and F. Karaesmen, "Designing Flexibility: Characterizing the Value of Cross-Training Practices," Working Paper, INSEAD, 2002. Available at www.ku.edu.tr/~fkaraesmen/pdfs/flex2102.pdf.

Buzacott, J. A., "Commonalities in Reengineered Business Processes: Models and Issues," *Management Science,* **42**, 1996, 768-782.

Buzacott, J.A., and J. G. Shanthikumar, *Stochastic Models of Manufacturing Systems*, Prentice Hall, 1993.

Cooper, R., *Introduction to Queueing Theory*, 2/E, North-Holland, New York, 1981.

Fine, C., and Freund, F., "Optimal Investment in Product Flexible Manufacturing Capacity", Management Science, **36**, 1990, pp. 449-466.

Gans, N., G. Koole and A. Mandelbaum, "Telephone Call Centers: Tutorial, Review and Research Prospects," *Manufacturing & Service Operations Management,* **5**, 2003, 79-141.

Garnett, O. and A. Mandelbaum, "An Introduction to Skills Routing and its Operational Complexities," Teaching Note, Technion, 2001. Available at www.ie.technion.ac.il/serveng/Homeworks/HW9.pdf.

Graves, S., "A Tactical Planning Model for a Job Shop," *Operations Research*, July-August 1986, Vol. 34, 522-533.

Gurumurthi S., and Benjaafar, S., "Modeling and Analysis of Flexible Queueing Systems", *Naval Research Logistics,* August 2004, Vol. 51, Issue 5, 755-782.

Hall, R.W., *Queueing Methods: For Services and Manufacturing*, Prentice Hall, 1991.

Hopp, W., E. Tekin, M.P. Van Oyen, "Benefits of Skill Chaining in Production Lines with Cross-trained Workers," Working Paper, Department of Industrial Engineering and Management Sciences, Northwestern University, Evanston, IL-60208-3119, 2001.

Jordan, W.J., and S.C. Graves, "Principles on the Benefits of Manufacturing Process Flexibility," *Management Science*, **41**, 1995, 577-594.

Kleinrock, L., *Queueing Systems: Computer Applications*, Vol. 2, Wiley, New York, 1976.

Koole. G., and A. Mandelbaum, "Queuing Models of Call Centers, An Introduction," *Annals of Operations Research*, **113**, 2002, 41–59.

Mandelbaum A., and M. Reiman, "On Pooling in Queueing Networks," *Management Science*, **44**, 1998, 971-981.

Ross, K. W., *Multiservice Loss Models for Broadband Telecommunication Networks*, Springer-Verlag, London, 1995.

Sennot, L. I., *Stochastic Dynamic Programming and the Control of Queueing Systems*, Wiley, New York, 1999.

Sheikhzadeh, M., Benjaafar, S., and D. Gupta, "Machine Sharing in Manufacturing Systems: Flexibility versus Chaining," *International Journal of Flexible Manufacturing Systems*, **10**, 1998, 351-378.

Shumsky, R. A., "Approximation and Analysis of a Call Center with Flexible and Specialized Servers," Working Paper, University of Rochester, 2003. Available at http://omg.simon.rochester.edu/omgHOME/shumsky/flex_serv.PDF.

Whitt, W., "Stochastic Models for the Design and Management of Customer Contact Centers: Some Research Directions," Working Paper, Columbia University, 2002. Available at www.columbia.edu/~ww2040/IEOR6707F02.html.

Appendix A1 : Computer Code for Single Stage Analysis

```
//SP_SP.cpp:Definestheentrypointfortheconsoleapplication.
//


#include<stdlib.h>
#include<iostream.h>
#include<fstream.h>
#include<string.h>
#include<stdio.h>
#include<math.h>


intmain(intargc,char*argv[])
{//0
charnames[25];
strcpy(names,"M");
ifstreaminFlexFile(names,ios::in);

if(!inFlexFile)
{//1
cerr<<"Filecouldnotbeopened\n";
exit(1);
}//1
floatFlexdata;
floattesta[25][25],indicator[10];

inta1,a2;
a1=0;
a2=0;


intcnt=0;
intcnt1=0;
introws=100;
intcols=100;
while(inFlexFile>>Flexdata)
{//1
testa[a1][a2]=Flexdata;
cnt=cnt+1;
if(cnt==2)
{//2
rows=testa[0][0];
cols=testa[0][1];
}//2
a2=a2+1;
if(a2==cols)
{//2
a1=a1+1;
a2=0;
}//2
}//1
intm,i,j,z,l1;
m=rows-1;
intQ2[10][10],T[10][10],R[10],C[10];
floatlam[10],mu[10];

for(i=0;i<=m-1;i++)
{//1
lam[i]=testa[i+1][m];
mu[i]=testa[i+1][m+1];
R[i]=testa[i+1][m+2];
for(j=0;j<=m-1;j++)
{//2
Q2[i][j]=testa[i+1][j];
}//2
C[i]=testa[0][i+3];
cout<<"CustomerPriority"<<i<<"="<<C[i]<<endl;
}//1
for(i=0;i<=m-1;i++)
{//1
for(j=0;j<=m-1;j++)
{//2
if(Q2[j][i]==1)
{//3
T[i][j]=Q2[j][i];
}//3
else
{//3
T[i][j]=0;
}//3

}//2
}//1


intp,b;
p=rows-1;
b=testa[0][2];

intst_sp_size;
st_sp_size=pow((b+1),m);

intstat_mat[100000][9];
intstat_flag[100000];

//FirstColumnofstate_matrixistheindexintothestatespace
for(i=0;i<=st_sp_size-1;i++)
{//1
stat_mat[i][0]=1;
stat_mat[i][1]=i+1;
stat_flag[i]=0;
}//1


//Alltheotherelementsinthematrix
intk=-1;
intcycle,n,l;
for(i=2;i<=m+1;i++)
{//1
l=0;
while(l<=st_sp_size-1)
{//2
k=k+1;
cycle=pow((b+1),(m+1-i));
if(k>b)
{//3
k=0;
}//3
n=1;
while(n<=cycle)
{//3
n=n+1;
stat_mat[l][i]=k;
l=l+1;
}//3
}//2
}//1
```

```
for(i=0;i<=st_sp_size-1;i++)
{//1
for(k=2;k<=m+1;k++)
{//2
if(stat_mat[i][0]!=0)
{//3
if(stat_mat[i][k]==0)
{//4
for(l=0;l<=m-1;l++)
{//5
if((Q2[l][k-2]==1)&&(stat_mat[i][l+2]>1))
{//6
stat_mat[i][0]=0;
}//6
}//5
}//4
}//3
}//2
}//1

intreal_size;
real_size=0;
for(i=0;i<=st_sp_size-1;i++)
{//1
if(stat_mat[i][0]!=0)real_size=real_size+1;
}//1


floatcount[29],temp[29],eps[29],beta[29],gam[29];
intflag,o,stat_indxa,stat_indxd,pow_val,flag1;
longpnter[100000];
floatrow_sum[100000];
for(j=0;j<st_sp_size;j++)
{//2
row_sum[j]=0.0;
}//2


for(i=0;i<29;i++)
{//1
count[i]=0.0;
temp[i]=0.0;
eps[i]=0.0;
beta[i]=0.0;
gam[i]=0.0;
}//1

strcpy(names,"LPFile.lp");
ofstreamoutLPFile(names,ios::out);
if(!outLPFile)
{//1
cerr<<"Filecouldnotbeopened\n";
exit(1);
}//1
strcpy(names,"Stat_mat");
ofstreamoutStat_mat(names,ios::out);
if(!outStat_mat)
{//1
cerr<<"Filecouldnotbeopened\n";
exit(1);
}//1

outLPFile<<"\\ProblemName:LPTest\n";
outLPFile<<"Maximize\n";
outLPFile<<"obj:1.0\n";
outLPFile<<"SubjectTo\n";
floattran_val,sum_col;
tran_val=0.0;
introwi,coli;
intcol_cnt;
col_cnt=0;
for(i=0;i<=st_sp_size-1;i++)
{//1
if(stat_mat[i][0]!=0)
{//2
sum_col=0.0;
col_cnt=0;
outLPFile<<"c"<<i<<":";
for(k=0;k<=m-1;k++)
{//3
temp[k]=0;
temp[k+m]=0;
}//3
for(k=0;k<=m-1;k++)
{//3
temp[k]=stat_mat[i][k+2]-1;
temp[k+m]=stat_mat[i][k+2]+1;
}//3
for(k=0;k<=m-1;k++)
{//3

stat_indxa=st_sp_size-1;
stat_indxd=st_sp_size-1;
for(l=0;l<=m-1;l++)
{//4
if(l!=k)
{//5
pow_val=pow((b+1),(m-l-1));
stat_indxa=stat_indxa-((b-stat_mat[i][l+2])*pow_val);
stat_indxd=stat_indxd-((b-stat_mat[i][l+2])*pow_val);
}//5
else
{//5
pow_val=pow((b+1),(m-l-1));
stat_indxa=stat_indxa-((b-temp[k])*pow_val);
stat_indxd=stat_indxd-((b-temp[k+m])*pow_val);
}//5
}//4
if(temp[k]==0)
{//4
tran_val=0.0;
flag=0;
for(l=0;l<=m-1;l++)
{//5
if((T[k][l]==1)&&(stat_mat[i][l+2]>1))
{//6
flag=1;
}//6
}//5
if(flag==0)
{//5
if(stat_mat[stat_indxa][0]!=0)
{//6
for(l=0;l<10;l++)
{//7
indicator[l]=1.0;
}//7
for(l=0;l<=m-1;l++)
{//7
if((stat_mat[i][l+2]==0))
{//8
for(l1=0;l1<=m-1;l1++)
{//9
if((Q2[l1][l]==1)&&(Q2[l1][k]==1))
{//10
if(R[l]<R[k])
{//11
indicator[l1]=0.0;
}//11
if((R[l]==R[k])&&(indicator[l1]>0.0))
{//11
if(l!=k)
{//12
indicator[l1]=indicator[l1]+1.0;
}//12
}//11
}//10
}//9
}//8
}//7
for(l=0;l<=m-1;l++)//forallcustomerss
{//7
if((Q2[l][k]==1)&&(indicator[l]==1.0))
{
tran_val=tran_val+lam[l];
}
if((Q2[l][k]==1)&&(indicator[l]>1.0))
{
tran_val=tran_val+(lam[l]/indicator[l]);
}
}//7
if(tran_val>0.0)
{//7
if(stat_flag[stat_indxa]==0)
{
stat_flag[stat_indxa]=1;
outStat_mat<<stat_indxa<<"\t";
for(z=0;z<=m-1;z++)
{
outStat_mat<<stat_mat[stat_indxa][z+2]<<"\t";
}
outStat_mat<<endl;
}
if(col_cnt==0)
{//8
outLPFile<<tran_val<<"x"<<stat_indxa;


sum_col=sum_col+tran_val;
row_sum[stat_indxa]=row_sum[stat_indxa]+tran_val;
tran_val=0.0;
col_cnt=col_cnt+1;
}//8
else
{//8
outLPFile<<'+'<<tran_val<<"x"<<stat_indxa;
row_sum[stat_indxa]=row_sum[stat_indxa]+tran_val;
tran_val=0.0;
col_cnt=col_cnt+1;
}//8
}//7
}//6
}//5
}//4
elseif((temp[k]<b)&&(temp[k]>0))
{//4
tran_val=0.0;
flag=0;
//Component4
if(flag==0)
//Component5
{//5
if(stat_flag[stat_indxa]==0)
{
stat_flag[stat_indxa]=1;
outStat_mat<<stat_indxa<<"\t";
for(z=0;z<=m-1;z++)
{
outStat_mat<<stat_mat[stat_indxa][z+2]<<"\t";
}
outStat_mat<<endl;
}
tran_val=lam[k];
//Component5
if(col_cnt==0)
{//7
outLPFile<<tran_val<<"x"<<stat_indxa;
row_sum[stat_indxa]=row_sum[stat_indxa]+tran_val;
sum_col=sum_col+tran_val;
tran_val=0.0;
col_cnt=col_cnt+1;
}//7
else
{
outLPFile<<'+'<<tran_val<<"x"<<stat_indxa;
sum_col=sum_col+tran_val;
row_sum[stat_indxa]=row_sum[stat_indxa]+tran_val;
tran_val=0.0;
col_cnt=col_cnt+1;
}
}//5
}//4
//ArrivalstatesandratestoNhavebeendefined.
//Nowwedefinethedeparturestatesandrates
if(temp[k+m]==1)
{//4
tran_val=0.0;
flag=0;
//Component6
if(flag==0)
//Component7
{//5
if(stat_mat[stat_indxd][0]!=0)
{//6
tran_val=mu[k];
//Component7
if(stat_flag[stat_indxd]==0)
{
stat_flag[stat_indxd]=1;
outStat_mat<<stat_indxd<<"\t";
for(z=0;z<=m-1;z++)
{
outStat_mat<<stat_mat[stat_indxd][z+2]<<"\t";
}
outStat_mat<<endl;
}
if(col_cnt==0)
{//7
outLPFile<<tran_val<<"x"<<stat_indxd;
sum_col=sum_col+tran_val;
row_sum[stat_indxd]=row_sum[stat_indxd]+tran_val;
tran_val=0.0;
col_cnt=col_cnt+1;
}//7
else
{//7
outLPFile<<'+'<<tran_val<<"x"<<stat_indxd;
sum_col=sum_col+tran_val;
row_sum[stat_indxd]=row_sum[stat_indxd]+tran_val;
tran_val=0.0;
col_cnt=col_cnt+1;
}//7
}//6
}//5
}//4

//Component8
elseif((temp[k+m]<=b)&&(temp[k+m]>=2))
{//4
tran_val=0.0;
```

```
flag=0;
if(stat_mat[stat_indxd][0]!=0)
{//6

for(l=0;l<10;l++)
{
indicator[l]=1.0;
}
for(l=0;l<=m-1;l++)
{//7
if((stat_mat[i][l+2]>0))
{//8
for(l1=0;l1<=m-1;l1++)
{//9
if(stat_mat[i][l1+2]>1)
{//10
if((Q2[l1][l]==1)&&(Q2[k][l]==1))
{//11
if(C[l1]<C[k])
{//12
indicator[l]=0.0;
}//12
if((C[l1]==C[k])&&(l1!=k))
{//12
if(indicator[l]>0.0)
{//13
indicator[l]=indicator[l]+1.0;
}//13
}//12
}//11
}//10
}//9
}//8
}//7
for(l=0;l<=m-1;l++)
{//7
if((Q2[k][l]==1)&&(indicator[l]==1.0))
{
if(stat_mat[i][l+2]>0)
{
tran_val=tran_val+mu[l];
}
}
if((Q2[k][l]==1)&&(indicator[l]>1.0))
{
if(stat_mat[i][l+2]>0)
{
tran_val=tran_val+mu[l]/indicator[l];
}
}
}
if(tran_val>0.0)
{
if(stat_flag[stat_indxd]==0)
{
stat_flag[stat_indxd]=1;
outStat_mat<<stat_indxd<<"\t";
for(z=0;z<=m-1;z++)
{
outStat_mat<<stat_mat[stat_indxd][z+2]<<"\t";
}
outStat_mat<<endl;
}
if(col_cnt==0)
{//7
outLPFile<<tran_val<<"x"<<stat_indxd;
sum_col=sum_col+tran_val;
row_sum[stat_indxd]=row_sum[stat_indxd]+tran_val;
tran_val=0.0;
col_cnt=col_cnt+1;
}//7
else
{//7
outLPFile<<'+'<<tran_val<<"x"<<stat_indxd;
sum_col=sum_col+tran_val;
row_sum[stat_indxd]=row_sum[stat_indxd]+tran_val;
tran_val=0.0;
col_cnt=col_cnt+1;
}//7
}
}//6
}//4
}//3
pnter[i]=outLPFile.tellp();
outLPFile<<'-'<<""<<"x"<<i<<"=0.0"<<endl;
if(stat_flag[i]==0)
{
stat_flag[i]=1;
outStat_mat<<i<<"\t";
for(z=0;z<=m-1;z++)
{
outStat_mat<<stat_mat[i][z+2]<<"\t";
}
outStat_mat<<endl;
}
}//2
}//1

outLPFile<<"c"<<st_sp_size<<":";
intreal_cnt,l_cnt;
real_cnt=0;
l_cnt=0;
for(i=0;i<=st_sp_size-1;i++)
{//1
if(stat_mat[i][0]!=0)
{//2
if(real_cnt!=0)
{//3
if(real_cnt==real_size-1)
{//4
outLPFile<<"+x"<<i<<"=1.0"<<endl;
outLPFile<<"End";
}//4
elseif(l_cnt==10)
{
outLPFile<<"+x"<<i<<endl;
l_cnt=0;
}
else
{//4
outLPFile<<"+x"<<i;
l_cnt=l_cnt+1;
}//4
}//3
else
{//3
outLPFile<<"x"<<i;
l_cnt=l_cnt+1;
}//3
real_cnt=real_cnt+1;
}//2
}//1
for(i=0;i<=st_sp_size-1;i++)
{
if(stat_mat[i][0]!=0)
{//2
outLPFile.seekp(pnter[i]+1);
outLPFile<<row_sum[i];
}
}
```

```
return0;                                                    }
```

Appendix A2 : Computer Code for Multi-Stage Analysis

```
#include <stdio.h>
#include <math.h>
#include <time.h>
#include <stdlib.h>

int main()
{//1

 //integer variable declarations

 int i, j , c1, c2, ok, row_indx, col_indx, M_row, M_col,
read_cnt, test_cnt, check_int, flag, k, k_bar;
 int m,z,l1, num_cus, num_res, wip_policy;
 int Q[50][50], T[50][50], Bsize[50], Respri[50][50],
Cuspri[50][50];
 int rt_policy, seq_policy;
 int num_aux_res;
 int num_unm_cus;
 int num_st_var;
 int no_arr_flag, no_dep_flag;

 //double variable declarations

 double M[15][15];
 double Mu[50][50], Phi[50][50];
 double Lam[50];

 //File variable declarations

 FILE *fpr, *fprr, *Stat_mat,*tran, *sparse, *problem_size,
*phi, *mu, *cuspri, *respri, *buffer, *lam ,*stat_test,
*control;
 FILE *nxt_sparse, *colstr, *rownd, *vals, *size_info;

 //Other (time variable declarations)

 clock_t start, end;

 system("cp M /tmp/");
 system("cp Mu /tmp/");
 system("cp Lam /tmp/");
 system("cp Buffer /tmp/");
 system("cp Control /tmp/");
 system("cp Phi /tmp/");
 system("cp Respri /tmp/");
 system("cp Cuspri /tmp/");



 fpr=fopen ("/tmp/M","r");
 Stat_mat=fopen ("/tmp/Stat_mat","w");
 problem_size=fopen ("/tmp/size_info", "w");

 test_cnt=0;

 for (row_indx=0;row_indx<M_row; row_indx++)
   {
     for (col_indx=0; col_indx<M_col; col_indx++)
{
 M[row_indx][col_indx]=0.0;
}
   }

 fscanf(fpr, "%lf", &M[0][0]);
 fscanf(fpr, "%lf", &M[0][1]);
 fscanf(fpr, "%lf", &M[0][2]);
 M_row=M[0][0];
 M_col=M[0][1];
 wip_policy=M[0][2];// if wip_policy==1 we have a local
bound on wip, otherwise a global bound.
 read_cnt=1;
 for (row_indx=0; row_indx<M_row; row_indx++)
   {//2
    for (col_indx=0; col_indx<M_col; col_indx++)
{
 if (read_cnt==1)
   {
    printf("%f ", M[0][0]);
    printf("No. of rows=%d\t",M_row);
    read_cnt++;
   }
 else if (read_cnt==2)
   {
    printf("%f ", M[row_indx][col_indx]);
    printf("No. of cols=%d\t",M_col);
    read_cnt++;
   }
 else if (read_cnt==3)
   {
    printf("%f ", M[row_indx][col_indx]);
    printf("WIP Policy=%d\t",wip_policy);
    read_cnt++;
   }
 else
   {
    fscanf(fpr, "%lf", &M[row_indx][col_indx]);
    printf("%f ", M[row_indx][col_indx]);
    read_cnt++;
   }
}
    printf("\n");
   }//2

 /* Defining Q, T, lam, mu, C and R*/
 num_aux_res=0;
 m=M_row-1;
 num_cus=M_row-1;
 num_res=M_col;

 phi=fopen ("/tmp/Phi","r");
 mu=fopen ("/tmp/Mu","r");
 cuspri=fopen ("/tmp/Cuspri","r");
 respri=fopen ("/tmp/Respri", "r");
 buffer=fopen ("/tmp/Buffer", "r");
 control=fopen ("/tmp/Control", "r");
 lam=fopen ("/tmp/Lam", "r");
 stat_test=fopen ("/tmp/stat_test", "w");

 read_cnt=1;
 test_cnt=0;

 for (row_indx=0;row_indx<50; row_indx++)
   {//2
    for (col_indx=0; col_indx<50; col_indx++)
{
 Mu[row_indx][col_indx]=0.0;
 Cuspri[col_indx][row_indx]=0;
      Respri[row_indx][col_indx]=0;
}
    for (col_indx=0; col_indx<50;col_indx++)
{
 Phi[row_indx][col_indx]=0.0;
}
    Bsize[row_indx]=0;
    Lam[row_indx]=0;
   }//2

 for (row_indx=0; row_indx<num_cus; row_indx++)
   {//2
    for (col_indx=0; col_indx<num_res; col_indx++)
{
 fscanf(mu, "%lf", &Mu[row_indx][col_indx]);
 fscanf(respri, "%d", &Respri[row_indx][col_indx]);
 fscanf(cuspri, "%d", &Cuspri[col_indx][row_indx]);
}
    for (col_indx=0; col_indx<num_cus; col_indx++)
{
 fscanf(phi, "%lf", &Phi[row_indx][col_indx]);
}
    fscanf(buffer, "%d", &Bsize[row_indx]);
    fscanf(lam, "%lf", &Lam[row_indx]);
   }//2

 fscanf(control, "%d", &rt_policy);
 fscanf(control, "%d",&seq_policy);
 fclose(fpr);
 fclose(mu);
 fclose(respri);
 fclose(cuspri);
 fclose(phi);
 fclose(buffer);
 fclose(lam);
 fclose(control);

 printf("Mu Matrix\n");
 for (row_indx=0;row_indx<num_cus; row_indx++)
   {
    for (col_indx=0; col_indx<num_res; col_indx++)
{
 printf("%lf\t", Mu[row_indx][col_indx]);
}
    printf("\n");
   }

 printf("Respri Matrix\n");
 for (row_indx=0;row_indx<num_cus; row_indx++)
   {
    for (col_indx=0; col_indx<num_res; col_indx++)
{
 printf("%d\t", Respri[row_indx][col_indx]);
}
    printf("\n");
   }

 printf("CusPri Matrix\n");
 for (row_indx=0;row_indx<num_cus; row_indx++)
   {
    for (col_indx=0; col_indx<num_res; col_indx++)
{
 printf("%d\t", Cuspri[col_indx][row_indx]);
}
    printf("\n");
   }

 printf("Phi Matrix\n");
 for (row_indx=0;row_indx<num_cus; row_indx++)
   {
    for (col_indx=0; col_indx<num_cus; col_indx++)
{
 printf("%lf\t", Phi[row_indx][col_indx]);
}
    printf("\n");
   }

 printf("Buffer Matrix\n");
 for (row_indx=0;row_indx<num_cus; row_indx++)
   {
    printf("%d\n", Bsize[row_indx]);
   }

 printf("Lam matrix\n");

 for (row_indx=0; row_indx<num_cus; row_indx++)
   {
    printf("%lf\n", Lam[row_indx]);
   }


 // All structural parameter inputs have been read

 //Transforming variables and the network.
 int * cus_res_match;
 cus_res_match=(int *) calloc (num_res, sizeof(int));
 for (i=0; i<num_res; i++)
   {
    cus_res_match[i]=0;
   }

 num_unm_cus=0;

 for (i=0; i<num_cus; i++)
   {//2
    flag=0;
    for (j=0; j<num_res; j++)
{
 if ((M[i+1][j]==1)&&(cus_res_match[j]==0)&&(flag==0))
   {
    flag=1;
    cus_res_match[j]=1;
   }
}
    if (flag==0)
{
 num_unm_cus++;
}
   }//2
 printf("Num of Unassigned Customers=%d\n",
num_unm_cus);

 for (i=0; i<num_res; i++)
   {//2
    for (j=0; j<num_cus; j++)
{
 if (M[j+1][i]==1)
   {
    num_aux_res++;
   }
}
   }//2

 num_st_var=num_aux_res+num_unm_cus;

 //creating aux matrices, AM, AQ, APhi, AMu, ARespri,
ACuspri, ALam, AB;

 start=clock();

 int **AQ, *AQ_temp, **ARespri, *ARespri_temp,
**ACuspri, *ACuspri_temp, *cus_st_var_ind,
*aux_res_cus_st_var_ind, *num_cus_sys,
*num_max_st_var_in_res_grp;
```

```
  int *AB, *cardQ, *cardQT, *res_grp_id, *res_grp_flag,
stat_var, *ui, *vi, *omega, *bi, *cus_id, *res_grp_bsy_flag,
*res_grp_excl_bsy_flag;
  int st_sp_size, st_sp_width;

  double *AMu, *ALam, **APhi, *APhi_temp,
*APhi_row_sum, *LQF;

  AQ_temp=(int *) calloc (num_st_var*num_st_var, sizeof
(int));
  AQ=(int **) calloc (num_st_var, sizeof(int *));

  ARespri_temp=(int *) calloc (num_st_var*num_st_var,
sizeof (int));
  ARespri=(int **) calloc (num_st_var, sizeof(int *));

  ACuspri_temp=(int *) calloc (num_st_var*num_st_var,
sizeof (int));
  ACuspri=(int **) calloc (num_st_var, sizeof(int *));

  AB=(int *) calloc (num_st_var,  sizeof (int));

  ACuspri=(int **) calloc (num_st_var, sizeof(int *));

  AMu=(double *) calloc (num_aux_res, sizeof(double));

  APhi_temp=(double *) calloc (num_st_var*num_st_var,
sizeof (double));
  APhi=(double **) calloc (num_st_var, sizeof(double *));

  APhi_row_sum=(double *) calloc (num_st_var,
sizeof(double *));

  ALam=(double *) calloc (num_st_var, sizeof(double));

  cardQT=(int *) calloc (num_res, sizeof(int));

  res_grp_id=(int *) calloc (num_aux_res, sizeof(int));

  res_grp_flag=(int *) calloc (num_res, sizeof(int));

  cardQ=(int *) calloc (num_cus, sizeof(int));

  vi=(int *) calloc (num_st_var, sizeof(int));

  ui=(int *) calloc (2*num_st_var, sizeof(int));

  omega =(int *) calloc (st_sp_size, sizeof(int));

  bi=(int *) calloc (num_st_var, sizeof(int));

  cus_id=(int *) calloc (num_st_var, sizeof(int));

  cus_st_var_ind= (int *) calloc (num_cus, sizeof(int));

  aux_res_cus_st_var_ind= (int *) calloc (num_st_var,
sizeof(int));

  res_grp_bsy_flag=(int *) calloc (2*num_res, sizeof(int));

  res_grp_excl_bsy_flag=(int *) calloc (2*num_st_var,
sizeof(int));

  LQF= (double *) calloc (num_aux_res, sizeof(int));

  num_cus_sys= (int *) calloc (num_cus, sizeof(int));

  num_max_st_var_in_res_grp= (int *) calloc (num_res,
sizeof(int));

  int condition[25];
  int r, t;

  for (i=0; i<25; i++)
   {
    condition[i]=0;
   }

  printf("Got Here 1\n");
  end=clock();

  if (ACuspri_temp==NULL || ACuspri==NULL)
   {
    printf ("error in ACuspri calloc allocation\n");
   }
  else
   {
    printf("ACuspri_calloc completed in %d%
milliseconds\n", (int) ((end-
start)*1E3/CLOCKS_PER_SEC));
   }

 //make all the single column matrices point to the beginning
of each row

  for (i=0; i<num_st_var; i++)
   {
    AQ[i]=AQ_temp+i*num_st_var;
    ARespri[i]=ARespri_temp+i*num_st_var;
    AMu[i]=0.0;
    APhi[i]=APhi_temp+i*num_st_var;
   }

  for (i=0; i<num_st_var; i++)
   {
    ACuspri[i]=ACuspri_temp+i*num_st_var;
    vi[i]=0;
    omega[i]=0;
    ui[i]=0;
    ui[i+num_st_var]=0;
    APhi_row_sum[i]=0.0;
   }

  for (i=0; i<num_res;i++)
   {
    cardQT[i]=0;
    res_grp_flag[i]=0;
    res_grp_bsy_flag[i]=0;
    res_grp_bsy_flag[i+num_res]=0;
    num_max_st_var_in_res_grp[i]=0;
   }

  for (i=0; i<num_st_var; i++)
   {
    for (j=0; j<num_st_var;j++)
{
 AQ[i][j]=0;
 APhi[i][j]=0.0;
}
    res_grp_excl_bsy_flag[i]=0;
    res_grp_excl_bsy_flag[i+num_st_var]=0;
    aux_res_cus_st_var_ind[i]=1000;
    cus_id[i]=1000;
   }

  for (i=0; i<num_cus; i++)
   {
    cus_st_var_ind[i]=0;
    num_cus_sys[i]=0;
   }
 //All the matrices have been created and initialized by
calloc. Now we populate the matrices.

 //1. Here we count the number of customers associated with
any resource, define the set of aux. servers, and redefine the
routing flexibilities.


  for (i=0; i<num_aux_res; i++)
    {
     omega[i]=1;
     bi[i]=1;
     LQF[i]=0.0;
     res_grp_id[i]=1000;
    }

  printf("Got Here 3\n");

  // Next, we assign the customers to the servers.

  int temp_int,temp_int1, unm_cus_indx, temp_count;
  double temp_double, temp_double1;

  unm_cus_indx=num_aux_res;

  int *cus_assn_flag;
  cus_assn_flag= (int *) calloc (num_cus, sizeof(int));

  read_cnt=0;

  for (i=0; i<num_cus; i++)// We are trying to assign
customers to auxiliary resource
    {//2
     cus_assn_flag[i]=0;
     //Customer i has not been assigned yet...
     for (j=0; j<num_res;j++)//Looping through  resources
{
 if (M[i+1][j]==1)
   {
    cardQT[j]++;
    cardQ[i]++;
    cus_id[read_cnt]=i;
    res_grp_id[read_cnt]=j;
    if ((cus_assn_flag[i]==0)&&(res_grp_bsy_flag[j]==0))
{
 cus_assn_flag[i]=1;
 res_grp_bsy_flag[j]=1;
 vi[read_cnt]=1;
 cus_st_var_ind[i]=read_cnt;
 bi[read_cnt]=bi[read_cnt]+Bsize[i];
 //printf("Customer %d Assigned to Resource %d\n", i, j);
 //printf("bi[%d]=%d\n", read_cnt, bi[read_cnt]);
}
    read_cnt++;
   }
}
    if (cus_assn_flag[i]==0)
{
 bi[unm_cus_indx]=Bsize[i];
 omega[unm_cus_indx]=0;
 vi[unm_cus_indx]=1;
 unm_cus_indx++;
 cus_id[read_cnt]=i;
 cus_st_var_ind[i]=read_cnt;
}
    }//2

  for (i=0; i<num_cus; i++) printf("cardQ[%d]=%d\n",i,
cardQ[i]);

  for (i=0; i<num_res; i++) printf("cardQT[%d]=%d\n",i,
cardQT[i]);

  printf("number of state variables=%d\n", num_st_var);

  // We have thus far defined the state variables and their
bounds. Next, we re-define the routing, prioroty, and
processing rate matrices.

  int temp_res_id, temp_cus_id, temp_res_id1, temp_cus_id1,
temp_st_var_id, temp_st_var_id1;

  for (i=0; i<num_st_var; i++)
    {//2
     if (vi[i]==1)
{
 temp_cus_id=cus_id[i];
 ALam[i]=Lam[temp_cus_id];
 for (j=0; j<num_st_var; j++)
   {
    if (vi[j]==1)
{
 temp_int=cus_id[j];
 APhi[i][j]=Phi[temp_cus_id][temp_int];
 aux_res_cus_st_var_ind[j]=j;
}
    if (omega[j]==1)
{
 temp_res_id=res_grp_id[j];
 if
((M[temp_cus_id+1][temp_res_id]==1)&&(temp_cus_id==c
us_id[j]))
    {
     AQ[i][j]=1;
     AMu[j]=Mu[temp_cus_id][temp_res_id];
     ARespri[i][j]=Respri[temp_cus_id][temp_res_id];
     ACuspri[j][i]=Cuspri[temp_res_id][temp_cus_id];
     aux_res_cus_st_var_ind[j]=i;
    }
}
    }
}
    }//2

  for (i=0;i<num_st_var; i++)
    {
     for (j=0; j<num_st_var; j++);
     {
APhi_row_sum[i]=APhi_row_sum[i]+APhi[i][j];
     }
    }

  for (i=0; i<num_st_var; i++)
    {
     printf("vi[%d]=%d, omega[%d]=%d,
cus_id[%d]=%d\n",i, vi[i],i, omega[i],i,cus_id[i]);
     if (omega[i]==1)
{
 printf("res_gpr_id[%d]=%d\n",i,res_grp_id[i]);
}

printf("aux_res_cus_st_var_ind[%d]=%d\n",i,aux_res_cus_st
_var_ind[i]);
    }
  for (i=0;i<num_cus;i++)
    {
     printf("cus_st_var_ind[%d]=%d\n",i, cus_st_var_ind[i]);
    }
  printf("All the parameters in the auxiliary network have
been defined\n");
```

```
 printf("This is the routing matrix in the auxiliary
network\n");

 for (i=0; i<num_st_var; i++)
   {
    for (j=0; j<num_st_var; j++)
{
 printf ("%d\t", AQ[i][j]);
}
    printf("%lf\t", AMu[i]);
    printf("%d\n",bi[i]);
   }
 //Now I am ready to define the state space.

 int p,b, stat_indxa, stat_indxd, stat_indxad, pow_val;

 p=M_row -1;
 b=M[0][2];

 int **stat_mat, *stat_temp;

 //st_sp_size=pow((b+1),m);

 st_sp_size=1;

 for (i=0;i<num_st_var;i++)
   {
    st_sp_size=st_sp_size*(bi[i]+1);
   }

 printf("State Space Size (Raw)=%d\n",st_sp_size);

 st_sp_width=num_st_var+4;
 start=clock();
 stat_temp=(int *) calloc (st_sp_size*st_sp_width, sizeof
(int));
 stat_mat=(int **) calloc (st_sp_size, sizeof(int *));

 end=clock();

 if (stat_temp==NULL || stat_mat==NULL)
   {
    printf ("error in stat_mat calloc allocation\n");
   }
 else
   {
    printf("stat_calloc completed in %d% milliseconds\n",
(int) ((end-start)*1E3/CLOCKS_PER_SEC));
   }

 //make stat_mat point to the beginning of each row

 for (i=0; i<st_sp_size; i++)
stat_mat[i]=stat_temp+i*st_sp_width;

 for (i=0;i<st_sp_size;i++)
   {
    stat_mat[i][0]=1;
    stat_mat[i][1]=i+1;
    //This number represents the raw state sequence number
    stat_mat[i][st_sp_width-1]=0;
    //this represents the real state sequence number after
removing invalid states
    stat_mat[i][st_sp_width-2]=0;
    //this represents flag used to prevent duplicate printing of
a state into Stat_mat
   }


 //Now we have to generate the state space
 int cycle,n,l,o;
 for (i=0;i<num_st_var;i++)
   {//2
    k=-1;
    l=0;
    while (l < st_sp_size)
{
 k=k+1;
 cycle=1;
 for (o=i+1; o<num_st_var; o++)
   {
    cycle=cycle*(bi[o]+1);
   }
 if (k>bi[i])
   {
    k=0;
   }
 n=1;
 while (n <= cycle)
   {
    n=n+1;
    stat_mat[l][i+2]=k;
    l=l+1;
   }
}
   }//2

 //Next we have to disqualify some states.
 //We eliminate the invalid states.
 //1.A state is invalid if for a resource group, if any two state
variables are at a maximum.
 //2.A state is invalid if there is a non-empty queue and if all
of the capable resources are idle.

 int num_bsy_res, num_pure_bsy_res, num_max_out_st_var;
 int real_size;
 real_size=0;

 for (i=0;i<st_sp_size;i++)
   {
    stat_mat[i][0]=1;
    for (j=0;j<num_res;j++)
{
 num_max_st_var_in_res_grp[j]=0;
}
    for (j=0;j<num_st_var;j++)
{
 res_grp_excl_bsy_flag[j]=0;
}
    for (z=0; z<num_aux_res; z++)
{
 temp_res_id=res_grp_id[z];
 res_grp_bsy_flag[temp_res_id]=0;
 res_grp_excl_bsy_flag[z]=0;
 temp_int=0;
 for (l=0; l<num_aux_res; l++)
   {
    if (res_grp_id[l]==res_grp_id[z])
{
 temp_int=temp_int+stat_mat[i][l+2];
}
   }
 if (temp_int>0)
   {
    res_grp_bsy_flag[temp_res_id]=1;
   }
 temp_int=temp_int-stat_mat[i][z+2];
 if (temp_int>0)
   {
    res_grp_excl_bsy_flag[z]=1;
   }
}

    j=0;
    while ((stat_mat[i][0]==1)&&(j<num_aux_res))
{
 temp_res_id=res_grp_id[j];
 if (stat_mat[i][j+2]==bi[j])
   {

num_max_st_var_in_res_grp[temp_res_id]=num_max_st_va
r_in_res_grp[temp_res_id]+1;
    if (num_max_st_var_in_res_grp[temp_res_id]>1)
{
 stat_mat[i][0]=0;
}
   }
 j++;
}
    j=0;
    while ((stat_mat[i][0]==1)&&(j<num_cus))
{
 temp_cus_id=cus_st_var_ind[j];
 if (omega[temp_cus_id]==1)
   {
    if (stat_mat[i][temp_cus_id+2]>1-
res_grp_excl_bsy_flag[temp_cus_id])
{
 k=0;
 while ((stat_mat[i][0]==1)&&(k<num_aux_res))
   {
    if (AQ[temp_cus_id][k]==1)
{
 temp_res_id=res_grp_id[k];
 if (res_grp_bsy_flag[temp_res_id]==0) stat_mat[i][0]=0;
}
    k++;
   }
}
   }
 else if (omega[temp_cus_id]==0)
   {
    if (stat_mat[i][temp_cus_id+2]>0)
{
 k=0;
 while ((stat_mat[i][0]==1)&&(k<num_aux_res))
   {
    if (AQ[temp_cus_id][k]==1)
{
 temp_res_id=res_grp_id[k];
 if (res_grp_bsy_flag[temp_res_id]==0) stat_mat[i][0]=0;
}
    k++;
   }
}
   }
 j++;
}
   }

 real_size=0;
 for (i=0;i<st_sp_size;i++)
   {
    if (stat_mat[i][0]==1)
{
 stat_mat[i][st_sp_width-1]=real_size;
 real_size++;
}
   }
```

```
 printf("REAL SIZE= %d \n", real_size);

 for (i=0;i<st_sp_size;i++)
   {
    for (j=0; j<st_sp_width; j++)
{
 fprintf(stat_test, "%d\t",stat_mat[i][j]);
 fflush(stat_test);
}
    fprintf(stat_test,"\n");
    fflush(stat_test);
   }

 fclose(stat_test);
 system("cp /tmp/stat_test stat_test");

 int tran_size, tran_indx1, tran_indx2;
 tran_size=real_size+1;
 printf("tran_size=%d\n", tran_size);

 double *row_sum, *b_mat, *tran_mat_val,
*tran_realloc_flag3;
 int *tran_row_val, *tran_col_val, tran_mem_cnt,
tran_entry_cnt, *tran_realloc_flag1, *tran_realloc_flag2;
 tran_mem_cnt=50;
 tran_entry_cnt=0;


 start=clock();

 row_sum=(double *) calloc (real_size, sizeof (double));
 b_mat=(double *) calloc (real_size, sizeof (double));
 tran_row_val=(int *) calloc (tran_mem_cnt, sizeof (int));
 tran_col_val=(int *) calloc (tran_mem_cnt, sizeof (int));
 tran_mat_val=(double *) calloc (tran_mem_cnt, sizeof
(double));

 end=clock();

 if (row_sum==NULL || b_mat==NULL)
   {
    printf ("error in row_sum or b_mat calloc allocation\n");
   }
 else
   {
    printf("row_sum and b_mat calloc completed in %d%
milliseconds\n", (int) ((end-
start)*1E3/CLOCKS_PER_SEC));
   }

 for (i=0; i<real_size; i++)
   {
    row_sum[i]=0.0;
    b_mat[i]=0.0;
   }

 b_mat[real_size-1]=1.0;

 double count[150],temp[150], beta[150],gam[150],
theta[150];
 int eps[150];

 for (i=0;i<150;i++)
   {//1
    count[i]=0.0;
    temp[i]=0.0;
    eps[i]=0;
    beta[i]=0.0;
    gam[i]=0.0;
   }//1
```

```
  double tran_val,sum_col, indicator[30];
  tran_val=0.0;
  int rowi,coli;
  int col_cnt;
  int flag1, flag2;
  col_cnt=0;
  int num_values;
  num_values=0;
  int stat_indxa_flag, stat_indxd_flag, stat_indxad_flag;


  for (i=0;i<st_sp_size;i++)
    {//2
     if (stat_mat[i][0] !=0)
{//3
  //printf("i%d\n",i);

  fprintf(Stat_mat, "%d\t", i);
  fflush(Stat_mat);
  for (z=0;z<=st_sp_width-1;z++)
    {
     fprintf(Stat_mat, "%d\t", stat_mat[i][z]);
     fflush(Stat_mat);
    }
  fprintf(Stat_mat, "\n");
  fflush(Stat_mat);
}
    }
  fclose(Stat_mat);

  for (i=0;i<st_sp_size;i++)
    {//2
     if (stat_mat[i][0] !=0)
{//3

  sum_col=0.0;
  col_cnt=0;

  //First I define the Nia and Nid states and their transition
rates.
  no_arr_flag=0;
  no_dep_flag=0;
  for (k=0;k<num_st_var;k++)
    {
     stat_indxa=0;
     stat_indxd=0;
     stat_indxad=0;
     stat_indxa_flag=1;
     stat_indxd_flag=1;

     cycle=1;
     for (l=num_st_var-1;l>=0;l--)
{
 if (l!=num_st_var-1)
   {
    cycle=cycle*(bi[l+1]+1);
   }
 if (l!=k)
   {
    stat_indxa=stat_indxa+cycle*stat_mat[i][l+2];
    stat_indxd=stat_indxd+cycle*stat_mat[i][l+2];
   }
 if (l==k)
   {
    if (stat_mat[i][k+2]>0)
{
 stat_indxa=stat_indxa+cycle*(stat_mat[i][k+2]-1);
}
    else if (stat_mat[i][k+2]==0)
{
 stat_indxa=stat_indxa+cycle*stat_mat[i][k+2];
 stat_indxa_flag=0;
}
    if (stat_mat[i][k+2]<bi[k])
{
 stat_indxd=stat_indxd+cycle*(stat_mat[i][k+2]+1);
}
    else if (stat_mat[i][k+2]==bi[k])
{
 stat_indxd=stat_indxd+cycle*stat_mat[i][k+2];
 stat_indxd_flag=0;
}
   }
}


    //Next, we have to determine which resource groups are
busy for the arrival and departure states.

    for (z=0; z<num_res; z++)
{
 res_grp_bsy_flag[z]=0;
 res_grp_bsy_flag[z+num_res]=0;
}

    for (z=0; z<num_aux_res; z++)
{
 temp_res_id=res_grp_id[z];
 res_grp_bsy_flag[temp_res_id]=0;
 res_grp_bsy_flag[temp_res_id+num_res]=0;
 temp_int=0;
 temp_int1=0;
 for (l=0; l<num_aux_res; l++)
   {
    if (res_grp_id[l]==res_grp_id[z])
{
 temp_int=temp_int+stat_mat[stat_indxa][l+2];
 temp_int1=temp_int1+stat_mat[stat_indxd][l+2];
}
   }
 if (temp_int>0)
   {
    res_grp_bsy_flag[temp_res_id]=1;
   }
 if (temp_int1>0)
   {
    res_grp_bsy_flag[temp_res_id + num_res]=1;
   }
 temp_int=temp_int-stat_mat[stat_indxa][z+2];
 temp_int1=temp_int1-stat_mat[stat_indxd][z+2];
 if (temp_int>0)
   {
    res_grp_excl_bsy_flag[z]=1;
   }
 if (temp_int1>0)
   {
    res_grp_excl_bsy_flag[z+num_st_var]=1;
   }
}
    if (num_st_var>num_aux_res)
{
 for (z=num_aux_res;z<num_st_var;z++)
   {
    res_grp_excl_bsy_flag[z]=0;
    res_grp_excl_bsy_flag[z+num_st_var]=0;
   }
}

    for (l=0; l<num_st_var; l++)
```

```
{
 ui[l]=0;
 ui[l+num_st_var]=0;
}
    for (l=0; l<num_st_var; l++)
{//5
 if ((vi[l]==1)&&(omega[l]==1)&&(wip_policy==0))
   {
    if (res_grp_excl_bsy_flag[l]==1)
{
 ui[l]=bi[l]-1-stat_mat[stat_indxa][l+2];
}
    if (res_grp_excl_bsy_flag[l+num_st_var]==1)
{
 ui[l+num_st_var]=bi[l]-stat_mat[stat_indxd][l+2];
}
    if (res_grp_excl_bsy_flag[l]==0)
{
 ui[l]=bi[l]-stat_mat[stat_indxa][l+2];
}
    if (res_grp_excl_bsy_flag[l+num_st_var]==0)
{
 ui[l+num_st_var]=bi[l]-stat_mat[stat_indxd][l+2];
}
   }
 else
   {
    ui[l]=bi[l]-stat_mat[stat_indxa][l+2];
    ui[l+num_st_var]=bi[l]-stat_mat[stat_indxd][l+2];
   }
}//5

    //Next we evaluate the indicator function gamma[k].
Reserved for arrival states.
    //This indicates whether resource k (if k<num_aux_res)
can accept a customer of type Tbar_k (Equation 18)

    if (omega[k]==1)
{
 temp_double=1.0;
 gam[k]=1.0; // Initially we assume that resource k can
accept an incoming customer of type Tbar_k
 temp_cus_id=aux_res_cus_st_var_ind[k];
 for (z=0; z<num_aux_res; z++)
   {
    temp_res_id=res_grp_id[z];
    if (res_grp_bsy_flag[temp_res_id]==0)
{
 if (ARespri[temp_cus_id][k]>ARespri[temp_cus_id][z])
   {
    gam[k]=0.0;
   }
 else if
((ARespri[temp_cus_id][k]==ARespri[temp_cus_id][z])&&(
k!=z))
   {
    temp_double=temp_double+1.0;
   }
 if (temp_double>1.0)
   {
    gam[k]=gam[k]/temp_double;
   }
}
   }
}

    //printf("gam[] has been defined for the arrival and
departure states for variable k\n",k);
    //Next, we evaluate the indicator function theta[r]
    for (z=0; z<num_st_var; z++)
{
 theta[z]=1.0;
}

    temp_cus_id=cus_id[k];

    if (vi[k]==1)
{//5
 for (r=0; r<num_aux_res; r++)
   {
    temp_double=1.0;
    if (AQ[k][r]==1)
{
 temp_res_id=res_grp_id[r];
 for (t=0; t<num_aux_res; t++)
   {
    if (res_grp_id[t]==res_grp_id[r])
{
 temp_cus_id=aux_res_cus_st_var_ind[t];
 if
((omega[temp_cus_id]==1)&&(stat_mat[stat_indxd][temp_c
us_id+2]>1-
res_grp_excl_bsy_flag[temp_cus_id+num_st_var]))
   {
    if (ACuspri[temp_cus_id][t]<ACuspri[k][r])
{
 theta[r]=0.0;
}
    else if
((ACuspri[temp_cus_id][t]==ACuspri[k][r])&&(k!=temp_cu
s_id))
{
 temp_double=temp_double+1.0;
}
   }
 if
((omega[temp_cus_id]==0)&&(stat_mat[stat_indxd][temp_c
us_id+2]>0))
   {
    if (ACuspri[temp_cus_id][t]<ACuspri[k][r])
{
 theta[r]=0;
}
    else if
((ACuspri[temp_cus_id][t]==ACuspri[k][r])&&(k!=temp_cu
s_id))
{
 temp_double=temp_double+1.0;
}
   }
}
   }
}
    if (temp_double>1.0)
{
 theta[r]=theta[r]/temp_double;
}
   }
}//5

    //condition[1]=1 if ALL of the candidate resource groups
for a customer are busy. Reserved for arrival states.

    if (vi[k]==1)
{//5
 condition[1]=1;
 for (l=0; l<num_aux_res; l++)
   {
    if (AQ[k][l]==1)
{
 temp_res_id=res_grp_id[l];
 if (res_grp_bsy_flag[l]==0)
   {
    condition[1]=0;
   }
}
   }
}//5

    //condition[2]=1 if ALL of the candidate resource groups
for a customer are busy. Reserved for departure states.

    if (vi[k]==1)
{//5
 condition[2]=1;
 for (l=0; l<num_aux_res; l++)
   {
    if (AQ[k][l]==1)
{
 temp_res_id=res_grp_id[l];
 if (res_grp_bsy_flag[l+num_res]==0)
   {
    condition[2]=0;
   }
}
   }
}//5

    //condition[0]=1 if the queue for customer k is at zero, or
non-empty, but where all of the candidate resource groups
are busy
    //In other words, an event with arrival of a customer of
this type, will cause a queue formation.

    if (vi[k]==1)
{
 condition[0]=0;
 if ((omega[k]==1)&&(stat_mat[stat_indxa][k+2]>=1-
res_grp_excl_bsy_flag[k])&&(condition[1]==1))
   {
    condition[0]=1;
   }
 if
((omega[k]==0)&&(stat_mat[stat_indxa][k+2]>=0)&&(cond
ition[1]==1))
   {
    condition[0]=1;
   }
}


    //condition[3]=1 if, for all the candidate customer classes
for each resource in the resource group for [k],
    //there are no customers in queue.
    //condition[3] is reserved as a flag for the arrival states.

    if (omega[k]==1)
{//5
 condition[3]=1;
 for (l=0; l<num_st_var; l++)
   {
    if (res_grp_id[k]==res_grp_id[l])
{
 temp_cus_id=aux_res_cus_st_var_ind[l];
 if
((omega[temp_cus_id]==1)&&(stat_mat[stat_indxa][temp_c
us_id+2]>1-res_grp_excl_bsy_flag[temp_cus_id]))
   {
    condition[3]=0;
   }
 else if
((omega[temp_cus_id]==0)&&(stat_mat[stat_indxa][temp_c
us_id+2]>0))
   {
    condition[3]=0;
   }
}
   }
}//5

    //condition[4]=1 if, for all the candidate customer classes
for each resource in the resource group for [k].
    //there are no customers in queue.
    //While condition[4] is similar to condition[3], it reserved
as a flag for the departure states

    if (omega[k]==1)
{
 condition[4]=1;
 for (l=0; l<num_st_var; l++)
   {
    if (res_grp_id[k]==res_grp_id[l])
{
 temp_cus_id=aux_res_cus_st_var_ind[l];
 if
((omega[temp_cus_id]==1)&&(stat_mat[stat_indxd][temp_c
us_id+2]>1-
res_grp_excl_bsy_flag[temp_cus_id+num_st_var]))
   {
    condition[4]=0;
   }
 else if
((omega[temp_cus_id]==0)&&(stat_mat[stat_indxd][temp_c
us_id+2]>0))
   {
    condition[4]=0;
   }
}
   }
}

    //condition[5]=1 if for vi[k]=1, the queue for that
customer is already non-empty.
    //In the event of a departure from this state, one of these
queued customers is a candidate for processing at
    //one of the valid resource groups. Condition reserved for
departure states.

    if (vi[k]==1)
{
 condition[5]=0;
 if
((condition[2]==1)&&(omega[k]==1)&&(stat_mat[stat_indx
d][k+2]>1-res_grp_excl_bsy_flag[k+num_st_var]))
   {
    condition[5]=1;
   }
 else if
((condition[2]==1)&&(omega[k]==1)&&(stat_mat[stat_indx
d][k+2]>0))
   {
    condition[5]=1;
   }
}



    //eps[r]=1 if, the queue of job class r is one of the LQF[r]
longest in the set of resources {r: AQ[k][r]=1.}
```

```
    if ((vi[k]==1)&&(condition[5]==1))
{
 for (r=0; r<num_aux_res; r++)
   {
    eps[r]=0;
    LQF[r]=0.0;
    temp_res_id=res_grp_id[r];
    if (AQ[k][r]==1)
{
 eps[r]=1;
 LQF[r]=0.0;
 temp_double=1.0;
 for (t=0; t<num_aux_res; t++)
   {
    if (eps[r]==1)
{
 if (res_grp_id[t]==res_grp_id[r])
   {
    temp_cus_id=aux_res_cus_st_var_ind[t];

temp_int=stat_mat[stat_indxd][temp_cus_id+2]+omega[temp
_cus_id]*res_grp_excl_bsy_flag[temp_cus_id+num_st_var];

temp_int1=stat_mat[stat_indxd][k+2]+omega[k]*res_grp_ex
cl_bsy_flag[k+num_st_var];
    if (temp_int>temp_int1)
{
 eps[r]=0;
}
    else if ((temp_int==temp_int1)&&(k!=temp_cus_id))
{
 temp_double=temp_double+1.0;
}
   }
}
   }
}
    if (eps[r]==1) LQF[r]=temp_double;
   }
}

    //Condition[6]=1 if, for a given class, a departure from
the system (or network) is possible for that class
    //In other words, the customer leaves the system with
probability strictly less than 1.

    if (APhi_row_sum[k]<1.0)
{
 condition[6]=1;
}

    //printf("The state classification conditions have been
defined for the arrival and departure states for variable
k\n",k);
    //Now we write the transition rates for the various control
policies and various state space conditions

    //Define Arrival Rates first...

    //Class A states
    if
((stat_indxa_flag==1)&&(condition[0]==1)&&(ui[k]>=1)&
&(ALam[k]>0.0)&&(vi[k]==1))
{//5
 tran_val=0.0; //reset this variable
 if ((rt_policy==0)||(rt_policy==1))//SP Routing Policy or
Random Routing Policy
   {
    tran_val=ALam[k];
   }
}//5
    else if ((omega[k]==1)&&(stat_indxa_flag==1))
{
 temp_res_id=res_grp_id[k];
 if
((res_grp_bsy_flag[temp_res_id]==0)&&(condition[3]==1)&
&(ui[k]>=1))
   {
    temp_cus_id=aux_res_cus_st_var_ind[k];
    tran_val=0.0; //reset this variable
    if (rt_policy==0)//Strict Priority Routing
{
 tran_val=ALam[temp_cus_id]*gam[k];
}
    if (rt_policy==1)//Random Routing
{
 temp_double=0.0;
 for (t=0; t<num_aux_res; t++)
   {
    if (AQ[temp_cus_id][t]==1)
{
 temp_res_id1=res_grp_id[t];
 temp_double=temp_double+1-
res_grp_bsy_flag[temp_res_id1];
}
   }
 if (temp_double>0.0)
   {
    tran_val=ALam[temp_cus_id]/temp_double;
   }
}
   }
}
    if ((tran_val>0.0)&&(stat_mat[stat_indxa][0]==1))
{
 tran_indx1=stat_mat[stat_indxa][st_sp_width-1];
 tran_indx2=stat_mat[i][st_sp_width-1];
 row_sum[tran_indx1]=row_sum[tran_indx1]+tran_val;
 check_int=real_size-1;
 if (tran_indx2!=check_int)
   {
    tran_row_val[num_values]=tran_indx2;
    tran_col_val[num_values]=tran_indx1;
    tran_mat_val[num_values]=tran_val;

    if ((tran_realloc_flag1 = realloc(tran_row_val, sizeof(int)
* (tran_mem_cnt + 1))) == NULL) {
printf("ERROR: realloc1 failed\n");
    }
    tran_row_val = tran_realloc_flag1;

    if ((tran_realloc_flag2 = realloc(tran_col_val, sizeof(int)
* (tran_mem_cnt + 1))) == NULL) {
printf("ERROR: realloc2 failed\n");
    }
    tran_col_val=tran_realloc_flag2;

    if ((tran_realloc_flag3 = realloc(tran_mat_val,
sizeof(double) * (tran_mem_cnt + 1))) == NULL) {
printf("ERROR: realloc1 failed\n");
    }
    tran_mat_val=tran_realloc_flag3;

    tran_mem_cnt++;

    num_values++;
   }
 tran_val=0.0;
}
```

```
    //Arrival Rates Have Been Defined. Now we define the
departure rates

    if
((stat_indxd_flag==1)&&(stat_mat[stat_indxd][k+2]==1)&&
(res_grp_excl_bsy_flag[k]==0)&&(condition[4]==1)&&(om
ega[k]==1))
{
 tran_val=0.0; //reset this variable
 if ((seq_policy==0)||(seq_policy==2))
   {
    tran_val=AMu[k]*(1.0-APhi_row_sum[k]);
   }
}
    if
((stat_indxd_flag==1)&&(condition[5]==1)&&(vi[k]==1))
{
 tran_val=0.0; //reset this variable
 if (seq_policy==0)//Strict Priority Sequencing
   {
    for (r=0; r<num_st_var; r++)
{
 if (AQ[k][r]==1)
   {
    tran_val=tran_val+AMu[r]*theta[r]*(1.0-
APhi_row_sum[r]);
   }
}
   }
 if (seq_policy==2)//LQF Policy
   {
    for (r=0; r<num_st_var; r++)
{
 if (AQ[k][r]==1)
   {
    if (LQF[r]>0.0)
{
 temp_double=eps[r];
 tran_val=tran_val+AMu[r]*(1.0-
APhi_row_sum[r])*temp_double/LQF[r];
}
   }
}
   }
}
    if ((tran_val>0.0)&&(stat_mat[stat_indxd][0]==1))
{
 tran_indx1=stat_mat[stat_indxd][st_sp_width-1];
 tran_indx2=stat_mat[i][st_sp_width-1];

 row_sum[tran_indx1]=row_sum[tran_indx1]+tran_val;

 check_int=real_size-1;
 if (tran_indx2!=check_int)
   {

    tran_row_val[num_values]=tran_indx2;
    tran_col_val[num_values]=tran_indx1;
    tran_mat_val[num_values]=tran_val;

    if ((tran_realloc_flag1 = realloc(tran_row_val, sizeof(int)
* (tran_mem_cnt + 1))) == NULL) {
printf("ERROR: realloc1 failed\n");
    }
    tran_row_val = tran_realloc_flag1;

    if ((tran_realloc_flag2 = realloc(tran_col_val, sizeof(int)
* (tran_mem_cnt + 1))) == NULL) {
printf("ERROR: realloc2 failed\n");
    }
    tran_col_val=tran_realloc_flag2;

    if ((tran_realloc_flag3 = realloc(tran_mat_val,
sizeof(double) * (tran_mem_cnt + 1))) == NULL) {
printf("ERROR: realloc1 failed\n");
    }
    tran_mat_val=tran_realloc_flag3;

    tran_mem_cnt++;

    num_values++;
    tran_val=0.0;
   }
 tran_val=0.0;
}


    // printf("The transition rates have been defined for the
arrival and departure states only for variable k\n",k);

    //Now, we look at the transitions that are caused by inter-
class exchanges of customers.

    for (k_bar=0; k_bar<num_st_var; k_bar++)
{
 if (k!=k_bar)
   {
    stat_indxad=0;
    stat_indxad_flag=1;
    cycle=1;
    for (l=num_st_var-1;l>=0;l--)
{
 if (l!=num_st_var-1)
   {
    cycle=cycle*(bi[l+1]+1);
   }
 if ((l!=k)&&(l!=k_bar))
   {
    stat_indxad=stat_indxad+cycle*stat_mat[i][l+2];
   }
 if (l==k)
   {
    if (stat_mat[i][k+2]>0)
{
 stat_indxad=stat_indxad+cycle*(stat_mat[i][l+2]-1);
}
    else if (stat_mat[i][k]==0)
{
 stat_indxad=stat_indxad+cycle*stat_mat[i][l+2];
 stat_indxad_flag=0;
}
   }
 if (l==k_bar)
   {
    if (stat_mat[i][k_bar+2]<bi[k_bar])
{
 stat_indxad=stat_indxad+cycle*(stat_mat[i][l+2]+1);
}
    else if (stat_mat[i][k_bar+2]==bi[k_bar])
{
 stat_indxad=stat_indxad+cycle*stat_mat[i][l+2];
 stat_indxad_flag=0;
}
   }
}
   }
 if (k==k_bar)
   {
    stat_indxad=i;
    }

  //Next, we have to determine which resource groups are
busy for the arrival and departure states.
  for (z=0; z<num_res; z++)
    {
     res_grp_bsy_flag[z]=0;
    }
  for (z=0; z<num_aux_res; z++)
    {
     temp_res_id=res_grp_id[z];
     res_grp_bsy_flag[temp_res_id]=0;
     temp_int=0;

     for (l=0; l<num_aux_res; l++)
 {
  if (res_grp_id[l]==res_grp_id[z])
    {
     temp_int=temp_int+stat_mat[stat_indxad][l+2];
    }
 }
     if (temp_int>0)
 {
  res_grp_bsy_flag[temp_res_id]=1;
 }
     temp_int=temp_int-stat_mat[stat_indxad][z+2];
     if (temp_int>0)
 {
  res_grp_excl_bsy_flag[z]=1;
 }
    }
  if (num_st_var>num_aux_res)
    {
     for (z=num_aux_res;z<num_st_var;z++)
 {
  res_grp_excl_bsy_flag[z]=0;
 }
    }

  for (l=0; l<num_st_var; l++)
    {
     ui[l]=0;
    }
  for (l=0; l<num_st_var; l++)
    {//5
     if ((vi[l]==1)&&(omega[l]==1)&&(wip_policy==0))
 {
  if (res_grp_excl_bsy_flag[l]==1)
    {
     ui[l]=bi[l]-1-stat_mat[stat_indxad][l+2];
    }
  if (res_grp_excl_bsy_flag[l]==0)
    {
     ui[l]=bi[l]-stat_mat[stat_indxad][l+2];
    }
 }
     else
 {
  ui[l]=bi[l]-stat_mat[stat_indxad][l+2];
 }
    }//5

  //Next we evaluate the indicator function gamma[k]
  //this indicates whether resource k (if k<num_aux_res) can
accept a customer of type Tbar_k (Equation 18)

  if (omega[k]==1)
    {
     temp_double=1.0;
     gam[k]=1.0; // Initially we assume that resource k can
accept an incoming customer of type Tbar_k
     temp_cus_id=aux_res_cus_st_var_ind[k];
     for (z=0; z<num_aux_res; z++)
 {
  temp_res_id=res_grp_id[z];
  if (res_grp_bsy_flag[temp_res_id]==0)
    {
     if (ARespri[temp_cus_id][k]>ARespri[temp_cus_id][z])
 {
  gam[k]=0.0;
 }
     else if
((ARespri[temp_cus_id][k]==ARespri[temp_cus_id][z])&&(
k!=z))
 {
  temp_double=temp_double+1.0;
 }
     if (temp_double>1.0)
 {
  gam[k]=gam[k]/temp_double;
 }
    }
 }
    }

  //Next, we evaluate the indicator function theta[r]

  for (z=0; z<num_st_var; z++)
    {
     theta[z]=1.0;
    }

  if (vi[k]==1)
    {//5
     for (r=0; r<num_aux_res; r++)
 {
  temp_double=1.0;
  if (AQ[k_bar][r]==1)
    {
     temp_res_id=res_grp_id[r];
     for (t=0; t<num_aux_res; t++)
 {
  if (res_grp_id[t]==res_grp_id[r])
    {
     temp_cus_id=aux_res_cus_st_var_ind[t];
     if
((omega[temp_cus_id]==1)&&(stat_mat[stat_indxad][temp_
cus_id+2]>1-res_grp_excl_bsy_flag[temp_cus_id]))
 {
  if (ACuspri[temp_cus_id][t]<ACuspri[k_bar][r])
    {
     theta[r]=0.0;
    }
  else if
((ACuspri[temp_cus_id][t]==ACuspri[k_bar][r])&&(k_bar!=
temp_cus_id))
    {
     temp_double=temp_double+1.0;
    }
 }
     if
((omega[temp_cus_id]==0)&&(stat_mat[stat_indxad][temp_
cus_id+2]>0))
 {
  if (ACuspri[temp_cus_id][t]<ACuspri[k_bar][r])
    {
     theta[r]=0;
    }
 else if
((ACuspri[temp_cus_id][t]==ACuspri[k_bar][r])&&(k_bar!=
temp_cus_id))
   {
    temp_double=temp_double+1.0;
   }
}
   }
}
   }
 if (temp_double>1.0)
   {
    theta[r]=theta[r]/temp_double;
   }
}
   }//5

 //condition[11]=1 if ALL of the candidate resource groups
for a customer [k] are busy
 //Reserved for interclass transitions. Reserved for state
variable accepting a customer as a result of an inter-class
transition.

 if (vi[k]==1)
   {//5
    condition[11]=1;
    for (l=0; l<num_aux_res; l++)
{
 if (AQ[k][l]==1)
   {
    temp_res_id=res_grp_id[l];
    if (res_grp_bsy_flag[l]==0)
{
 condition[11]=0;
}
   }
}
   }//5

 //condition[12]=1 if ALL of the candidate resource groups
for a customer [k_bar] are busy
 //Reserved for inter-class transitions, but for a state variable
that experiences a unit decrease as a result of a transition.
 if (vi[k_bar]==1)
   {//5
    condition[12]=1;
    for (l=0; l<num_aux_res; l++)
{
 if (AQ[k_bar][l]==1)
   {
    temp_res_id=res_grp_id[l];
    if (res_grp_bsy_flag[l]==0)
{
 condition[12]=0;
}
   }
}
   }//5

 //condition[10]=1 if the queue for customer k is at zero, or
non-empty, but where all of the candidate resource groups
are busy
 //In other words, an event with arrival of a customer of this
type, will cause a queue formation.
 //Reserved for inter-class transitions for state variable that
experiences unit increase, as a result of the transition.

 if (vi[k]==1)
   {
    condition[10]=0;
    if ((omega[k]==1)&&(stat_mat[stat_indxad][k+2]>=1-
res_grp_excl_bsy_flag[k])&&(condition[11]==1))
{
 condition[10]=1;
}
    if
((omega[k]==0)&&(stat_mat[stat_indxad][k+2]>=0)&&(con
dition[11]==1))
{
 condition[10]=1;
}
   }


 //condition[13]=1 if, for all the candidate customer classes
for each resource in the resource group for [k],
 //the number of customers in queue is less than 1.
 //condition[13] is reserved as a flag for the state variable
that has a unit increase as a result of the class transition
 //condition [13] is also used for the self-transition case.

 if (omega[k]==1)
   {//5
    condition[13]=1;
    for (l=0; l<num_st_var; l++)
{
 if (res_grp_id[k]==res_grp_id[l])
   {
    temp_cus_id=aux_res_cus_st_var_ind[l];
    if
((omega[temp_cus_id]==1)&&(stat_mat[stat_indxad][temp_
cus_id+2]>1-res_grp_excl_bsy_flag[temp_cus_id]))
{
 condition[13]=0;
}
    else if
((omega[temp_cus_id]==0)&&(stat_mat[stat_indxad][temp_
cus_id+2]>0))
{
 condition[13]=0;
}
   }
}
   }//5

 //condition[14]=1 if, for all the candidate customer classes
for each resource in the resource group for [k_bar],
 //the number of customers in queue is less than 1.
 //condition[14] is reserved as a flag for the state variable
that has a unit decrease as a result of the class transition
 //condition [14] may also be used for the self-transition
case.
 if (omega[k_bar]==1)
   {
    condition[14]=1;
    for (l=0; l<num_st_var; l++)
{
 if (res_grp_id[k_bar]==res_grp_id[l])
   {
    temp_cus_id=aux_res_cus_st_var_ind[l];
    if
((omega[temp_cus_id]==1)&&(stat_mat[stat_indxad][temp_
cus_id+2]>1-res_grp_excl_bsy_flag[temp_cus_id]))
{
 condition[14]=0;
}
    else if
((omega[temp_cus_id]==0)&&(stat_mat[stat_indxad][temp_
cus_id+2]>0))
{
 condition[14]=0;
}
   }
}
   }


 //condition[15]=1 if for vi[k_bar]=1, the  for all the
candidate resources, the queue for that customer is already
non-empty.
 //In the event of a departure from this state, one of these
queued customers is a candidate for processing at
 //one of the valid resource groups.

 if (vi[k_bar]==1)
   {
    condition[15]=0;
    if
((condition[12]==1)&&(omega[k_bar]==1)&&(stat_mat[stat
_indxad][k_bar+2]>1-res_grp_excl_bsy_flag[k_bar]))
{
 condition[15]=1;
}
    else if
((condition[12]==1)&&(omega[k_bar]==1)&&(stat_mat[stat
_indxad][k_bar+2]>0))
{
 condition[15]=1;
}
   }

 //eps[r]=1 if, the queue of job class r is one of the LQF[r]
longest in the set of resources {r: AQ[k_bar][r]=1.}
    if ((vi[k_bar]==1)&&(condition[15]==1))
   {
    for (r=0; r<num_aux_res; r++)
{
 eps[r]=0;
 LQF[r]=0.0;
 temp_res_id=res_grp_id[r];
 if (AQ[k_bar][r]==1)
   {
    eps[r]=1;
    LQF[r]=0.0;
    temp_double=1.0;
    for (t=0; t<num_aux_res; t++)
{
 if (eps[r]==1)
   {
    if (res_grp_id[t]==res_grp_id[r])
{
 temp_cus_id=aux_res_cus_st_var_ind[t];

temp_int=stat_mat[stat_indxad][temp_cus_id+2]+omega[tem
p_cus_id]*res_grp_excl_bsy_flag[temp_cus_id];

temp_int1=stat_mat[stat_indxad][k_bar+2]+omega[k_bar]*re
s_grp_excl_bsy_flag[k_bar];
 if (temp_int>temp_int1)
   {
    eps[r]=0;
   }
 else if ((temp_int==temp_int1)&&(k_bar!=temp_cus_id))
   {
    temp_double=temp_double+1.0;
   }
}
   }
}
   }
 if (eps[r]==1) LQF[r]=temp_double;
}
   }

 //Condition[16]=1 if, for a given class, a departure from the
system (or network) is possible for that class
 //In other words, the customer leaves the system with
probability strictly less than 1.
 if (APhi_row_sum[k_bar]<1.0)
   {
    condition[16]=1;
   }

 //Revision Progress as of 14/07/04 17:38 PM.

 //Now we write the transition rates for the inter-class
(including self-) transitions.
 //We have to write the transition rates separately for each
combination of routing-sequencing policies.
 if ((k!=k_bar)&&(stat_indxad_flag==1))
   {
    temp_res_id=res_grp_id[k];
    if ((vi[k]==1)&&(condition[10]==1)&&(ui[k]>=1))
{
 if
((omega[k_bar]==1)&&(stat_mat[stat_indxad][k_bar+2]==1
)&&(res_grp_excl_bsy_flag[k_bar]==0)&&(condition[14]==
1))
   {
    tran_val=0.0;//reset this variable
    if
(((rt_policy==0)&&(seq_policy==2))||((rt_policy==1)&&(se
q_policy==2)))
{
 temp_cus_id=aux_res_cus_st_var_ind[k_bar];
 tran_val=AMu[k_bar]*APhi[temp_cus_id][k];
}
    if
(((rt_policy==1)&&(seq_policy==0))||((rt_policy==0)&&(se
q_policy==0)))
{
 temp_cus_id=aux_res_cus_st_var_ind[k_bar];
 tran_val=AMu[k_bar]*APhi[temp_cus_id][k];
}
    //Note that both if conditions above result in the same
transition rate.
   }
 else if ((vi[k_bar]==1)&&(condition[15]==1))
   {
    tran_val=0.0; //reset this variable
    if
(((rt_policy==0)&&(seq_policy==2))||((rt_policy==1)&&(se
q_policy==2)))
{
 for (r=0; r<num_aux_res; r++)
   {
    if (AQ[k_bar][r]==1)
{
 if (LQF[r]>0.0)
   {
    temp_cus_id=aux_res_cus_st_var_ind[r];

tran_val=tran_val+AMu[r]*APhi[temp_cus_id][k]*eps[r]/LQ
F[r];
   }
}
   }
}
```

```
    if
(((rt_policy==1)&&(seq_policy==0))||((rt_policy==0)&&(se
q_policy==0)))
{
 for (r=0; r<num_aux_res; r++)
   {
    if (AQ[k_bar][r]==1)
{
 temp_cus_id=aux_res_cus_st_var_ind[r];
 tran_val=tran_val+AMu[r]*theta[r]*APhi[temp_cus_id][k];
}
   }
}
   }
}
    else if
((omega[k]==1)&&(res_grp_bsy_flag[temp_res_id]==0)&&(
condition[13]==1))
{
 tran_val=0.0;
 if
((omega[k_bar]==1)&&(stat_mat[stat_indxad][k_bar+2]==1
)&&(res_grp_excl_bsy_flag[k_bar]==0)&&(condition[14]==
1))
   {
    temp_cus_id1=aux_res_cus_st_var_ind[k_bar];
    temp_cus_id=aux_res_cus_st_var_ind[k];
    tran_val=0.0; //reset this variable
    if
(((rt_policy==0)&&(seq_policy==1))||((rt_policy==0)&&(se
q_policy==0)))
{

tran_val=AMu[k_bar]*APhi[temp_cus_id1][temp_cus_id]*g
am[k];
}
    if
(((rt_policy==1)&&(seq_policy==2))||((rt_policy==1)&&(se
q_policy==0)))
{
 tran_val=AMu[k_bar]*APhi[temp_cus_id1][temp_cus_id];
 temp_double=0.0;
 for (t=0; t<num_aux_res; t++)
   {
    temp_res_id1=res_grp_id[t];
    temp_double=temp_double+AQ[temp_cus_id][t]*(1-
res_grp_bsy_flag[temp_res_id1]);
   }
 if (temp_double>0.0)
   {
    tran_val=tran_val/temp_double;
   }
}
   }
 else if ((vi[k_bar]==1)&&(condition[15]==1))
   {
    tran_val=0.0; //reset this variable
    temp_cus_id=aux_res_cus_st_var_ind[k];
    if ((rt_policy==0)&&(seq_policy==1))
{
 for (r=0; r<num_aux_res; r++)
   {
    if (AQ[k_bar][r]==1)
{
 if (LQF[r]>0.0)
   {
    temp_cus_id1=aux_res_cus_st_var_ind[r];

tran_val=tran_val+AMu[r]*APhi[temp_cus_id1][temp_cus_i
d]*eps[r]/LQF[r];
   }
}
   }
 tran_val=tran_val*gam[k];
}
    if ((rt_policy==1)&&(seq_policy==2))
{
 for (r=0; r<num_aux_res; r++)
   {
    if (AQ[k_bar][r]==1)
{
 if (LQF[r]>0.0)
   {
    temp_cus_id1=aux_res_cus_st_var_ind[r];

tran_val=tran_val+AMu[r]*APhi[temp_cus_id1][temp_cus_i
d]*eps[r]/LQF[r];
   }
}
   }
 temp_double=0.0;
 for (t=0; t<num_aux_res; t++)
   {
    temp_res_id1=res_grp_id[t];
    temp_double=temp_double+AQ[k_bar][t]*(1-
res_grp_bsy_flag[temp_res_id1]);
   }
 if (temp_double>0.0)
   {
    tran_val=tran_val/temp_double;
   }
}
    if ((rt_policy==1)&&(seq_policy==0))
{
 for (r=0; r<num_aux_res; r++)
   {
    if (AQ[k_bar][r]==1)
{
 temp_cus_id1=aux_res_cus_st_var_ind[r];

tran_val=tran_val+AMu[r]*theta[r]*APhi[temp_cus_id1][te
mp_cus_id];
}
   }
 temp_double=0.0;
 for (t=0; t<num_aux_res; t++)
   {
    temp_res_id1=res_grp_id[t];
    temp_double=temp_double+AQ[k_bar][t]*(1-
res_grp_bsy_flag[temp_res_id1]);
   }
 if (temp_double>0.0)
   {
    tran_val=tran_val/temp_double;
   }
}
    if ((rt_policy==0)&&(seq_policy==0))
{
 for (r=0; r<num_aux_res; r++)
   {
    if (AQ[k_bar][r]==1)
{
 temp_cus_id1=aux_res_cus_st_var_ind[r];

tran_val=tran_val+AMu[r]*theta[r]*APhi[temp_cus_id1][te
mp_cus_id];
}
   }
 tran_val=tran_val*gam[k];
}
```

```
    }
}
     if ((tran_val>0.0)&&(stat_mat[stat_indxad][0]==1))
{
 tran_indx1=stat_mat[stat_indxad][st_sp_width-1];
 tran_indx2=stat_mat[i][st_sp_width-1];
 row_sum[tran_indx1]=row_sum[tran_indx1]+tran_val;
 check_int=real_size-1;
 if (tran_indx2!=check_int)
   {
    tran_row_val[num_values]=tran_indx2;
    tran_col_val[num_values]=tran_indx1;
    tran_mat_val[num_values]=tran_val;

    if ((tran_realloc_flag1 = realloc(tran_row_val, sizeof(int)
* (tran_mem_cnt + 1))) == NULL) {
printf("ERROR: realloc1 failed\n");
    }
    tran_row_val = tran_realloc_flag1;

    if ((tran_realloc_flag2 = realloc(tran_col_val, sizeof(int)
* (tran_mem_cnt + 1))) == NULL) {
printf("ERROR: realloc2 failed\n");
    }
    tran_col_val=tran_realloc_flag2;

    if ((tran_realloc_flag3 = realloc(tran_mat_val,
sizeof(double) * (tran_mem_cnt + 1))) == NULL) {
printf("ERROR: realloc1 failed\n");
    }
    tran_mat_val=tran_realloc_flag3;

    tran_mem_cnt++;

    num_values++;
  }
 tran_val=0.0;
}
  }
 if (k==k_bar)
   {
    tran_val=0.0;
    if
((omega[k]==1)&&(stat_mat[stat_indxad][k+2]==1)&&(res
_grp_excl_bsy_flag[k]==1)&&(condition[13]==1))
{
 temp_cus_id=aux_res_cus_st_var_ind[k];
 tran_val=0.0;//reset this variable
 if
(((rt_policy==0)&&(seq_policy==2))||((rt_policy==0)&&(se
q_policy==0)))
   {

tran_val=AMu[k]*APhi[temp_cus_id][temp_cus_id]*gam[k]
;
   }
 if
(((rt_policy==1)&&(seq_policy==2))||((rt_policy==1)&&(se
q_policy==0)))
   {

tran_val=AMu[k_bar]*APhi[temp_cus_id][temp_cus_id];
    temp_double=0.0;
    for (t=0; t<num_aux_res; t++)
{
 temp_res_id1=res_grp_id[t];
 temp_double=temp_double+AQ[temp_cus_id][t]*(1-
res_grp_bsy_flag[temp_res_id]);
}
    if (temp_double>0.0)
{
 tran_val=tran_val/temp_double;
}
   }
}
    if ((vi[k]==1)&&(condition[15]==1))
{
 tran_val=0.0; //reset this variable
 if
(((rt_policy==0)&&(seq_policy==2))||((rt_policy==1)&&(se
q_policy==2)))
   {
    for (r=0; r<num_aux_res; r++)
{
 if (AQ[k][r]==1)
   {
    if (LQF[r]>0.0)
{
 temp_cus_id1=aux_res_cus_st_var_ind[r];

tran_val=tran_val+AMu[r]*APhi[temp_cus_id1][k]*eps[r]/L
QF[r];
}
   }
}
   }
 if
(((rt_policy==1)&&(seq_policy==0))||((rt_policy==0)&&(se
q_policy==0)))
   {
    for (r=0; r<num_aux_res; r++)
{
 if (AQ[k][r]==1)
   {
    temp_cus_id1=aux_res_cus_st_var_ind[r];

tran_val=tran_val+AMu[r]*theta[r]*APhi[temp_cus_id1][k];
   }
}
   }
}
    if ((tran_val>0.0)&&(stat_mat[stat_indxad][0]==1))
{
 tran_indx1=stat_mat[stat_indxad][st_sp_width-1];
 tran_indx2=stat_mat[i][st_sp_width-1];
 row_sum[tran_indx1]=row_sum[tran_indx1]+tran_val;
 check_int=real_size-1;
 if (tran_indx2!=check_int)
   {
    tran_row_val[num_values]=tran_indx2;
    tran_col_val[num_values]=tran_indx1;
    tran_mat_val[num_values]=tran_val;

    if ((tran_realloc_flag1 = realloc(tran_row_val, sizeof(int)
* (tran_mem_cnt + 1))) == NULL) {
printf("ERROR: realloc1 failed\n");
    }
    tran_row_val = tran_realloc_flag1;

    if ((tran_realloc_flag2 = realloc(tran_col_val, sizeof(int)
* (tran_mem_cnt + 1))) == NULL) {
printf("ERROR: realloc2 failed\n");
    }
    tran_col_val=tran_realloc_flag2;

    if ((tran_realloc_flag3 = realloc(tran_mat_val,
sizeof(double) * (tran_mem_cnt + 1))) == NULL) {
printf("ERROR: realloc1 failed\n");
    }
    tran_mat_val=tran_realloc_flag3;
```

```
      tran_mem_cnt++;

      num_values++;
    }
  tran_val=0.0;
}
    }
}
    }
}
    }
  fclose(Stat_mat);



  system ("cp /tmp/Stat_mat Stat_mat");
  printf(" Number of Values Here = %d\n", num_values);




  for (i=0; i<real_size-1;i++)
    {
      tran_row_val[num_values]=i;
      tran_col_val[num_values]=i;
      tran_mat_val[num_values]=-row_sum[i];
      num_values++;
      if ((tran_realloc_flag1 = realloc(tran_row_val, sizeof(int)
* (tran_mem_cnt + 1))) == NULL) {
printf("ERROR: realloc1 failed\n");
      }
      tran_row_val = tran_realloc_flag1;
      if ((tran_realloc_flag2 = realloc(tran_col_val, sizeof(int)
* (tran_mem_cnt + 1))) == NULL) {
printf("ERROR: realloc2 failed\n");
      }
      tran_col_val=tran_realloc_flag2;
      if ((tran_realloc_flag3 = realloc(tran_mat_val,
sizeof(double) * (tran_mem_cnt + 1))) == NULL) {
printf("ERROR: realloc1 failed\n");
      }
      tran_mat_val=tran_realloc_flag3;
      tran_mem_cnt++;
    }
  for (i=0; i<real_size;i++)
    {
      tran_row_val[num_values]=real_size-1;
      tran_col_val[num_values]=i;
      tran_mat_val[num_values]=1.0;
      num_values++;

      if ((tran_realloc_flag1 = realloc(tran_row_val, sizeof(int)
* (tran_mem_cnt + 1))) == NULL) {
printf("ERROR: realloc1 failed\n");
      }
      tran_row_val = tran_realloc_flag1;
      if ((tran_realloc_flag2 = realloc(tran_col_val, sizeof(int)
* (tran_mem_cnt + 1))) == NULL) {
printf("ERROR: realloc2 failed\n");
      }
      tran_col_val=tran_realloc_flag2;
      if ((tran_realloc_flag3 = realloc(tran_mat_val,
sizeof(double) * (tran_mem_cnt + 1))) == NULL) {
printf("ERROR: realloc1 failed\n");
      }
      tran_mat_val=tran_realloc_flag3;
      tran_mem_cnt++;
    }
  FILE *tran_values_file;
  tran_values_file=fopen("/tmp/tran_val_file", "w");
  for (i=0; i<num_values; i++)
    {
      fprintf(tran_values_file, "%d\t %d\t %lf\n",
tran_row_val[i], tran_col_val[i], tran_mat_val[i]);
      fflush(tran_values_file);
    }
  fclose(tran_values_file);
  system ("cp /tmp/tran_val_file tran_val_file");

  int * tran_row_indx, *tran_col_indx, *row_cnt_indx,
*new_row_cnt_indx, *tran_dupl_ind;
  double *tran_values, *new_values;

  tran_row_indx=(int *) calloc (num_values, sizeof (int));
  tran_col_indx=(int *) calloc (num_values, sizeof (int));
  tran_values=(double *) calloc (num_values, sizeof
(double));
  tran_dupl_ind = (int *) calloc (num_values, sizeof(int));
  row_cnt_indx=(int *) calloc (real_size, sizeof (int));
  new_row_cnt_indx=(int *) calloc (real_size, sizeof (int));

  int row_cnt, col_temp, row_temp, max_num_row_vals;
  row_cnt=0;
  col_cnt=0;

  for (i=0; i<real_size; i++)
    {
      row_cnt_indx[i]=0;
    }

  int row_temp1, col_temp1, val_temp1;
  for (i=0; i<num_values; i++)
    {
      col_temp=tran_col_val[i];
      row_cnt_indx[col_temp]++;//increment the number of
rows in column number col_temp
    }

  max_num_row_vals=0;

  for (i=0; i<real_size; i++)
    {
      if (row_cnt_indx[i]>max_num_row_vals)
{
  max_num_row_vals=row_cnt_indx[i];//determining the
maximum depth of the transition matrix in terms of number
of non-zero rows
}
    }

  printf("MAX NUMBER OF ROW_VALS=%d\n",
max_num_row_vals);//max_number of row_values

  double **sparse_tran_mat, *sparse_tran_temp;
  int true_num_values;

  sparse_tran_temp= (double *) calloc
(2*real_size*max_num_row_vals, sizeof (double));
  sparse_tran_mat= (double **) calloc (real_size, sizeof
(double *));

  //this matrix contains the sparse compacted version of the
transition matrix

  if (sparse_tran_temp==NULL || sparse_tran_mat==NULL)
    {
      printf ("error in sparse_tran_mat calloc allocation\n");
    }
  else
```

```
    {
     printf("sparse_tran_mat calloc completed\n");
    }

  //Make sparse_tran_mat point to the beginning of each row
in sparse_tran_temp

  for (i=0; i<real_size; i++)
sparse_tran_mat[i]=sparse_tran_temp+2*i*max_num_row_v
als;

  for (i=0;i<real_size;i++)
    {
     for (j=0; j<2*max_num_row_vals;j++)
{
 sparse_tran_mat[i][j]=0.0;//initialize to avoid errors
}
    }

  for (i=0; i<real_size; i++)
    {
     new_row_cnt_indx[i]=0;//this index
    }

  for (i=0; i<num_values; i++)
    {
     col_temp=tran_col_val[i];//the column number of the
sparse element
     row_temp=new_row_cnt_indx[col_temp];//the row
number of a sparse element

sparse_tran_mat[col_temp][row_temp]=tran_row_val[i];//for
each column store the row number

sparse_tran_mat[col_temp][row_temp+1]=tran_mat_val[i];//f
or each column store the value

new_row_cnt_indx[col_temp]=new_row_cnt_indx[col_temp
]+2;//for each column store the number of entries made
    }

  true_num_values=num_values;

  for (i=0; i<real_size; i++)
    {
     if (new_row_cnt_indx[i]>0)
{
 for (j=0; j<max_num_row_vals;j++)
   {
    if
((sparse_tran_mat[i][2*j+1]>0.0)||(sparse_tran_mat[i][2*j+1]
<0.0))
{
 temp_int=sparse_tran_mat[i][2*j];
 for (k=j+1;k<max_num_row_vals;k++)
   {
    temp_int1=sparse_tran_mat[i][2*k];
    if
((temp_int1==temp_int)&&((sparse_tran_mat[i][2*k+1]>0.0
)||(sparse_tran_mat[i][2*k+1]<0.0)))
{

sparse_tran_mat[i][2*j+1]=sparse_tran_mat[i][2*j+1]+sparse
_tran_mat[i][2*k+1];
 sparse_tran_mat[i][2*k]=0.0;
 sparse_tran_mat[i][2*k+1]=0.0;
 true_num_values=true_num_values-1;
}
   }
}
    }
}
    }


  printf("NUMBER OF VALUES=%d\n", num_values);
  printf("TRUE NUMBER OF VALUES=%d\n",
true_num_values);

/*    for (i=0; i<num_values; i++) */
/*    { */
/*    printf("row_indx= %d\t", tran_row_indx[i]);print out
the values to check. */
/*    printf("col_indx= %d\t", tran_col_indx[i]); */
/*    printf("tran_val= %lf\t", tran_values[i]); */
/*    printf("i = %d\n", i); */
/*    } */


  int val_cnt;
  val_cnt=0;

/*  for (i=0; i<real_size; i++) */
/*    { */
/*     printf ("%d\t", i); */
/*     for (j=0; j<2*max_num_row_vals; j++) */
/* { */

/*  printf("%lf\t", sparse_tran_mat[i][j]); */
/*  val_cnt++; */
/* } */
/*     printf("EOC\n"); */
/*    } */

/*  printf("Value Count = %d\n", val_cnt); */



  //Next step is create the CSC format input data for the
dgssfs routine

  int *colptr, *rowind, true_num_val_cnt;
  double *values;


  colptr=(int *) calloc (real_size+1, sizeof (int));
  rowind=(int *) calloc (num_values, sizeof (int));
  values=(double *) calloc (num_values, sizeof (double));


  for (i=0; i<=real_size; i++)
    {
     colptr[i]=0;
    }

  for (i=0; i<num_values; i++)
    {
     values[i]=0.0;
     rowind[i]=0;
    }
  int val_indx;
  val_indx=0;
  col_cnt=0;
  row_indx=0;
  true_num_val_cnt=0;
  for (i=0; i<real_size; i++)
    {
     colptr[i]=val_indx+1;
     for (j=0; j<max_num_row_vals;j++)
{
 if
((sparse_tran_mat[i][2*j+1]>0.0)||(sparse_tran_mat[i][2*j+1]
<0.0))
   {
    rowind[val_indx]=sparse_tran_mat[i][2*j]+1;
    values[val_indx]=sparse_tran_mat[i][2*j+1];
    val_indx++;
    true_num_val_cnt++;
   }
}
   }
 true_num_values=true_num_val_cnt;




 colptr[real_size]=val_indx+1;

 for (i=0; i<=real_size; i++)
   {
    printf("colptr[%d]=%d\n", i, colptr[i]);
   }

 for (i=0; i<true_num_values; i++)
   {
    printf("rowind[%d]=%d values[%d]=%lf\n", i, rowind[i],
i, values[i]);
   }

 colstr=fopen ("/tmp/colstr", "w");
 rownd=fopen ("/tmp/rownd", "w");
 vals=fopen ("/tmp/vals", "w");

 for (i=0; i<=real_size; i++)
   {
    fprintf(colstr, "%d\n", colptr[i]);
    fflush(colstr);
   }
 fclose(colstr);

 for (i=0; i<true_num_values; i++)
   {
    fprintf(rownd, "%d\n", rowind[i]);
    fflush(rownd);
    fprintf(vals, "%lf\n", values[i]);
    fflush(vals);
   }
 fclose(rownd);
 fclose(vals);

 system ("cp /tmp/colstr colstr");
 system ("cp /tmp/rownd rownd");
 system ("cp /tmp/vals vals");

 fprintf(problem_size, "%d\n", real_size);
 fflush(problem_size);
 fprintf(problem_size, "%d\n", true_num_values);
 fflush(problem_size);
 fclose(problem_size);

 system ("cp /tmp/size_info size_info");

 char names[25];
 strcpy(names, "fortrancall");
 system(names);

 FILE *P_val, *results;
 double * pval;
```

```
 pval=(double *) calloc (real_size, sizeof (double));

 P_val=fopen ("/tmp/P_val", "r");

 for (i=0; i<real_size; i++)
   {
    fscanf(P_val, "%lf", &pval[i]);
    printf("%lf\n", pval[i]);
   }
 fclose(P_val);

 system ("cp /tmp/P_val P_val");

 double *res_thr, *cus_thr;

 res_thr=(double *) calloc (m+1, sizeof (double));
 cus_thr=(double *) calloc (m+1, sizeof(double));

 for (i=0; i<=m; i++)
   {
    res_thr[i]=0.0;
   }

 for (i=0; i<=m; i++)
   {
    cus_thr[i]=0.0;
   }

 for (i=0; i<st_sp_size; i++)
   {
    if (stat_mat[i][0]!=0)
{
 for (j=2; j<st_sp_width-2; j++)
   {
    if (stat_mat[i][j]==0)
{
 row_indx=stat_mat[i][st_sp_width-1];
 res_thr[j-2]=res_thr[j-2]+pval[row_indx];
}
    if (stat_mat[i][j]==b)
{
 row_indx=stat_mat[i][st_sp_width-1];
 cus_thr[j-2]=cus_thr[j-2]+pval[row_indx];
}
   }
}
   }

 results=fopen ("/tmp/Results", "w");
 /*
 for (i=0; i<m; i++)
   {
    temp_double=1.0-res_thr[i];
    temp_double1=Mu[i];
    res_thr[i]=temp_double*temp_double1;
    res_thr[m]=res_thr[m]+res_thr[i];
    fprintf(results, "%f\t", res_thr[i]);
    fflush(results);
   }
 */
 fprintf(results, "%f\n", res_thr[m]);
 fflush(results);

 for (i=0; i<m;i++)
   {
    temp_double=1.0-cus_thr[i];
    temp_double1=Lam[i];
    cus_thr[i]=temp_double*temp_double1;
    cus_thr[m]=cus_thr[m]+cus_thr[i];
    fprintf(results, "%f\t", cus_thr[i]);
    fflush(results);
   }

 fprintf(results, "%f\n", cus_thr[m]);
 fflush(results);

 fclose(results);
 system ("cp /tmp/Results Results");

 free(pval);
 free(res_thr);
 free(cus_thr);
 free(row_sum);
 free(stat_temp);
 free(stat_mat);
 free(b_mat);
 free(rowind);
 free(colptr);
 free(values);
 printf("%d\n",ok);

 return 0;
}

 printf("%d\n",ok);

 return 0;
}
```