Performance Evaluation of an Utility Model Based Network Admission Controller

by

Eric Peter Gowland
B.Eng, University of Victoria, 2001

A Thesis Submitted in Partial Fulfillment of the Requirements
for the Degree of

MASTER OF APPLIED SCIENCE

in the Department of Electrical and Computer Engineering

Supervisor: Dr. Eric G. Manning

ABSTRACT

This work continues research into Utility Model based Network Admission Control.

Given a network that can provide guaranteed Quality of Service, a decision must be made as to which user sessions requesting access to this service gain admission. The Utility Model can be used to solve this as a resource allocation problem. The goal is to maximize a value function (Utility) while obeying the resource constraints of the network.

We present a Model Implementation of a Utility Model based Network Admission Controller. A thorough and systematic series of performance evaluation experiments are run using this model, and the results are presented.

The main contributions of this work are a systematic methodology for evaluating the performance of a Network Admission Control system, along with a detailed characterization of the performance of our Model Implementation.

Table of Contents

List of Tables

List of Figures

Acknowledgement

I would first and foremost like to acknowledge the support of my supervisor, Dr. Eric G. Manning.

Next I must thank my family, for their constant love and support: Paul and Lori Gowland, my father and mother, and Lindsay and Leah, my two sisters.

The support of all the members of UVic's Panda group has proven invaluable.

Last, but certainly not least, the many friends and mentors who have helped me throughout my time at UVic: Ted Davis, Kevin Buckham, Melanie Findlater, and Morgan Hay to name but a few.

Dedication

Ronald Gowland and Douglas Smith.

Who continue to inspire me.

Chapter 1

Introduction

1.1 Objective

The primary goal of this work is to improve upon and evaluate in detail the performance of a Utility Model based Network Admission Controller ([18],[19],[20], [21],[22],[23],[29], [31],[32],[33],[35]).

Performance can be interpreted in many different ways. In the context of a Network Admission Controller, there are two primary metrics. The first is the quality of the admission solution. This is concerned with how good a job the controller does at managing network resources and meeting objective functions for the network. More on how the Utility Model defines and achieves this can be found in later chapters.

The second metric for Admission Controller performance is the time taken to arrive at a given solution. This could also be termed *decision time* or *runtime*. This is crucially important in an application environment where decisions must be made in realtime, as to which sessions get access to the network and which do not.

Previous work on an Utility Model based Network Admission Controller has focused on this first metric, sometimes at the expense of the second. This was a reasonable approach for proving the concept of such a controller, and for showing that valuable gains in the efficiency of network resource allocation could be gained through its employment. When considering an actual implementation of such a controller, however, the second metric becomes increasingly relevant. The controller must admit, upgrade or reject sessions in

an acceptably short amount of time.

Thus, controller runtime is the focus of this work, the primary goal being to improve and make quantitative measurements of this metric while preserving the efficient resource allocation or near-optimality of obtained solutions achieved in previous work.

Implicit to achieving this goal is to establish a sound methodology and experimental process for characterizing the performance of a Network Admission Controller. This includes identifying the independent and dependant variables involved and designing experiments to manipulate and observe these.

This process will likely identify avenues for improvement of either the performance or functionality of the Admission Controller being evaluated.

1.2 Motivation

Here, some discussion of what a Network Admission Controller is and why research in this area is of importance will be provided.

The envisioned widespread adoption of high-bandwidth, real-time multimedia applications has been a driving force behind data network research for some time. Increasingly, the Internet or similar data networks are seen as the mechanism for delivering not just the Best-Effort Service adequate for email and file-transfer applications, but the guaranteed Quality of Service (QoS) necessary for two-way voice, movies, video-conferencing, gaming and other demanding applications. Many of these applications are already available and have customers. They are not yet commonplace in households and businesses, but the general consensus in industry, and our assumption, is that they will become so.

One of the primary barriers to making these applications ubiquitous is delivering absolute, end-to-end QoS. Several Internet Engineering Task Force (IETF) groups have been working on QoS related problems for some time. Integrated Services (IntServ [26]), Differentiated Services (DiffServ [30]), Multi Protocol Label Switching (MPLS [4]) and Resource Reservation Protocol (RSVP [27]) are but some of their proposed solutions for Inter-

net Protocol (IP [13]) based networks. These and other methods for providing end-to-end QoS in a network are discussed further in the next chapter.

Given methods for ensuring end-to-end QoS, a problem arises in determining who should receive such service. One approach is to provide sufficient resources to make this service available for everybody, but this has the potential for high inefficiency. A more realistic scenario is the classic situation of a group of users competing for the limited resources available in a network. Determining whom to allow to utilize the network, and the amount of resources to dedicate to each admitted user, so as to maximize some metric of value or utility, is the problem of *Network Admission Control.*

To this end, we present a Utility Model based Network Admission Controller. The Utility Model [31] is a resource allocation model developed with QoS related problems in mind.

Previous work [18] has shown that such a controller can achieve near optimal assignment of network resources. Our motivation in continuing that work is to demonstrate that this controller is suitable for deployment in a real application environment. This will be achieved through a re-design and re-implementation of the controller combined with a thorough performance evaluation and characterization.

1.3 Scope & Key Assumptions

There are several underlying assumptions and questions of scope that must be discussed.

To fully consider the Network Admission Control process, one must study not only the admission decision making process, but also how requests are gathered and admission notifications returned to the users. The controller presented in this work assumes that some external process collects and delivers admission requests in a batch to the controller, which returns its *admission solution* to this same entity. This solution consists of the set of requests to be admitted and their QoS levels.

The nature of the request gathering process is beyond the scope of this work. Our

performance evaluation focuses purely on the computational performance of the admission controller, independent of the processes supporting it.

The controller presented here is of a centralized design. The issues inherent in such a centralized control scheme are not discussed in this work, although previous research [34] has discussed them.

Specific assumptions relating to the underlying models used in our Network Admission Controller and to the experiments performed, are discussed in the following chapters.

1.4 Organization

The remainder of this thesis is organized as follows.

Chapter 2 discusses additional background information. Chapter 3 discusses solutions to the *Multiple-Choice Multi-Dimensional Knapsack Problem* (MMKP). Chapter 4 introduces the *Model Implementation* of a Utility Model Based Network Admission Controller, and discusses its architecture. The methodology used in designing the performance evaluation experiments is discussed in Chapter 5. Chapter 6, 7 and 8 present the parameters and results of these experiments. Chapter 9 and 10 present conclusions and suggestions for future work, respectively.

Chapter 2

Background

This work is a continuation of Network Admission Control research based on the Utility Model. Thus, there are a few background topics that need to be discussed to establish the context of our work.

2.1 Data Network Quality of Service

> "the Internet focuses more on where to send packets and little on the *when*."
> - Grenville Armitage [9] -

This statement catches the essence of what network quality of service means, and why existing network technology does not provide it. The Internet Protocol (IP) is very robust when it comes to (eventually) delivering a given packet to its destination, if that destination is reachable. However, it provides no guarantee as to *when that packet will arrive.*

Yet the list of so-called killer-apps, those often talked about world-changing applications of data networks, contains several applications where timeliness in information delivery is of vital importance. Interactive voice or video, delivery of multimedia content, or interactive gaming (which can combine all of these) are, by their nature, sensitive to the timely delivery of data.

In discussing network quality of service, and specifically IP QoS, there are several parameters to consider. The first two metrics mentioned are usually bandwidth and delay. *Bandwidth* refers to the rate at which data can be sent through the network (generally

in some variant of bits per second). For a quality of service sensitive application to exhibit consistent behaviour, it must have sufficient and guaranteed transmission bandwidth through the network for the lifetime of the session.

For example, a two-way video conference must have sufficient bandwidth in both directions for the continuous delivery of the video and audio feed - approximately 3-4 Mb/s for an intermediate quality MPEG stream [1]. The amount required depends upon the desired perceived quality of the session. If a particular link cannot provide the bandwidth required, the network cannot provide the session with the desired QoS.

Delay refers to the end-to-end delay introduced by the latency in links and the forwarding delays in routers. For an interactive session such as our video conference, bounds on delay are essential to prevent the participants from noticing lag in their conversational interaction.

A third metric, *Jitter*, refers to the phenomena of a steady stream of packets becoming clumped together in time as they are forwarded through the network. This disrupts the temporal ordering of the information and can be difficult for applications to handle. If left unaltered by the application, jitter would be evident as (for example) variation in the frame rate of video.

To provide consistency in an application, the network must be capable of guaranteeing end-to-end bandwidth availability, an upper bound on end-to-end delay and bounds on jitter. The question of how to accomplish this has attracted a great deal of research. In relation to IP networks, this can be broken down into three main areas. We will discuss each of these briefly.

2.1.1 Per-hop QoS

This term refers to what can be done for quality of service at the lowest level of granularity - the individual routers joining two or more links in the network. Generally, work in this area revolves around classifying different types of traffic, and using multiple queues and scheduling techniques to provide the desired behaviour (Classify, Schedule, Queue archi-

tecture - CQS [9]).

As a simple example, a router might identify two priority classes of traffic according to information in the IP header. The higher priority goes into one queue, all other traffic into the other. When forwarding packets, the router might use a scheme where it forwards two packets from the high priority queue for every one in the low priority, unless the high priority queue is empty. More complex approaches involve more queues, more advanced classification techniques, and better scheduling schemes.

There is a cost associated with the amount of processing that must be done to support a more complex scheme. This affects the granularity of traffic differentiation. For example, packet flows could be sorted by individual application session, allowing prioritizing of traffic according to both user and application. This obviously requires a great deal more computation than differentiating traffic according to application type. Some approaches suggest finer differentiation at the edge of the network, translated to coarser organization in the core. Thus, complexity is kept closer to the applications at the edge while core routers can be simple and performance focused.

Another important per-hop behaviour is traffic shaping and policing. This refers to how a router handles bursts in traffic. Along with guaranteeing a minimum transmission rate, a maximum may also be specified for a given traffic class. This helps smooth (shape) traffic as it propagates through the network, reducing jitter. Policing refers to dropping packets from flows that exceed their authorized transmission rate, enforcing any service contract. A less extreme approach is to first mark packets exceeding a certain rate bound, and either treat them differently or drop them if they continue to violate the rules.

In the context of Network Admission Control, consider that each per-hop behaviour could introduce a great deal of complexity. Our controller abstracts these behaviours to more general QoS metrics and models this as a resource allocation problem. Other admission control schemes might use other approaches.

2.1.2 Routing and Traffic Engineering

Traditional IP routing generally uses a shortest-path approach within each subnet, such as Open Shortest Path First (OSPF [12]). This does not take advantage of multiply-connected topologies, where traffic can be distributed more evenly amongst alternate paths to reduce the number of choke points in the network. Thus, non-shortest path routing techniques, such as *explicit routing*, are important for improving overall QoS. Explicit routing refers to explicitly controlling or specifying each hop for packets of a given flow.

Manipulating how traffic is routed through the network with the goal of improving performance is generally referred to as Traffic Engineering. There is again a trade off to consider in deciding how specific the routing mechanism is. Finding and securing explicit paths for each application session would require far more computation and messaging than defining paths for general application types.

Routing is related to admission control in determining how network QoS capabilities are represented. The route for a session will affect which resources it consumes, and different routes might imply different resource requirements. If the admission controller bases its decision on resource availability, it must be aware of the impact which routing will have.

2.1.3 Signalling and Provisioning

This covers the management concerns of achieving QoS. It refers to how actual behaviour at the nodes of a network is coordinated. This can involve the dissemination of classification and corresponding desired behaviour throughout the network, and the request for and assignment of resources at each hop in a given sessions path. Signalling refers to the automated version of this process, which can vary in how dynamically it responds to changes. Provisioning usually indicates direct human control - that is, the manual configuration of the network. While accomplishing the same outcome, this is generally orders of magnitude slower in responding to changes.

Signalling is closely tied to admission control, as a signalling process would be re-

quired to request admission to the network, determine whether the network can support the request, and assign resources once a request is granted.

2.2 The Utility Model & The Multiple-Choice Multi-Dimension Knapsack Problem

The Utility Model was first introduced by Manning & Khan [31]. It was presented as a model for solving multi-user resource allocation problems, specifically in the context of a distributed multimedia system.

The Utility Model considers the different resources available in a system as a vector. For example, in a single computer, the available resources could include CPU cycles, memory, and network interface bandwidth. A scalar value for each of these would indicate the amount of that resource available for use, and together these would form the systems *available resource vector.*

Users or programs competing for these resources organize their requests into *Service Level Agreements* (SLAs). These specify what resources a given session requires in exchange for providing a certain value of utility. *Utility* can be mapped to any objective function desired - A typical mapping might be to revenue. Another might be user satisfaction. The definition of an appropriate utility function is left beyond the scope of this work.

Users specify how much utility to provide in exchange for a given set of resource allotments. A single SLA can specify multiple levels of resource-utility mappings. We will term these *Admission Profiles.* These can be considered to represent different mappings of QoS resource requirements to different offered utility values. Generally, the greater the resource requirements for a given Admission Profile, the more utility would be offered.

Returning to our single computer example, a given application might require 20% of the CPU, 5% of the memory, and 25% of the network bandwidth for its base QoS level, and

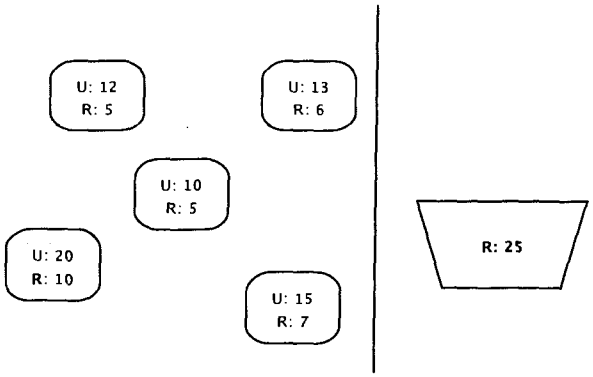

Figure 2.1. *A simple, classical Knapsack Problem.*

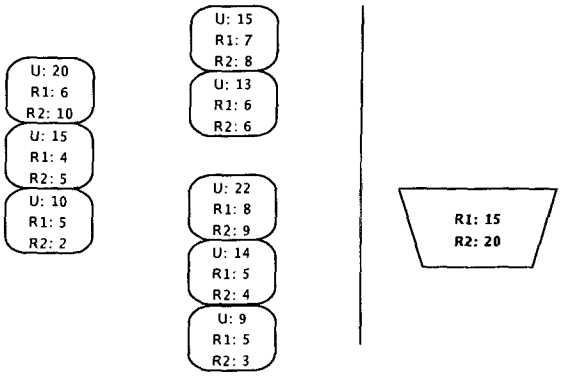

Figure 2.2. *A MMKP Knapsack Problem.*

25%, 8% and 50% for an enhanced level. It would likely offer a lower amount of utility for the base level than the enhanced one, although this is not a necessary condition. Several different users might present SLAs. The *Admission Control Problem* is to determine which SLAs get admitted at which QoS levels to maximize the total utility of the system.

To solve the admission control problem, the Utility Model uses a *Multi Dimension, Multiple-Choice Knapsack Problem* (MMKP). The knapsack problem is a classic problem in computer science. Given a *knapsack* with a volume constraint, and several *stones* or *items*, each with a volume and a weight, the problem is to select items to place in the knapsack such that the weight is maximized while the volume constraint is not violated. In other words, choose the heaviest subset of items that fits in the knapsack.

This classic version of the problem is extended for use in the Utility Model. Instead of a single volume constraint, there is a vector of constraints, and each item specifies a value for each resource constraint in the vector. Additionally, the stones are organized into separate groups, and at most one item from each group can be chosen. There is still a single weight value for each item, and the problem remains to maximize the total weight while observing all of the constraints.

This version of the problem easily maps to Utility Model SLAs. Weight corresponds to Utility, and the multiple constraints are each one of the resource constraints in the system.

2.3 SLAOpt - The Utility Model Applied to Network Admission Control

Watson et al. [29] first applied the Utility Model to Network Admission Control. Here, the resources under consideration are the bandwidths available on each of the links in a network. Users wish to be provided with a certain bandwidth between two points. Multiple bandwidths may be specified, representing different levels of QoS. Additionally, users can specify a delay constraint for each level of QoS which states that the total delay along the path between the two points cannot exceed a certain value. This model ignores jitter bounds.

If we consider a practical network as a graph, multiple paths between the two points (nodes) can exist. Explicit routing is assumed, effectively transforming the Network Admission Control Problem to a Call Admission problem. Each available path can be paired with different QoS levels to create a series of Admission Profiles for the Utility Model. A given path can be paired with a given QoS level only if it meets the delay constraint for that QoS level. Thus, the Admission Profiles consist of bandwidth requirements for a subset of the network links (the bandwidth along a single path) and the utility for that QoS level. A given path may be appropriate for several QoS levels.

Thus, network admission is mapped to a MMKP, and the techniques developed for the Utility Model can be used to solve it for maximum utility.

The Utility Model is used to determine which sessions are admitted to the network. As it will not admit sessions in any combination that will use more than the available bandwidth on a given link, all admitted sessions are guaranteed to have the bandwidth they need for the QoS level they were admitted at. It is assumed that some form of traffic policing would be applied to ensure a given session does not operate so as to consume more than the bandwidth specified for the QoS it was authorized for.

Watson termed this application of the Utility Model *SLAOpt*, for *Service Level Agreement Optimizer*. A simulation implementation was developed to demonstrate proof of concept. This is discussed further in Chapter 4.

2.4 The Utility Model and Existing Network QoS Mechanisms

Several technologies have been proposed to meet the data network QoS requirements discussed above. For example, The Internet Engineering Task Force's (IETF's) DiffServ (Differentiated Services [30]) and IntServ (Integrated Services [26]) provide mechanisms for specifying traffic classes and corresponding behaviour. IP tunnelling or MPLS (Multi-Protocol Label Switching [4]) can both be used to achieve explicit routing, a form of circuit switching. RSVP (Resource Reservation Protocol [27]) is a signalling protocol independent of the underlying QoS mechanisms.

All of these are proposals and technologies which have been several years in the making, and may see deployment and use in the future. This raises the question: How will the proposed Utility Model based Network Admission Controller would work with such technologies? It is important to see that Admission Control is a distinct activity that can be independent and complementary to the actual underlying methods used to provide QoS.

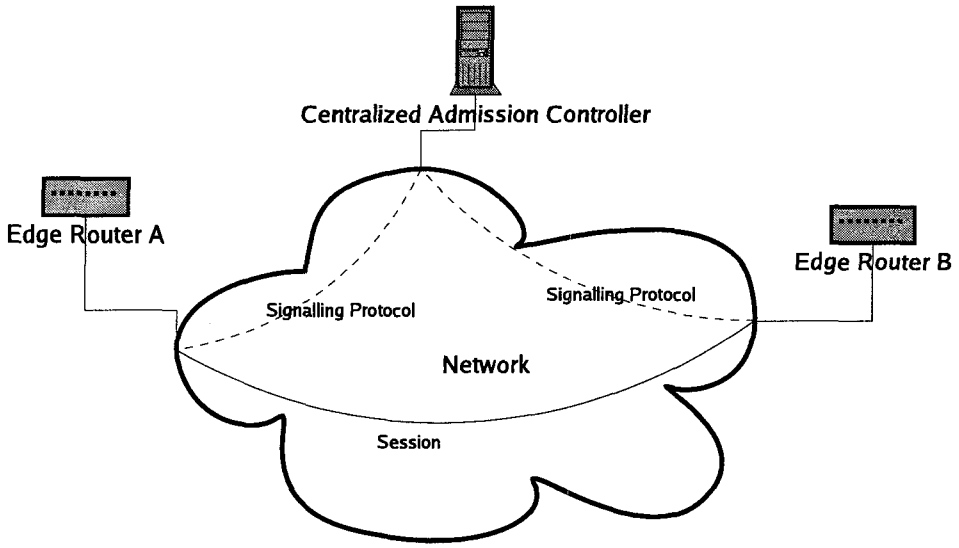

Figure 2.3. *A Centralized Admission Control Scheme.*

For example, we might have an IP network utilizing MPLS for explicit routing. Application sessions would seek admission to this network through the use of a signalling protocol some signalling protocol. These requests could encapsulate all of the information included in the conceptual SLA used in SLAOpt. Internal to the network, a centralized admission controller, aware of the status and current commitment of network resources, could use the Utility Model to make the admission decision, and then use the signalling protocol to set up the appropriate MPLS path and additional per-hop behaviour, as well as inform the application of its admission status. Figure 2.3 shows this architecture.

The addition of an admission controller ensures that network resources are not over-committed. This avoids congestion and its associated impact on QoS. It also allows efficient heuristics to be applied to resource allocation. In evaluating an admission controller, it is necessary to consider both of the measures of performance previously introduced - solution quality and decision time.

The most basic of admission controllers would work on a First-Come, First-Serve basis, admitting a session immediately if there were sufficient resources. The Utility Model based

controller would likely batch admission requests over a given epoch or time interval and then run its heuristic to determine an efficient assignment of network resources amongst the batch of requests accumulated throughout an epoch. However, this would come at the price of a delay in admission to the network. The tolerable delay would depend on the application, and may be close to real time in some cases. As long as the instantaneous batch size was small enough, this could be handled by the controller.

Chapter 3

Solving the Multiple Choice Multi-Dimension Knapsack Problem

The Multiple Choice Multi-Dimension Knapsack Problem (MMKP) lies at the core of the Utility Model. The majority of the computational work for a Utility Model controller is in finding a solution to this problem. Therefore, we present a brief discussion of algorithms and heuristics for the MMKP.

Utility Model research has resulted in a variety of techniques being developed for finding a MMKP solutions. The first two presented date from Khan et al.'s [31] original work. Following these, the techniques employed in Watson et al.'s [29] network orientated work are discussed. Finally, Akbar et al.'s [18] primary contributions are presented, along with some other approaches which that work explored.

It is important to draw a conceptual boundary between the methods used to solve the MMKP, and policies or procedures developed for a particular problem application. The MMKP is a general mathematical problem. An application, such as Network Admission Control, simply introduces restrictions imposed by the physical nature of the network and the traffic, on formulating the MMKP instance that needs to be solved.

3.1 BBLP - An Exact Solution

The MMKP is an NP-Hard problem. In practical terms, this means that an exact solution to the problem cannot be computed in a reasonable amount of time, unless the problem instance is relatively small. One exact solution algorithm, Branch and Bound with Linear Programming (BBLP), was presented by Khan [31].

As the name suggests, the BBLP solution is based upon classic branch and bound techniques. A solution search tree is explored, and branches are chosen based upon how close they move the solution to a computed upper bound on the actual solution value. In this case, choosing a particular branch corresponds to choosing an item from one of the groups in the MMKP. Linear programming techniques are used to compute the upper bound for each of the possible branches.

A more detailed explanation of BBLP can be found in Khan's work [31]. Akbar [18] showed that this algorithm's running time did indeed increase exponentially with the size of the MMKP problem, and quickly became impractical for practical applications.

3.2 HEU - *Original Heuristic*

As with any NP-Hard problem, computing a solution for a large problem instance requires a heuristic. Khan [31] presented the first such heuristic for use in the context of solving an MMKP for the Utility Model.

Termed simply HEU (for Heuristic), this approach uses the concept of *aggregate resource usage* to choose items from the groups of the MMKP. Starting with the lowest item of each group selected, the item with the greatest negative gain in aggregate resource usage is selected to be upgraded. This can be thought of as an item that will reduce resource usage while not affecting utility earned. If no such item exists, the item with the highest utility gain per unit of increased aggregate resource usage is selected for upgrade. This is repeated until no further upgrades are feasible (i.e. any further upgrade will violate resource

constraints).

Both Khan [31] and Akbar [18] provide some computational analysis of this heuristic and its comparative performance to the BBLP algorithm. The worst case time complexity is:

$$O(mn^2(l-1)^2) \tag{3.1}$$

Where:

$$\begin{aligned} n &= NumberofGroups \\ l &= NumberofItemsperGroup \\ m &= ResourceDimension \end{aligned}$$

3.3 Watson's Heuristics

Watson [29] first applied the utility model as a network admission controller. He explored the effect the constraints introduced by a network environment had on the problem. The majority of his work focused on these effects and the process of formulating Network Admission Control as a Utility Model problem.

One primary contribution here was in exploring multiple suitable paths in the network and matching these with QoS levels. This was a process of finding different combinations of resources that would support the requirements of a given SLA.

Watson also explored the ways in which user requests were presented to the controller. He postulated that computational work could be saved by iteratively considering an incoming batch of new SLAs:

1. On their own.
2. Together with admitted SLAs that have overlapping resource requirements.

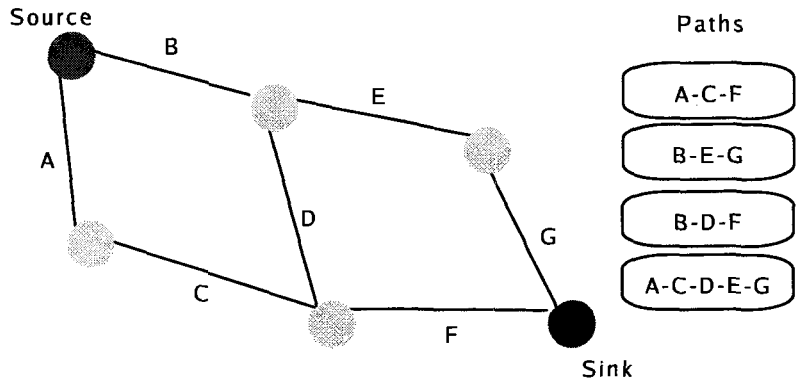

Figure 3.1. *Multiple Paths in a Network*

Effectively, this means the system first tries to admit new SLAs only with currently available bandwidth. If that fails, it tries to admit new SLAs by rearranging the resource consumption of *nearby* admitted SLAs, so as to free bandwidth on key links. This could be done with or without permitting QoS changes to the admitted SLAs.

In his model implementation, the criteria for *nearness* of SLAs was simply sharing one or both end points. Two admission rounds were performed for a new batch. The first considered just the new batch, the second considered SLAs rejected in the first round together with admitted SLAs considered *near* to these.

At its core, Watson's work used a slightly modified version of Khan's HEU. These modifications mostly involved the ordering of QoS levels and paths when considering possible upgrades.

3.4 MHEU & IHEU - Akbar's Heuristics

Akbar[18] further modified and refined the original HEU heuristic. Modified HEU (MHEU) involves two functional changes. First, if the original solution is infeasible, MHEU will search for a feasible solution rather than failing (as HEU would). Second, and more significant, it attempts to escape from local maxima in the solution space. It does this by attempting an upgrade that would normally be infeasible, followed by one or more down-

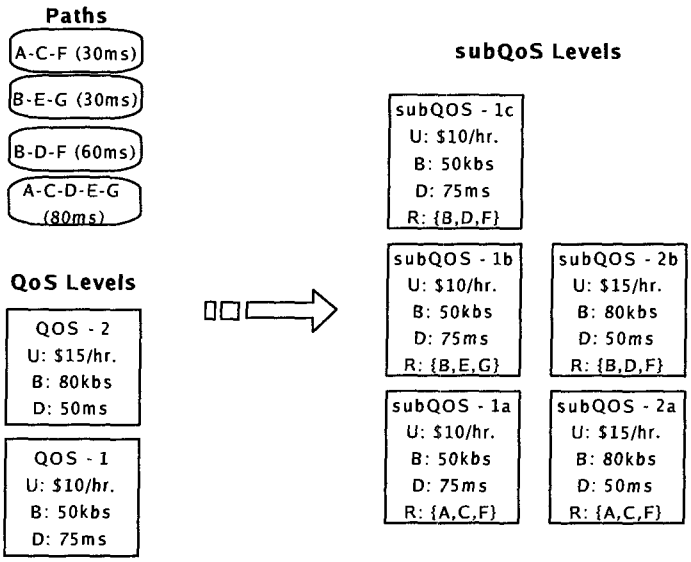

Figure 3.2. *Combining paths and QoS levels in MHEU.*

grades to make the solution feasible again.

IHEU (Incremental Heuristic) is essentially MHEU applied to an existing partial solution of an MMKP instance. The idea is that over time, in an admission controller, the input to the MMKP will consist of some sessions already admitted at a given level and other new sessions not admitted at all. Some previously admitted sessions will have been removed (i.e. finished). Instead of solving the entire problem with all current sessions reset to their lowest level, IHEU attempts to improve the solution from the current admission state. It runs the MHEU heuristic with this problem state as the initial solution.

MHEU and IHEU were both used in the network context introduced by Watson [29]. To handle multiple paths suitable for a given QoS level, they translate one QoS level into several *SubQoS* levels (Figure 3.2). Each SubQoS level offers the same utility but has a resource requirement vector corresponding to one of the suitable paths. Thus, from the MMKP's perspective, multiple paths simply increases the number of items available in a given group. This was a refinement of Watson's work in this area.

The effectiveness of Akbar's local maxima escape techniques, in terms of the compu-

tation cost they incur and benefit they yield, is investigated later in this work. The MHEU heuristic is used in most of the experiments presented in this work.

Akbar performed some basic performance analysis, focused on how MHEU and IHEU performed compared to the optimal solution found by BBLP. Results indicated these heuristics did rather well, as they generally attained utility above 90% of that found by BBLP.

3.5 Other Approaches

Akbar considered a few other approaches to generating MMKP solutions. These included a Greedy heuristic, an application of Moser's heuristic and a Convex Hull approach. Further details can be found in [18].

Chapter 4

Network Admission Controller Design & Implementation

4.1 Model Implementation Requirements

In order to perform a meaningful evaluation of Utility Model based Network Admission Control, a satisfactory implementation was required. Since the focus of this evaluation is the performance of the controller heuristic, a Model Implementation controlling a simulated networking environment would be sufficient.

A number of specific requirements were identified for this Model Implementation:

- It must allow *meaningful* performance data to be gathered.
- It should show some performance improvement over previous work by optimizing coding style.
- It should be implemented utilizing good software engineering practises to establish a clean code-base for instrumentation and for continuing research.

Meaningful performance data in this context refers to data which accurately reflects controller performance. This means the Model Implementation must provide easy manipulation over independent variables with clear and accurate observation of dependant variables, and that interference with observations from underlying Operating Systems, Virtual Machines or other factors should be minimized.

4.2 Previous Work: SLAOpt

At the beginning of this work, a previous Network Admission Control simulator based on the Utility Model had been implemented. SLA Optimizer (SLAOpt) originated with Watson's work [29], as previously discussed. It was intended to demonstrate the feasibility of using Utility Model based Network Admission Control.

Below the design of SLAOpt is discussed and some analysis of its suitability for this performance evaluation presented.

4.2.1 SLA Interface

The purpose of SLAOpt was to control access to a simulated network. To accomplish this, SLAOpt defined a format for Service Level Agreements (SLAs) submitted by potential users of the network. These were to conform to an XML formatted document. Each SLA defined a source and destination node from the network, and several QoS levels. Each QoS level included required end-to-end bandwidth, maximum tolerable end-to-end delay and offered utility for that service level. Jitter or packet drop tolerances are not specified.

Of course, this definition was a simplification of what an actual application might require. The question of how to map user preferences or requirements into actual numerical values for system resource requirements is quite complex and was considered beyond the scope of SLAOpt. Recall that this implementation is intended to show proof of concept, and for that this definition was sufficient. Similarly, it is considered sufficient for our performance evaluation of a utility model controller. It is assumed a mapping will be performed externally, from application requirements to these (or similar) QoS parameters.

Additionally attention should be drawn to the fact that this interface provides only for point-to-point communication. It did not provide for multicast communications. Multicast is important, as it can greatly improve the efficiency of network resource usage, which is also the purpose of using the Utility Model for admission control.

4.2.2 Network Modeling

SLAOpt modeled the network as a directed graph. That is, links were mapped to edges and routers or interconnection points to nodes in the graph. Each link was assigned two values: A maximum capacity (bandwidth) and a constant delay value. Each link also had a single start and end point. No forwarding delay is associated with the nodes. Router delay could be modeled by adding additional links representing each router with an ingress and egress node. The network topology was assumed to be static - that is, without failures or capacity upgrades or downgrades.

A directed graph implies uni-directional links. However, a directed graph can be used to model bi-directional links simply by specifying two edges for each link.

The network is also assumed to support explicit routing. The Admission Controller can specify an explicit route for a given session, and the network will route all traffic from that session along that route. This makes the Network Admission problem similar to *Call Admission*, the problem of admitting voice calls to a circuit-switched voice data network. Explicit routing allows the controller explicit control over the allocation of the network resources as they are modeled.

For the requirements of our performance evaluation, this simulation model was considered adequate. The response of the Network Admission Controller to various failure modes is beyond the scope of our work, but is covered in related works [16].

4.2.3 Controller Structure

SLAOpt was designed for direct interaction with a single user. An extensive Graphical User Interface (GUI) was developed to allow interaction with the Network Admission Controller. The controller maintained state internally, and the user could select batches of SLAs (stored in XML files) for admission, and then observe the resulting load on the network and the utility generated. SLAs could also be removed to simulate session expiration, or inspected to determine their current operating parameters.

This approach allowed a user to exercise very fine grained control and was excellent for demonstrating the concepts involved. However, it was not designed for the type of lengthy, repetitive and statistically sound testing required for a thorough performance evaluation.

4.2.4 SLAOpt Implementation

The original SLAOpt implementation was written in Java. Java provides a very intuitive, object oriented environment with an extensive library of GUI constructs. Thus, it is ideal for prototypes such as SLAOpt, and for rapid development of a variety of applications.

There are two primary shortfalls evident in the SLAOpt implementation, from our point of view of performance analysis and improvement.

First, the implementation is very specific to prototyping and proof of concept. Classes handling actual admission control computation were often not separated from those controlling the GUI. The utility model core was also written specifically for the chosen network model and SLA interface, thus limiting re-usability or extensibility. Generally, the coding style was difficult to follow and maintain. The SLAOpt prototype had undergone several changes by different parties since Watson's original, each exploring different theoretical aspects of the Utility Model admission control concept. Thus, the focus was more often on achieving functionality rather than efficient operation.

The second point is the choice of Java itself as an implementation language. Although an excellent choice for prototyping, Java still lags behind languages such as C in performance. Additionally, the requirement for a virtual machine and garbage collection, etc, force additional variability into the situation when attempting to measure or improve performance. As stated, we wish to provide *meaningful* performance data by eliminating such obfuscations.

4.3 Utility Model Network Admission Controller - Model Implementation Re-Design

From the above discussion it can be concluded that the original SLAOpt was lacking in some of the key requirements identified for our performance evaluation. This presented a choice between working with and improving the existing code, or starting from scratch with a clean design.

The latter approach was chosen for a variety of reasons. It would allow the implementation to be focused on the goals of this research rather than previous work. A cleaner, more modular design than SLAOpt's could be established, which would be reusable for future work. It was felt the goals of this research would be better served by a fresh implementation.

This did not mean abandoning all of the work done in SLAOpt, however. The network model and SLA interface were preserved with minor modification. And the core of the work, the Utility Model, would use previously developed heuristics. The primary changes were in the structural design of the software implementation.

To this end, the following design was created. It seeks to separate the Utility Model core from higher level abstractions (such as the SLA interface), allowing the core to be reused with different models, and vice versa. The top level interface and model is mapped to this core through a translation layer. Although this adds some computation, it creates a far cleaner design with reusable components. Additionally, performance profiling would allow the cost of this translation to be separated from that of the actual admission calculations in the Utility Model core.

4.3.1 Utility Model Core - The MMKP Solver

This is the core of the design, where a heuristic is applied to solve an MMKP problem. The interface to the utility model core is a simple function call, taking an instance of a MMKP

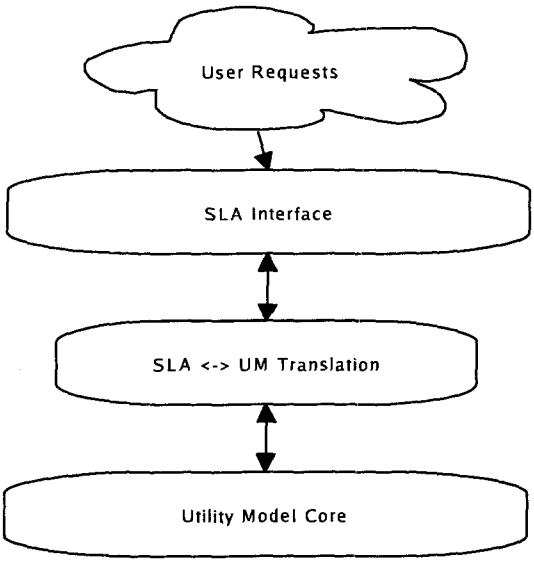

Figure 4.1. *New Network Admission Controller Design.*

as its argument. Thus, it could be used with any application which can present its problem as an MMKP.

The generic MMKP is represented as a data structure with multiple groups, each group containing one or more items. An item contains a utility value and a list of resource requirements. In the SLAOpt context, a group maps to an SLA and an item to a QoS level of that SLA, with the resource requirements for a given item mapping to the required bandwidth on each of the links in a path through the network which would satisfy that SLA.

The heuristic used in this Model Implementation was Akbar's MHEU[18]. However, any heuristic for solving an MMKP could be substituted, using the same MMKP interface to the core module.

The core module does not maintain admission state, i.e. it does not remember which groups are admitted and which are not. It simply takes the given list and attempts to improve upon the current solution by applying the heuristic. State would be maintained by whatever entity was calling the core module.

The core module is the focus of this performance analysis. However, an external, SLA

based interface is used to place this evaluation in the appropriate context.

4.3.2 Service Level Agreement Interface & Network Model

The SLA and Network model were generally the same as those in the original SLAOpt, but with some changes.

The SLA XML definition was modified to include current state information. This included the currently active QoS level and the path used to fulfil that service level. This allowed the admission state of the system to be stored in the XML document rather than inside the admission controller.

An additional XML definition was created to describe the network model. This included a list of all nodes in the network, and a list of all links. The nodes contained only labelling information, while the links contained labels, source and destination node labels, total capacity and delay metrics.

Both these XML objects were incorporated into an *Admission State Document*, which included all the SLAs currently admitted or seeking admission, along with the network those SLAs were associated with. An example of such a document is shown in Figure 4.2. The format is defined in Appendix B.

Thus, the Model Implementation controller could be seen as a simple procedural invocation, which would take an XML document as an input, attempt to improve the admission solution, and then return an updated XML document. The calling party would be responsible for maintaining state between calls to the controller if so desired. This would likely be a software layer designed for a specific application.

This approach was better suited than the original stateful controller to the evaluation and testing of the controller. Different interfacing approaches, such as batching admission requests, could be performed outside of the controller.

The Network and SLA models used in this Model Implementation make the same assumptions as SLAOpt.

```
<?xml version= '1.0' ?>
<admission_state>
<sla id = "0"
     source = "0"
     destination = "1"
     active_profile = "-1">
  <qos  capacity = "84000"
        delay = "0.9"
        utility = "100"></qos>
</sla>
<sla id = "1"
     source = "0"
     destination = "1"
     active_profile = "-1">
  <qos  capacity = "83000"
        delay = "0.9"
        utility = "100"></qos>
</sla>
<network>
  <node id = "0" name = "Node 1"></node>
  <node id = "1" name = "Node 2"></node>
  <link id="0" source="0" destination="1"  capacity="100000" delay="0.100000"
></link>
</network>
</admission_state>
```

Figure 4.2. *Sample XML Admission State Document.*

4.3.3 Translation Layer

As described, the utility model core was not written specifically for the SLA and network model used at the interface layer. Some intermediate translation is necessary, to convert the admission problem described by the SLAs and network model into a MMKP that the core heuristic can handle. This would be true of any application using a Utility Model implementation at its core. The problem must be translated into the appropriate format.

In the context of this SLA and network model, the primary step in this process is assigning paths to an SLA. Multiple paths through the model network may be capable of satisfying a given SLA's QoS levels. The translation layer in our implementation contains a routing algorithm which computes a number of lowest cost paths from an SLA's source to destination. The cost function used is the delay metric for the links, thus these are the lowest delay paths.

The number of paths to compute is given as a parameter to the controller. This tells the controller how many paths to generate and consider for each SLA. For each QoS level in

each SLA, each path is checked to determine if it can meet the end-to-end delay constraint. If it can, an item is added in the corresponding MMKP group using the links in that path and that QoS level's parameters. Hence, a given QoS level in an SLA may map to multiple items within a MMKP group, if multiple paths are capable of satisfying that level's delay constraint.

$$Groups\{\} = Paths\{\} \; X \; QoSLevels\{\} \tag{4.1}$$

When the Utility Model heuristic obtains a solution, the SLAs corresponding to MMKP groups must be updated to indicate the current QoS level and the path used to provide this service.

Martins' algorithm [5] for finding the Kth shortest loopless paths in a directed graph was used to find the paths for this implementation. Other algorithms could be used, and it may be interesting to explore the use of non-shortest paths as a means to better distribute load.

As mentioned, this implementation assumes that the controller can specify explicit routes for traffic in the network. Depending on the underlying network design, this may or may not be reasonable. Determining the resources which a given SLA will require might rely on some underlying routing mechanism. For example, in a traditional OSPF routed network with static routing tables, the admission controller could determine whether the one available path for two given end points met SLA requirements at different QoS levels. An MMKP group could then be created appropriately. The worst case would be an underlying network where routing tables change often and independently of the admission controller. Such a network would be difficult to guarantee QoS for in any case. Thus, the necessity of guaranteeing resources for the lifetime of a session implies that a circuit switching style model be used.

4.4 Implementation

C was chosen as the implementation language for the Model Implementation of the Utility Model based Network Admission Controller. This choice was made for several reasons.

It avoided the virtual machine and other overhead involved in Java (i.e. garbage collection). It was felt a simpler procedural language was more appropriate for a performance evaluation than a higher level object-oriented one. C is the likelier choice for an industrial deployment of a Network Admission Conroller. Finally, several free, stable and proven tools are available for developing and analyzing programs written in C (specifically, the gcc GNU C compiler [36] and the gprof GNU profiler [14]).

In coding the design, great importance was attached to maintaining the modularity described. Additionally, care was taken to ensure that a good and efficient coding style was used. Several performance improvements were made over the existing codebase simply by taking additional care with things such as loop termination conditions and data structure creation and maintenance.

Chapter 5

Network Admission Controller Performance Evaluation Methodology

5.1 Experimental Process

In any experimental process, there are independent and dependent variables. Independent variables are those the researcher manipulates. Dependent variables are those the researcher observes for the effects of these manipulations.

Evaluating a Network Admission Controller is no exception to these fundamental principles. In the previous chapter, a new Model Implementation of a Utility Model based Network Admission Controller was presented. This chapter discusses the methodology used to evaluate the performance of this controller.

Recall that performance can be characterized by two general metrics:

1. *Solution Quality* - How good is the solution? i.e. How close to the optimal utility does the solution produce?
2. *Runtime* - How long did it take to arrive at a solution?

These are, generally, the dependent variables for our performance evaluation experiments.

The independent variables include all possible inputs to the system. These are:

1. User Requests (i.e. SLAs).

2. Network Parameters (i.e. network topology).
3. Controller Parameters (i.e. the number of paths to find in our Model Implementation Controller).

In order to perform a valid evaluation, we need to map these general variables to the specific variables provided by the controller implementation under observation. Next, a series of experimental plans dictating the manipulation of the independent variables is required. This will also imply how the observations are presented. To be truly thorough, experiments should measure the response of the system to all of the available independent variables. Depending on the system, this can imply a very large number of experiments. Constraints can be added based on reasonable assumptions to keep the process tractable. That is, the ranges of values which may occur in a real application can be applied to constrain the inputs.

This chapter discusses the mapping of the above general variables to those specifically made available by the Model Implementation of the previous chapter.

The three following chapters discuss the experiments that constitute our evaluation.

5.2 Independent Variables

Here we discuss the input variables provided by our Model Implementation and how they can be manipulated.

5.2.1 User Request Parameters

The Model Implementation represents user requests as SLAs. These requests are presented to the controller in a batch in the Admission State Document submitted. This document includes both SLAs requesting admission and those representing already admitted sessions. The parameters of this batch of requests are summarized in Table 5.1.

Several of the parameters in Table 5.1 are variables that each SLA or each QoS level

Table 5.1. *Model Implementation SLA Input Batch Parameters.*

Parameter	Type	Description
Batch Size	Single-Value	The number of requests in the batch.
Number of QoS Levels	Single-Value	The number of QoS levels for each SLA.
Source - Destination	Distribution	The source and destination for each SLA.
Bandwidth	Distribution	The bandwidth requested by each QoS level of each SLA.
Delay	Distribution	The delay constraint requested by each QoS level of each SLA.
Utility	Distribution	The utility offered by each QoS level of each SLA.

will specify. Thus, an experimental process needs to consider the statistical distribution of the values used for these variables.

Runtime is expected to be primarily affected by batch size and the number of QoS levels per SLA, as these directly affect the size of problem considered by the MHEU heuristic and the number of decisions it must make.

These two variables will also affect solution quality by affecting the flexibility which the controller has to seek optimality.

The delay constraint could impact runtime by limiting the number of paths available to service a given SLA, again affecting problem size.

Source - Destination, Bandwidth and Utility are not expected to significantly affect runtime, but will affect the quality of the solution. This should be qualified by saying they could affect the runtime if they have values that drastically simplify the problem. For example, if all SLAs had bandwidth requirements exceeding the maximum link bandwidths, no feasible solution would be possible and the MHEU heuristic would quickly finish.

5.2.2 Network Parameters

Network parameters are the information about the network which characterize it for the Admission Controller. In the case of our Model Implementation, these are represented in the network model used. The topology of the network to be considered is included in the Admission State Document submitted to the controller.

It is difficult to explicitly label and manipulate the variability represented by the network topology. The size and topology of the network could both affect how well the Admission Controller performs.

Although we cannot simply specify a range of values for network topology, we can still manipulate it systematically. Several approaches are available for an experimental process. One is to specify a network model which will minimize interference with all other independent variables, so that their influence may be evaluated. Another is to constrain topology characteristics to those expected in real-world application environments, effectively using models of real networks. Both these approaches are used in our experiments.

Topology can be expected to impact both solution quality and runtime. The size of the network will determine how many resources must be considered, affecting computation time. The structure of the network will determine whether any choke points are present that may cause the network to be under-utilized, or whether a large number of redundant paths provide improved flexibility.

5.2.3 Controller Parameters

The only controller parameter provided by the Model Implementation is the limit on the number of paths provided for each SLA. This is a simple numeric parameter.

The number of paths considered directly affects problem size, as each QoS level of each SLA may have to be considered with this number of paths. As this affects the number of choices the heuristic considers, it will have an impact on runtime.

The number of paths may also affect solution quality through the flexibility the con-

Table 5.2. *Summary of Solution Quality Related Dependent Variables.*

Variable	Description
Utility	Measures how well controller is meeting its objective function.
SLAs Admitted	Measures how many SLAs out of a batch have been admitted.
Resources Consumed	Measures the amount of network bandwidth consumed.

troller has in assigning resources to SLAs.

5.3 Dependent Variables

Here we discuss the mapping of the general dependent variables presented for admission control to specific variables observable in the Model Implementation.

5.3.1 Solution Quality

The primary measure of solution quality is the utility attained by the admission controller. This shows how well the controller is meeting its objective function. The higher the value, the better the solution.

The number of SLAs admitted from a given batch are included here. Although this does not represent solution quality in the same way as utility, it does provide insight into the nature of the solution obtained.

Similarily, the amount of bandwidth committed by the admission controller is a useful metric for showing how efficiently the network is being utilized.

Observing these values in combination provides a good view of solution quality. Representing them relative to there respective maximum values is also useful. For instance, utility obtained can be compared to the total utility offered, SLAs admitted to the total batch size and resources consumed to the maximum resources available.

5.3.2 Runtime

Runtime is a fairly straight forward variable. It is simply the amount of time taken by the admission controller to arrive at a solution. By using a profiling tool, we can significantly expand the amount of information that can be observed relating to runtime.

For example, we can observe the runtime on a per function call basis, determining which parts of the implementation code the controller spends most of its time in. The runtime presented in the results sections for the experiments run on the Model Implementation is strictly the time required to solve the MMKP by the heuristic. This represented the vast majority of the total time taken by the controller.

To obtain this information from our Model Implementation, we use the GNU Profiler (gprof [14]). Example gprof output is shown in Figure 5.1. This output was generated and appended to the Admission State Document output by the controller for further analysis.

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 65.27      1.71     1.71  7744008     0.00     0.00  check_upgrade_feasibility
 16.22      2.13     0.42  3458055     0.00     0.00  calculate_change_in_aggreg
ate_resource_consumption
 14.89      2.52     0.39     1121     0.00     0.00  do_feasible_upgrade
  1.91      2.58     0.05      850     0.00     0.00  quick_sort_links
  0.57      2.59     0.01        1     0.01     0.01  check_solution_feasibility
  0.38      2.60     0.01     5925     0.00     0.00  calculate_scaled_change_in
_resource_consumption_available
  0.38      2.61     0.01      850     0.00     0.00  find_shortest_path
  0.19      2.62     0.01     1970     0.00     0.00  consume_resources
  0.19      2.62     0.01     1970     0.00     0.00  free_resources
  0.00      2.62     0.00     8434     0.00     0.00  check_duplicate_paths
  0.00      2.62     0.00     7906     0.00     0.00  copy_path
  0.00      2.62     0.00     7056     0.00     0.00  find_cycle
  0.00      2.62     0.00     6246     0.00     0.00  build_shortest_path
  0.00      2.62     0.00     6115     0.00     0.00  forms_cycle
  0.00      2.62     0.00     5396     0.00     0.00  join_paths
  0.00      2.62     0.00     1044     0.00     0.00  calculate_scaled_change_in
_resource_consumption_overconsumed
  0.00      2.62     0.00      850     0.00     0.00  find_paths
  0.00      2.62     0.00      850     0.00     0.00  sla_to_xml
  0.00      2.62     0.00        1     0.00     2.62  admit_slas
  0.00      2.62     0.00        1     0.00     0.00  build_resources
  0.00      2.62     0.00        1     0.00     0.06  build_sessions
  0.00      2.62     0.00        1     0.00     0.00  cache_session_state
  0.00      2.62     0.00        1     0.00     0.00  do_downgrade
  0.00      2.62     0.00        1     0.00     0.01  do_non_feasible_upgrade
  0.00      2.62     0.00        1     0.00     0.01  find_feasible_solution
  0.00      2.62     0.00        1     0.00     0.00  generate_xml_admission_sta
te
  0.00      2.62     0.00        1     0.00     0.00  getdoc
  0.00      2.62     0.00        1     0.00     0.00  network_to_xml
  0.00      2.62     0.00        1     0.00     0.00  parse_network
  0.00      2.62     0.00        1     0.00     0.00  parse_sla_list
  0.00      2.62     0.00        1     0.00     0.00  parse_xml_admission_state
  0.00      2.62     0.00        1     0.00     0.00  restore_cached_session_sta
te
  0.00      2.62     0.00        1     0.00     2.56  solve_knapsack
```

Figure 5.1. *Sample of gprof output used to measure Controller Runtime.*

Chapter 6

Experiments - Controller Validation

6.1 Experiment Purpose

Before meaningful performance evaluation could be performed it was necessary to establish confidence that the Model Implementation was operating correctly. A series of validation experiments was devised to do so.

6.2 Experiment Design

Each of the experiments in this series attempts to validate the Model Implementation by running a test for which the solution is known. If the known solution is not returned, then the controller is not functioning as expected. Conversely, a correct result builds confidence in the controller's functioning.

These tests are specific to the heuristic used in the Model Impelmentation, namely MHEU. The selection of these particular validation tests is also based upon knowledge of the internal workings of this heuristic.

Each validation experiment in this series is presented with its defining parameters and results. Significant conclusions that can be drawn from these results are also discussed.

Note that the implementation of shortest-path algorithm (Martins [5]) used in the Model Implementation was independently verified.

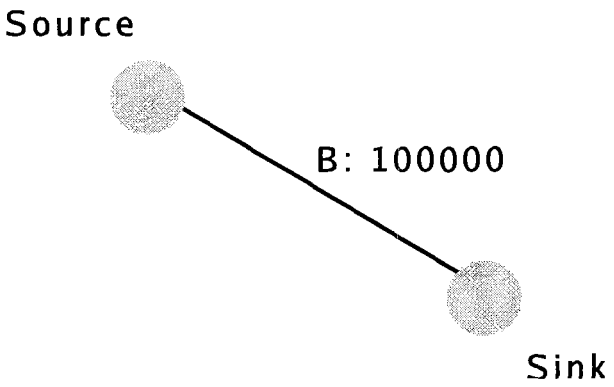

Figure 6.1. *Simple network used in validation experiments.*

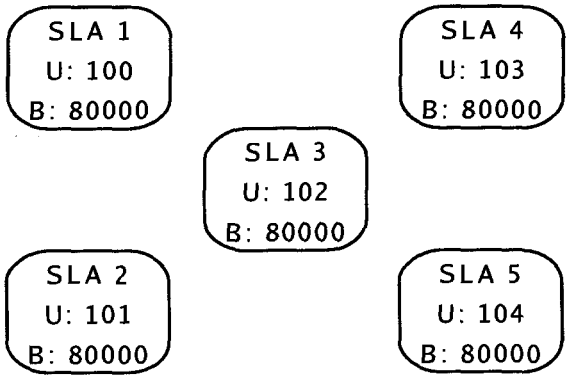

Figure 6.2. *Admission validation with monotonically increasing utility.*

6.3 Experiment - Very High Contention Validation

6.3.1 Parameters

This test attempted to validate the behaviour of the admission controller in selecting the best SLA for admission out of a batch, when resources were sufficient to admit only one SLA.

A test network with a single link and two nodes was used (Figure 6.1). All SLAs requested a session on this link. The SLAs each had one QoS level. SLA parameters were arranged so that no more than one SLA could be admitted.

Two simple tests were conducted.

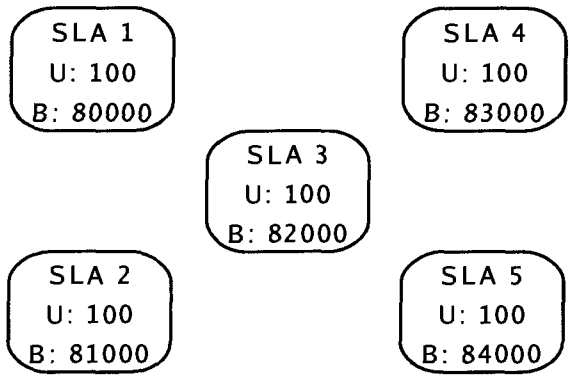

Figure 6.3. *Admission validation with monotonically increasing resource requirements.*

In the first (Figure 6.2), SLAs numbered *1-5* offered monotonically increasing utility for the same amount of bandwidth on the link. The expected result was that *SLA 5* would be selected, as it offered the highest utility per unit resource.

In the second test (Figure 6.3), SLAs numbered *1-5* offered the same utility for a monotonically increasing amount of bandwidth on the link. The expected result was that *SLA 1* would gain admission as its utility to resource ratio was highest.

6.3.2 Results

Both test cases performed as expected. From this we can conclude the fundamental behaviour of the admission controller is correct. That is, it will pick the best SLA for an empty link, according to the utility to resource ratio, to upgrade the solution at any given point. This fits the desired behaviour of the heuristic.

6.4 Experiment - Very Low Contention Validation

6.4.1 Parameters

The purpose of this test was to show that the admission controller would not fail to grant admission to all SLAs at their maximum QoS level if it could do so.

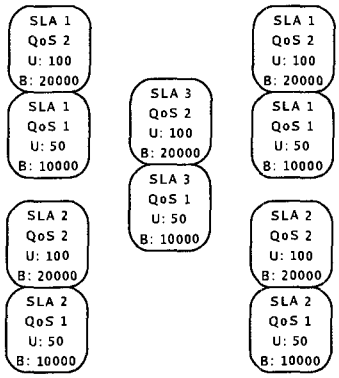

Figure 6.4. *Low contention validation SLA batch.*

The same network configuration as the previous test was used, and a batch of SLAs constructed such that all could be accommodated at their maximum QoS level without violating the bandwidth constraint (Figure 6.4).

6.4.2 Results

As expected, all the SLAs in the test batch were admitted at their maximum QoS level.

6.5 Experiment - Pre-Admission State Behaviour Validation

6.5.1 Parameters

This experiment was designed to test the controller's behaviour when presented with an existing solution (i.e. some SLAs already admitted). The single link network was used once again in this experiment.

Two cases were designed. The first presented a set of SLAs admitted at their lowest QoS level. Each SLA had an additional QoS level which offered more utility for greater resource consumption (Figure 6.5). Parameters were specified such that only one could be

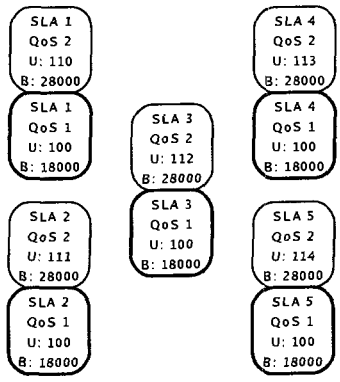

Figure 6.5. *Feasible existing solution, one feasible upgrade validation.*

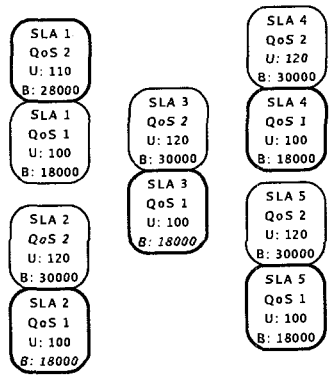

Figure 6.6. *Infeasible initial solution validation.*

upgraded. The desired behaviour was for the controller to perform the *best* upgrade.

The second test batch, shown in Figure 6.6, was created with an admission state that violated the resource constraints of the network. SLAs were structured with two QoS levels, and one was at its highest level which resulted in a violation of the resource constraint. Other second levels in the batch would not violate the constraint. The desired behaviour was for the controller to correct the invalid admission state.

6.5.2 Results

Both cases performed as expected. In the first, the SLA whose second QoS level offered the best utility to resource ratio was selected and upgraded.

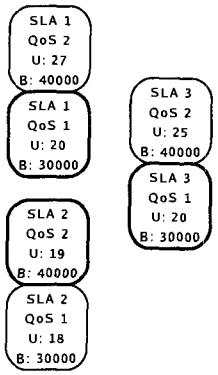

Figure 6.7. *Local maxima escape behaviour test.*

In the second, the least cost downgrade was performed followed by the best viable upgrade. This matched the theoretical behaviour for the heuristic.

6.6 Experiment - Local Maxima Escape Test Case

6.6.1 Parameters

This case was designed to validate the MHEU heuristic's local maxima escape behaviour. A test batch was designed that would utilize this routine.

The single link network was used once again, and SLAs were structured with two possible QoS states. All except one were set to their lowest state, but the solution was still valid (Figure 6.7). However, one of the upgrade states not selected would yield better utility than the one selected. The desired heuristic behaviour was for the currently upgraded SLA to be downgraded and the best upgrade selected.

6.6.2 Results

The controller performed as expected, first downgrading and then upgrading to the better solution.

Chapter 7

Experiments - Un-Constrained Heuristic Evaluation

7.1 Experiment Purpose

These experiments focus on determining the operational limits of the MHEU heuristic used in the Model Implementation. Parameter manipulation is performed with the goal of exploring mathematical limitations. Thus, independent variable values are not chosen to reflect an actual application environment. This is what is implied by *un-constrained* evaluations.

The focus of these experiments is on controller runtime. They aim to characterize the response of runtime to the various independent variables. Other dependant variables are also observed to provide additional insight into the controller's behaviour.

7.2 Experiment Design

In Chapter 5 the input parameters available for manipulation in the Model Implementation were introduced (Table 5.1). This sequence of experiments attempts to isolate several of these parameters and determine their impact on the implementations performance.

As exhaustively testing all of the parameters would be extraordinarily time consuming, constraints had to be placed on which parameters to focus on and on the values used. These experimental design decisions are presented below.

Table 7.1. *Manipulation of SLA Parameters for MHEU Evaluations.*

Parameter	Manipulation Description
Batch Size	Values selected through a range.
Number of QoS Levels	Values selected through a range.
Source - Destination	Random Uniformly Distributed selection process used.
Bandwidth	Determined by *Contention*. Individual values selected from Random Uniform Distribution in the range of the average value $+/-30\%$.
Delay	Held constant and set high enough to avoid limiting path selection.
Utility	Average value determined to hold batch sum constant. Individual values selected from Random Uniform Distribution in the range of average value $+/-30\%$.

7.2.1 User Request Parameters

Table 7.1 shows each of the parameters that can be manipulated in the input SLA batch and how it is manipulated in these experiments.

Contention is a value that was defined to assist in manipulating the average bandwidth of the SLA QoS sessions.

$$C = \frac{nB_{avg}P_{avg}}{B_{total}} \tag{7.1}$$

Where:

$$C = Contention$$
$$n = Batch\ Size$$
$$B_{avg} = Average\ Bandwidth\ per\ QoS\ Level$$

$$P_{avg} = Average\ Path\ Length$$
$$B_{total} = Sum\ of\ All\ Link\ Bandwidths$$

It is thus a measure of what fraction of the network resources a given SLA batch is requesting. This was considered more intuitive than simply varying average QoS level bandwidth directly.

Source and Destination nodes were uniformly distributed to minimize their impact on the problem. Their actual distribution would depend upon the application environment, whereas these experiments were concerned with general characterization.

The delay constraint was effectively ignored here as its only effect would be to reduce the number of paths the controller could consider for a given SLA. As the number of paths was being directly manipulated, this would be redundant.

Utility was also effectively a constant in these experiments. The average utility used for the uniform distribution was selected based on:

$$U_{avg} = \frac{10000}{n} \tag{7.2}$$

Where:

$$U_{avg} = Average\ Utility$$
$$n = Batch\ Size$$

Thus, the sum of the average utilities offered for any batch would be the same, and we could compare the achieved utilities of batches independent of other parameters. Note that the range within which utility values are statistically varied is also an application dependant variable, that could be linked to topics in economics such as pricing schemes. This is beyond our scope. Although the range of utility will have some impact on the decisions the heuristic must make, it will not affect the number of such decisions and thus not impact runtime.

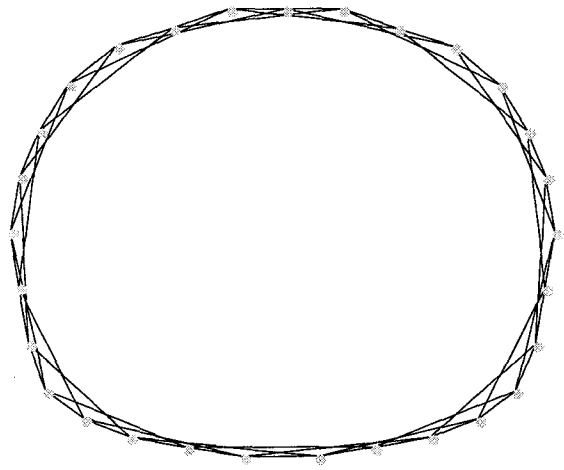

Figure 7.1. *The Augmented Ring Toplogy Used in Experiments.*

7.2.2 Network Parameters

The network model used seeks to minimize variation in the length and cost of paths available between any two nodes. It is a *29* node augmented ring structure, with each link having a bandwidth parameter of *100000* and a delay metric of *0.1*. Units are omitted as this is a mathematical evaluation. This topology was chosen as representative of the size of an enterprise network. Figure 7.1 shows the topology of the network.

7.2.3 Controller Parameters

The only controller parameter was the number of paths to consider for each SLA. This was varied from *1* to *5*. *5* was selected as a reasonable limit given the toplogy of the network.

7.2.4 Experimental Framework

A substantial experimental framework was implemented to allow these experiments to be automated and conducted systematically. This included a SLA input batch generator and several scripts for generating input, running the admission controller and collating results.

Each experiment in the series is presented. First, the precise parameters of the experi-

ment are defined. Then, experimental results are shown and discussed. Note that while no separate Contention characterization was performed, it was varied in several of the experiments.

7.3 Experiment - Batch Size Characterization

This test investigated the performance of the controller in relation to the number of SLAs in the admission batch.

7.3.1 Parameters

Batch size was increased from *1* per network node to *20* per network node (*29* to *580*), increasing by one for each test run. Contention was also varied from *0.5* to *1.5*, increasing by *0.2*. *50* trials with different random seeds were performed. The number of QoS levels and paths considered were both held constant at *3*.

As increasing batch size effectively increases the problem size, it was expected that runtime would rise exponentially with batch size. This would match the analytical runtime bound presented by Akbar [18]. Contention was not expected to affect the shape of this curve. However, as higher contention indicates each SLA is requesting more resources, for the same batch size higher contention would result in the heuristic using up network resources earlier. Essentially, increased blocking leads to reduced runtime.

7.3.2 Results

Refer to Figure 7.2. As expected, runtime increased exponentially. Note the similarly shaped but translated curves for different contention values. Higher contention resulted in lower runtime, as expected. It should be noted that standard deviation increased as batch size increased, but was very small for all tested values.

Figure 7.3 shows the ratio of the solution utility to the sum of the maximum utility

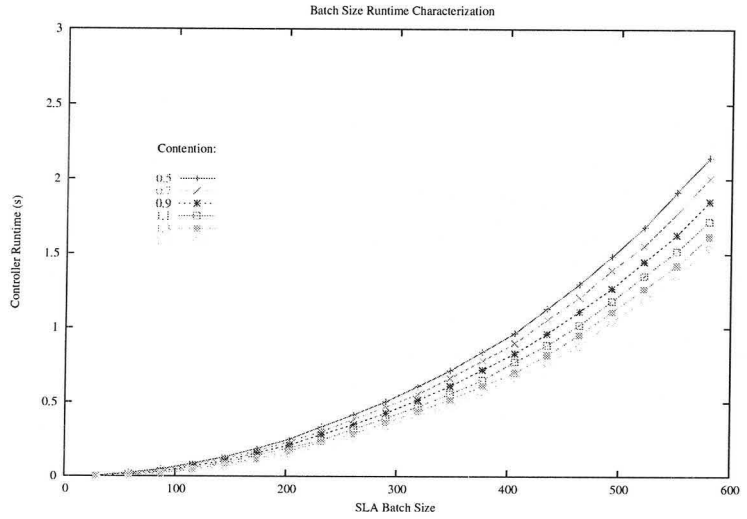

Figure 7.2. *Admission Controller batch size characterization - Runtime.*

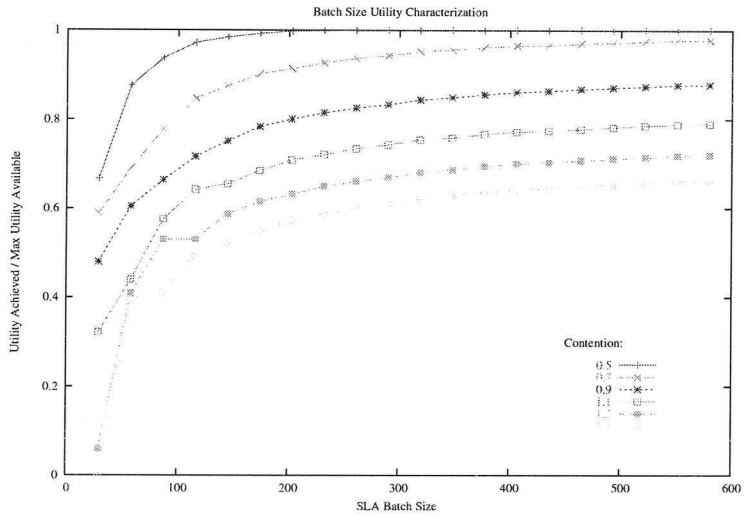

Figure 7.3. *Admission Controller batch size characterization - Achieved Utility.*

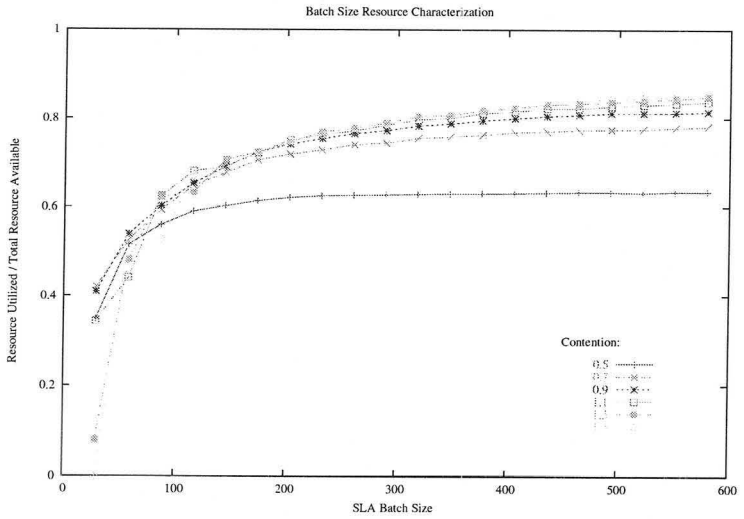

Figure 7.4. *Admission Controller batch size characterization - Resource Utilization.*

offered by each SLA. As batch size increases, this ratio remains relatively constant although it displays a slight logarithmic shape earlier. This initial behaviour is due to the lower granularity that fewer SLAs implies - the controller has less flexibility. The contention parameter dictates how high this ratio can get, as higher contention results in fewer SLAs admitted at higher QoS levels, and thus lower utility is achieved. With low contention, we see that almost all SLAs are admitted at their highest QoS level.

The resource utilization ratio described by Figure 7.4 indicates what fraction of the total network bandwidth was utilized. Topology and the end point distribution of the admission batch will generally determine how good this ratio can get. The input sets and network used here should permit a relatively high ratio as the topology has no critical paths and the SLA endpoints are uniformly distributed. Contention also determines how high the ratio goes. For lower values, the total resource demand is lighter and topological limits do not play a role. Higher contention values cause us to reach these limits more quickly. Increasing batch size improves the problem granularity and shows higher resource utilization as more SLAs are admitted, up to the steady state limit.

The admission rate (ratio of SLAs admitted to batch size) in Figure 7.5 increases with

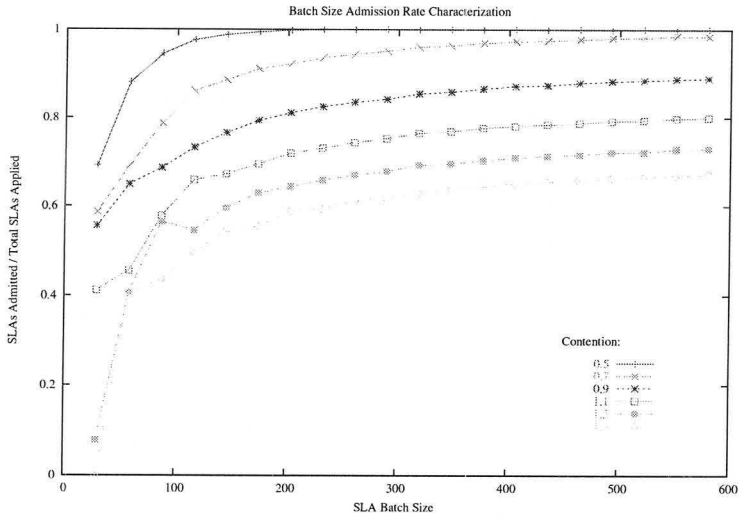

Figure 7.5. *Admission Controller batch size characterization - Admission Rate.*

batch size along a roughly logarithmic curve, approaching a steady state value. Increased batch size provides more options for the heuristic to choose from, improving the admission rate. Higher contention indicates each SLA asks for a greater percentage of the network resources, and thus the admission rate is lowered.

It should be noted that for the achieved utility, resource utilization and admission rate curves, the initial unstable and rapidly increasing portion of the result curve may be in part due to the method used to calculate the average bandwidth requested by each SLA. This value is calculated based on the contention, and for lower batch sizes may result in each SLA requesting the same or higher bandwidth than the maximum actually provided by the links. This will result in a greater sensitivity to the distribution of SLA endpoints, as admitting a single SLA may commit all of the resources of the path it utilizes. As batch size increases, each SLA's average bandwidth will become a lower fraction of the individual link bandwidths, allowing a given link or path to accommodate several SLAs

This experiment was repeated with the MHEU heuristic's local maxima escape technique disabled. The resulting data was almost identical to the above results, with only higher contention values and batch sizes showing a slight increase in runtime (on the order

of tens of milliseconds) for an equally marginal improvement in solution quality. This suggests that this technique was of neutral value for the sets of data tested. Whether it might prove useful in avoiding local maxima in actual application data is an open question.

7.4 Experiment - QoS Level Granularity Characterization

This test was designed to investigate the effect of the number of QoS levels defined by each SLA on admission controller performance.

7.4.1 Parameters

Once again, *50* trials were performed with different random seeds. The number of QoS levels was varied from *1* to *5* per SLA. Increasing the number of levels increases the complexity of the problem, but provides the controller with more choices to work with. Thus, it was expected that runtime would increase and the quality of solution would do likewise.

The primary experiment was run for the same range of batch sizes as the batch size characterization, but with a constant contention of *1.0*. This would give an indication of how these two parameters interact to affect runtime. This experiment was also conducted with a constant batch size of *10* per node and varied contention values, to illustrate the affect of increasing QoS level count in isolation.

Path count was kept constant at *3* paths per SLA. The same augmented ring network model was used.

7.4.2 Results

Results were generally as expected. Adding a QoS level to each SLA had a significant effect upon runtime. As *3* alternate paths were considered, each QoS level added *3* new choices to the MMKP. In Figure 7.6 the effect can be seen as this combined with increasing batchsize display a distinct exponential characteristic. The increased runtime is magnified

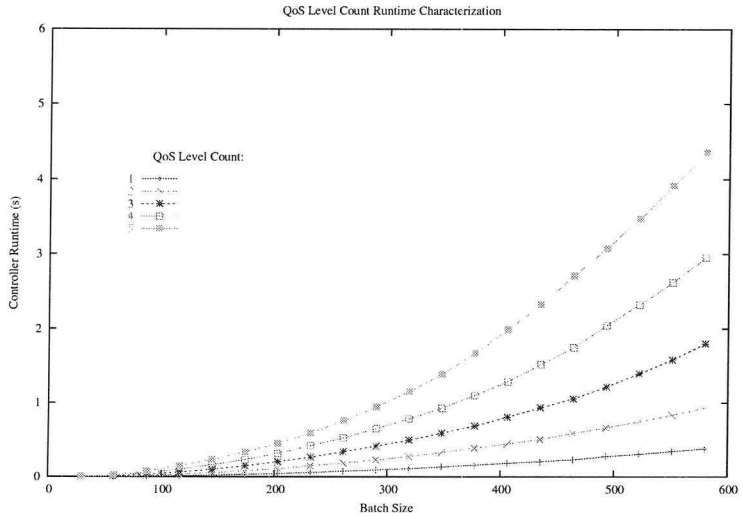

Figure 7.6. *QoS Level & Batchsize Characterization - Runtime.*

with higher batch sizes. Figure 7.7 shows the effect of increasing QoS level count independent of batch size.

Computationally, the effect of adding QoS levels, alternate paths, or SLAs can be translated into increasing the number of choices presented to the core MMKP.

Figure 7.8 was at first surprising in that it shows the ratio of achieved utility to offered utility decreasing as the number of QoS levels is increased. An improvement in solution quality had been expected. However, upon reviewing the structure of the experiment this result was validated. The input data sets were generated with a constant average utility. As more QoS levels are added, there are more QoS levels offering higher utility. This increases the maximum offered utility. However, the resource constraints do not relax, thus the utility of the admitted solutions remains relatively constant or improves at a rate less than the maximum, thus resulting in the decreasing ratio shown.

Resource utilization increased very slightly as further QoS levels were added. This was as expected, as more granularity was available to the heuristic and a closer fit could be found. Increasing contention brought resource utilization closer and closer to an asymptote of just below *80%*.

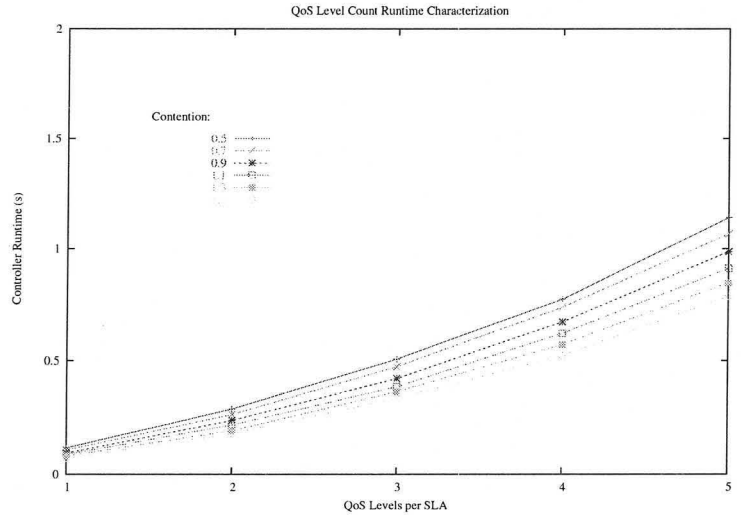

Figure 7.7. *QoS Level & Contention Characterization - Runtime.*

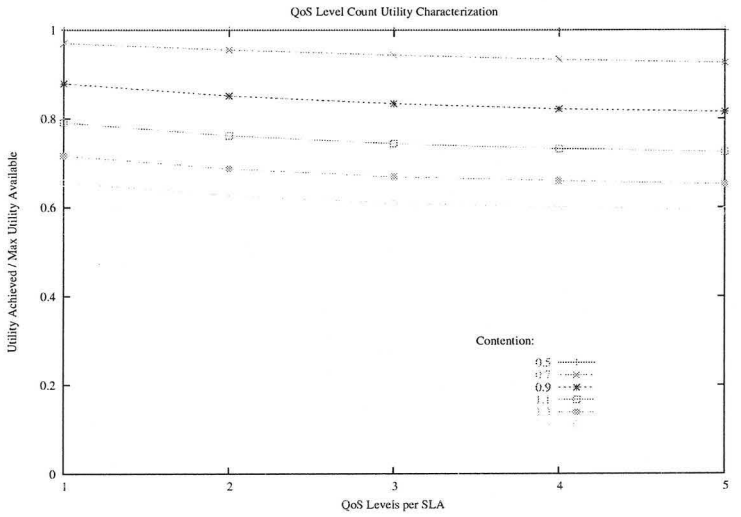

Figure 7.8. *QoS Level & Contention Characterization - Achieved Utility.*

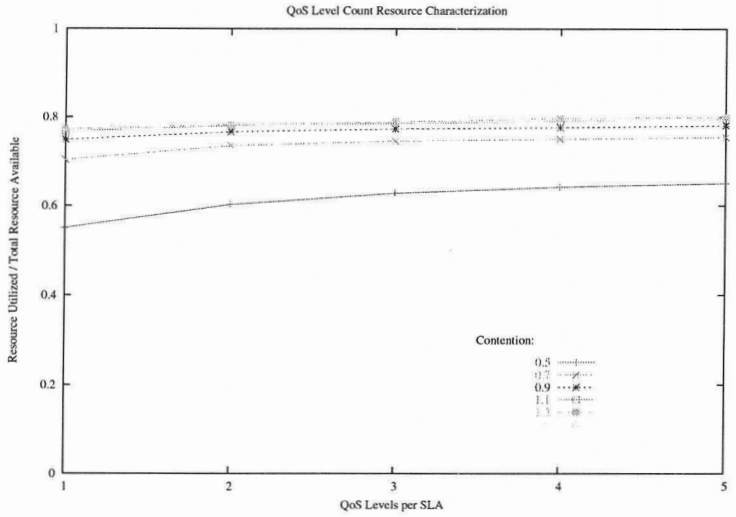

Figure 7.9. *QoS Level & Contention Characterization - Resource Utilization.*

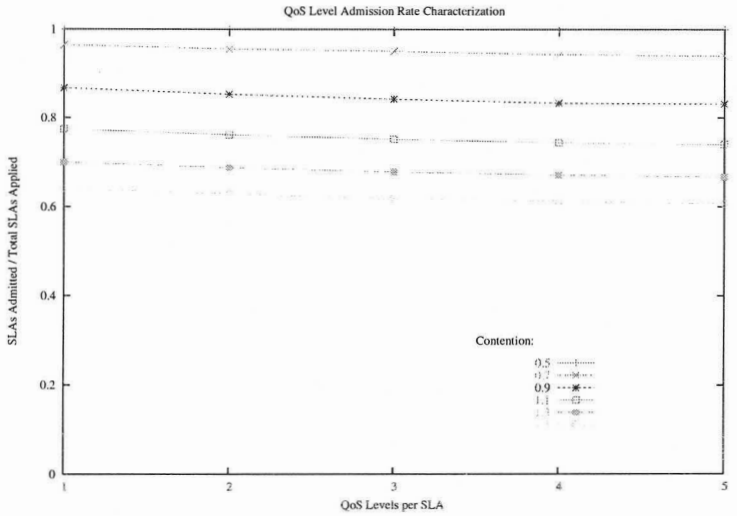

Figure 7.10. *QoS Level & Contention Characterization - Admission Rate.*

The admission ratio in Figure 7.10 was also initially puzzling. The ratio decreases slightly as more QoS levels are added. This indicates that more resources were being committed to fewer SLAs as the solution space was explored. The increasing number of QoS levels offering higher utility in exchange for more resource may explain this behaviour, as fewer SLAs are admitted at a higher QoS level. This effect was slight.

7.5 Experiment - Available Path Count Characterization

This experiment tested the effect of different numbers of available alternate paths on the admission controller. In an actual application, path selection would likely be governed by a policy internal to the network. As mentioned, this experiment used a shortest path finding algorithm to select the paths to be considered.

7.5.1 Parameters

This test is functionally equivalent to the previous one, in that its effect on the MMKP is also to increase the number of items in each group. The path count was varied from *1* to *5* for *50* trials with different random seeds. As in the previous experiment, one sequence was conducted varying batch size and another varying contention. These used the same parameters as the previous experiment. The same augmented ring network model was used.

7.5.2 Results

Results were very similar to the previous section, as expected. Figures 7.11 and 7.12 show the runtime curves for this experiment. These are very similar to the curve for QoS levels, although runtimes are slightly lower. As with the previous experiment, increasing the number of alternate paths increases the choices available to the heuristic, requiring more computation.

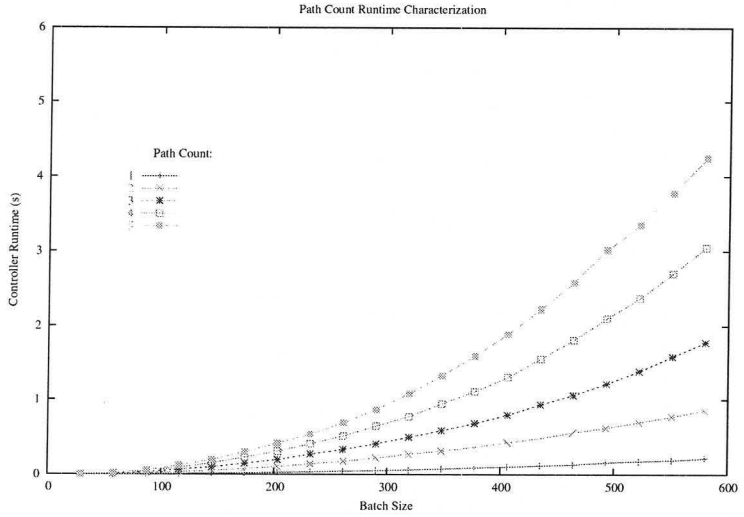

Figure 7.11. *Path Count & Batchsize Characterization - Runtime.*

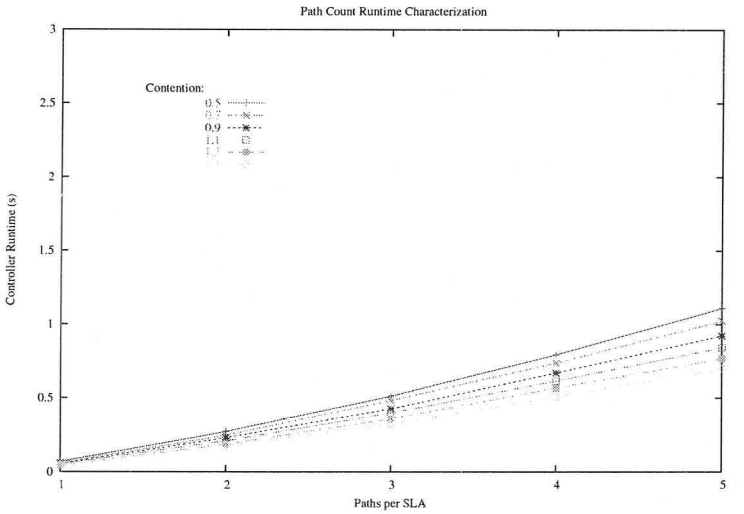

Figure 7.12. *Path Count & Contention Characterization - Runtime.*

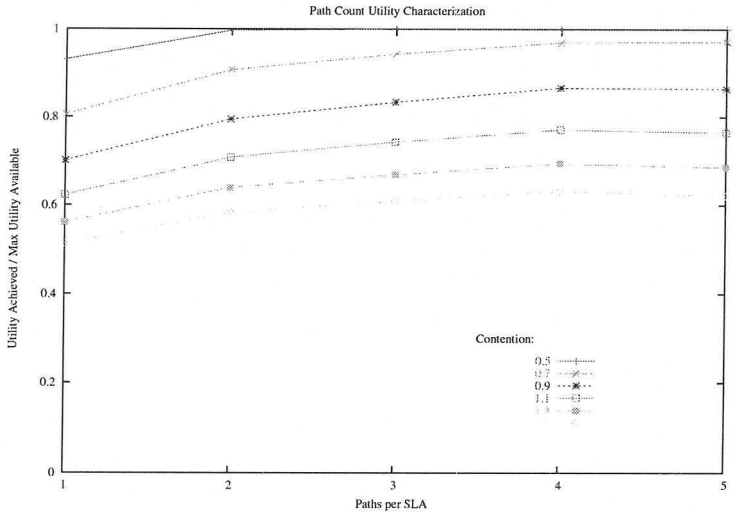

Figure 7.13. *Path Count & Contention Characterization - Utility Achieved.*

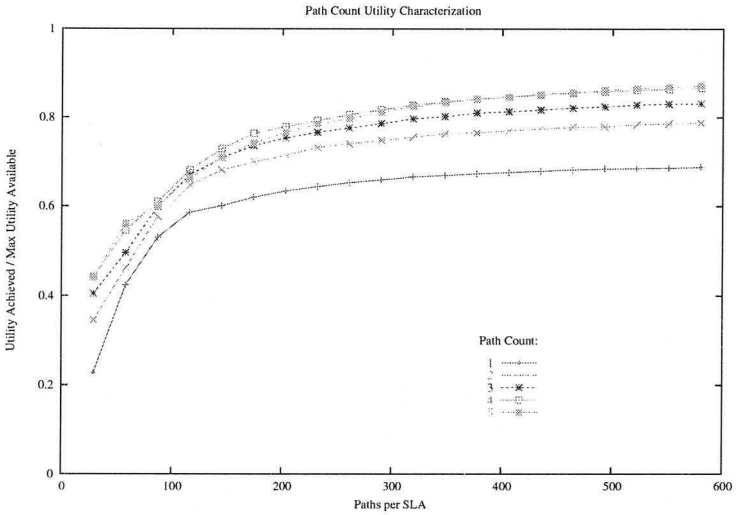

Figure 7.14. *Path Count & Batchsize Characterization - Utility Achieved.*

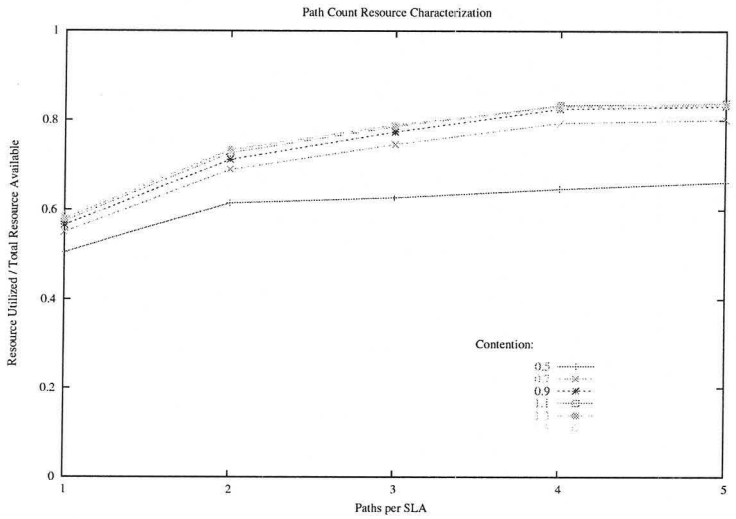

Figure 7.15. *Path Count & Contention Characterization - Resource Utilization.*

The utility achieved shown in Figure 7.13 exhibits the behaviour expected but not seen in Figure 7.8. That is, it increases approximately logarithmically as more alternate paths are made available. As more paths are available, the heuristic has more routing options for each SLA, allowing it to find a better fit for the resource constraints of the network. Figure 7.14 confirms this behaviour, and also shows there are diminishing returns for using additional paths. The additional utility achieved decreases for each additional path considered.

Note that for lower contention the maximum utility is more rapidly achieved, as all sessions can be admitted at their highest QoS level. Higher contention results in sessions being admitted at lower levels, translating the curve lower.

Resource utilization (Figure 7.15) exhibited behaviour very similar to that seen in the previous section (Figure **??**). This was a logarithmic curve approaching an asymptote of just over *80%* of resources consumed. This limit is determined by the network topology and distribution of end points of the SLAs. More alternate paths allowed the heuristic more routing options, in turn allowing more sessions to be admitted. This would result in more resources allocated.

The utility ratio curve was translated up as contention increased, because each SLA

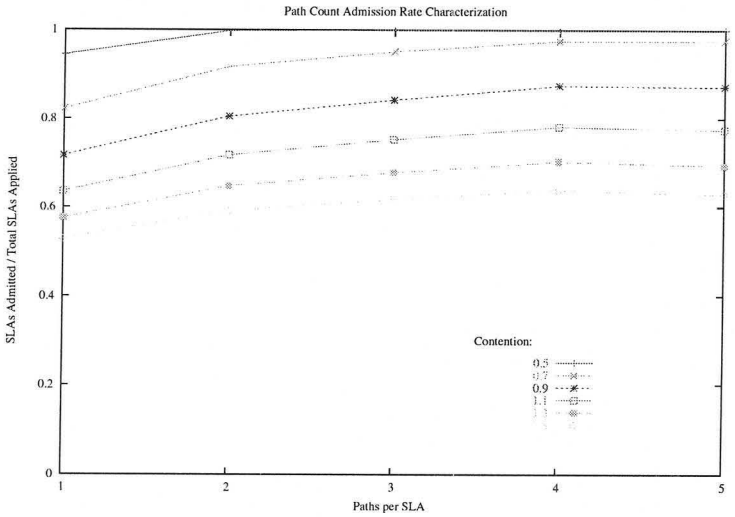

Figure 7.16. *Path Count & Contention Characterization - Admission Rate.*

required a greater percentage of system resources.

Unlike the results of the previous section (Figure 7.10), Figure 7.16 demonstrated the expected behaviour. The admission rate improved approximately logarithmically as the number of alternate paths was increased. This was due to there being more routing options allowing the controller to accommodate more SLAs. The admission ratio did appear to fall slightly from the asymptote at *5* paths per SLA. As more paths are considered, longer paths must be considered. If some of these are utilized, more resources will be used for some SLAs. Although fewer SLAs may be admitted due to this, a better solution in terms of achieved utility may be obtained. Note that the utility ratio continues to improve even when the admission ratio falls after *4* paths.

Higher contention translated the admission ratio curve lower, as fewer SLAs could be admitted when they each required more resources.

7.6 Experiment - Pre-Admission Characterization

This test sought to analyze the quality of the solution found when a partially pre-admitted solution was submitted.

7.6.1 Parameters

The primary parameter varied here was termed the pre-admission ratio. An initial batch of SLAs was submitted to the admission controller. A set of admitted SLAs resulted. The pre-admission ratio was defined as the fraction of these admitted SLAs that were kept and included in a second batch to be submitted again to the admission controller. The remaining SLAs in the second batch were newly generated without any initial state.

Thus, these pre-admission experiments sought to explore the effect of re-using part of an existing admission control solution. This approach would be suited to application environments, such as video on demand with no start time restrictions, where session start and end times vary, and some SLAs remain active when new entries are being considered. This iterative approach should reduce computation time, as less work remains to be done. The quality of the solution would be expected to decrease somewhat as the pre-admission ratio increased, as the heuristic would have less flexibility. This was the approach taken in Akbar's IHEU, which was essentially the same as MHEU with the exception of starting with a partial solution.

The same network was used as for the previous experiments in this chapter, with a constant batch size of *290*, *3* QoS levels per SLA and *3* alternate paths considered per SLA. Once again, *50* random trials were performed. Contention was varied from *0.5* to *1.5* as in previous experiments. The pre-admission ratio was varied from *0* to *0.9*. The input data sets were also submitted to the controller a second time with all SLAs reset to *not yet admitted.* Thus, a direct comparison could be made between the iterative approach and starting from scratch with the same data.

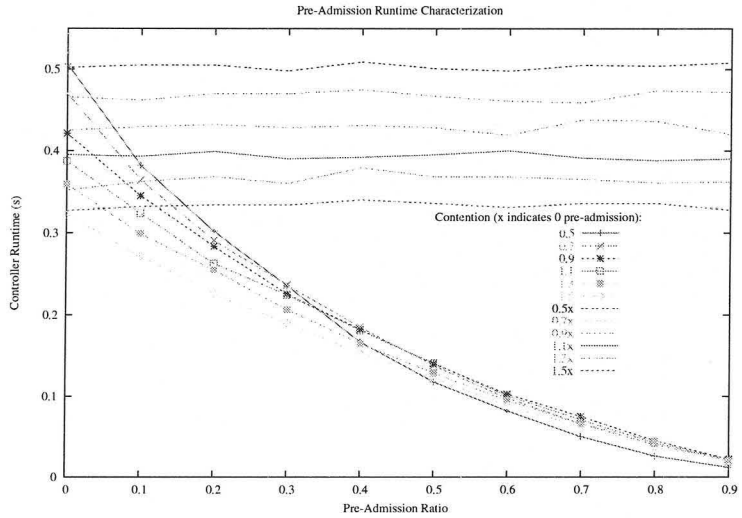

Figure 7.17. *Pre-Admitted Solution Characterization - Runtime.*

7.6.2 Results

As expected, Figure 7.17 shows runtime dropping drastically as the pre-admission ratio increases. Fewer resources remain to be allocated, and the heuristic thus has less work to do. Runtime remains relatively constant for the non-admitted data sets, with increasing contention increasing the runtime. This shows there are significant runtime performance benefits from using an iterative approach.

Figure 7.18 shows the achieved utility to total offered utility ratio. It can be seen that these values are fairly close for both the pre-admitted and non-admitted data sets, although the non-admitted sets show a slight improvement at higher pre-admission ratios. This is expected as the higher the pre-admission ratio, the fewer variables are left for the heuristic to manipulate in seeking a solution. However, the magnitude of this divergence is very small, suggesting that the runtime benefits of an incremental approach come at a low cost to the quality of the solution.

The resource utilization ratio (committed resource over total available resource) is shown in Figure 7.19. Once again, the pre-admitted and non-admitted data sets have similar values until they diverge at higher pre-admission ratio values. In this case, the non-admitted

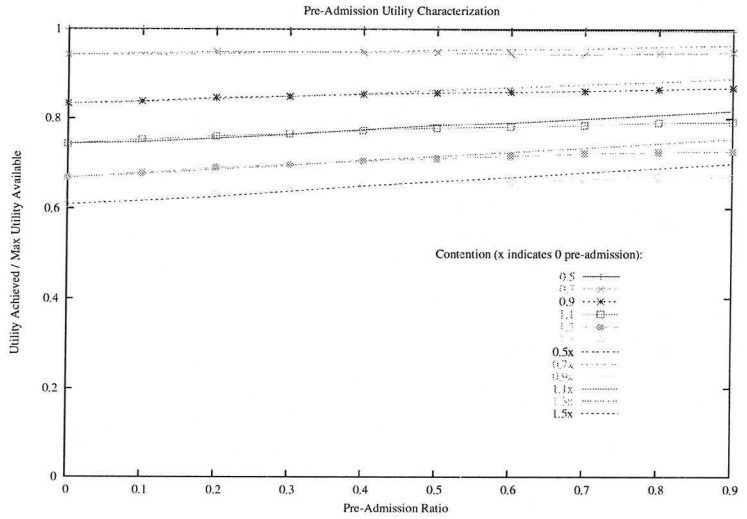

Figure 7.18. *Pre-Admitted Solution Characterization - Utility Achieved.*

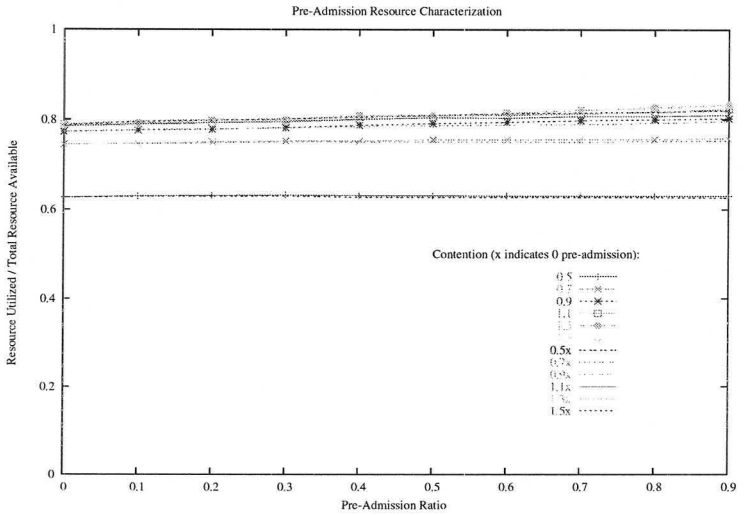

Figure 7.19. *Pre-Admitted Solution Characterization - Resource Utilization.*

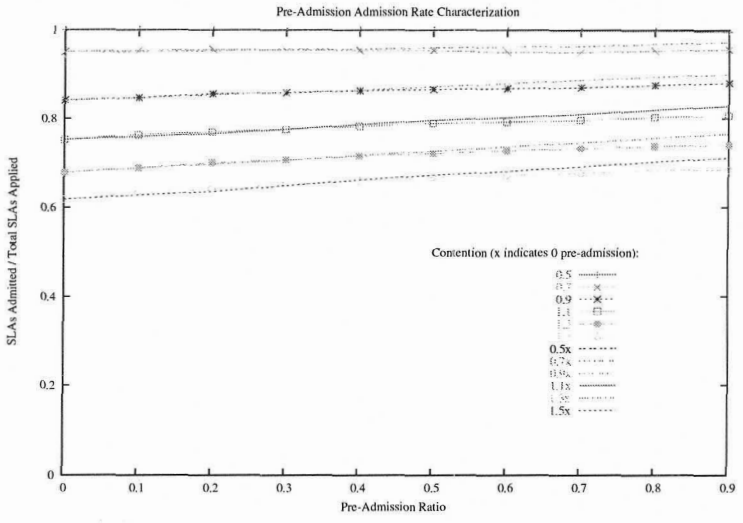

Figure 7.20. *Pre-Admitted Solution Characterization - Admission Rate.*

values are slightly lower. This indicates the heuristic finding a lower cost (thus better) solution as it has more flexibility in the non-admitted case. However, once again the magnitude of this divergence is very low.

The SLA admission rate shown in Figure 7.20 shows results very similar to the utility ratio in Figure 7.18. That is, the pre-admitted and non-admitted data are very close until the non-admitted results become slightly better for higher pre-admission ratios. The explanation for this is as before. The heuristic has more flexibility for the non-admitted case. The magnitude of this divergence illustrates once again that the solution quality sacrificed for the incremental approach is fairly insignificant.

Chapter 8

Experiments - Constrained Heuristic Evaluation

8.1 Experimental Purpose

The previous experiment was primarily a mathematical investigation of the behaviour of the MHEU heuristic. This was useful in characterizing its behaviour, but the parameter values used in the experiments bear little resemblance to values that would correspond to a realistic application.

In this series of experiments, we provide a model for performing an evaluation in a specific application context, where the values of the independent variables more accurately reflect real world values. The purpose of such an evaluation would be to evaluate the admission controllers suitability for a particular application environment, comprising an actual network.

8.2 Experimental Design

The independent variables of the Model Implementation must be set according to the application environment being modeled. We will model a *Video on Demand* (VoD) type of application. Parameter specifications are explained below.

Table 8.1. *SLA Configuration for Application Simulation.*

Parameter	Configuration Description
Batch Size	Varied within a range.
Number of QoS Levels	1, 2 or 3.
Source - Destination	Static Source, Destination Randomly selected with a Uniform Distribution.
Bandwidth	4, 10 or 25 Mbps to simulate MPEG Video Streams.
Delay	Ignored as one-way video stream can be buffered.
Utility	Uniformly Distributed in ranges 4-7, 7-10, 10-15. Units considered as cost per movie ($/movie).

8.2.1 User Request Parameters

Table 8.1 shows how the input SLA batches were configured for the experiment.

The values chosen for bandwidth are based upon estimated requirements for MPEG streams of various quality [1]. We assume an application environment where requests are submitted ahead of time and gathered into a single batch. The VoD sessions all start and end at approximately the same time for a given batch. Thus, using the IHEU iterative approach is inappropriate as we don't have a partial solution at the start of the sessions. Customers submit SLAs specifying the amount they are willing to pay to watch a movie at one of the QoS levels during a given time block.

Customers can specify up to *3* QoS Levels, although to simplify the simulation all SLAs within a batch specify the same number.

8.2.2 Network Parameters

We use a network topology based upon MichNet (Figure 8.1 [17]), an actual backbone network installation in the state of Michigan. Bandwidths are based upon those found on a

map of this topology. Delays are ignored, as a one-way VoD session can be buffered at the client or at a proxy.

We modify the topology such that all links to external networks connect to a single node, and we consider this node as the source of our video data. Thus, we can view this application as distributing video streams from an external provider to the citizens of Michigan over MichNet.

8.2.3 Controller Parameters

The number of paths considered by the controller was again the only controller parameter.

8.2.4 Experimental Framework

As in the previous chapter, a support framework was implemented to perform these experiments. This once again included code to generate input batches and a series of scripts to automate the input generation, test running and result collating.

8.3 Experiment - Application Simulation

8.3.1 Parameters

The parameters were specified as discussed above. Batch Size was varied from *100* to *1500*, at *7* different data points. *50* trials with different random seeds were performed. The number of QoS levels and Paths considered were each varied from *1* to *3*. QoS levels are added starting with the lowest (i.e. QoS Level 1, then QoS Levels 1 and 2, etc).

From previous experiments, we can conjecture that runtime will increase exponentially with batchsize and SLA complexity (number of QoS levels and paths considered).

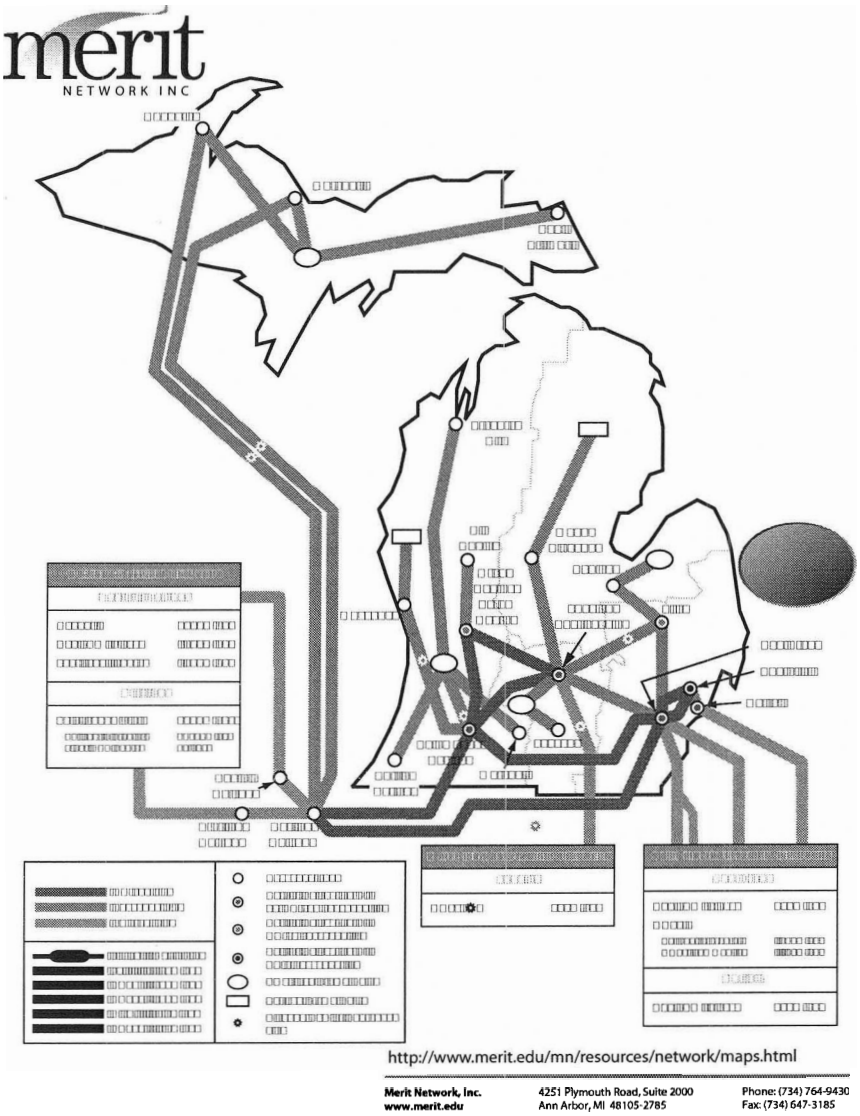

Figure 8.1. *A Topology Map of MichNet*

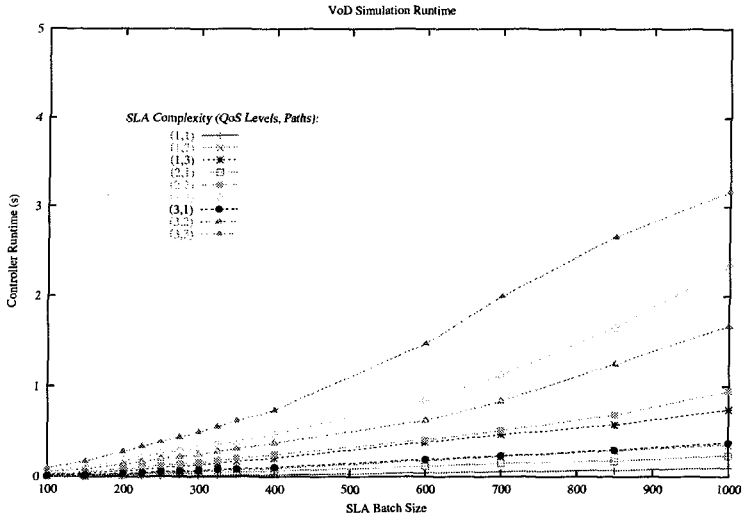

Figure 8.2. *Runtime Performance of Simulation Experiments.*

8.3.2 Results

The runtime results are shown in Figure 8.2 below.

As expected, the runtime exhibited an exponential response to increasing problem size. As batchsize increased, the number of QoS levels and paths considered had an increasing effect upon computation time.

Figure 8.3 shows the ratio of utility achieved to total utility offered. All the trials showed a decreasing value over time. This is as expected, as the total maximum utility offered was not held constant in these trials. The more SLAs in the batch, the greater the maximum available utility. As the average resource requirement per SLA is also constant, the resource limitations of the network will result in a limit on the number of SLAs that can be admitted. This will limit the total attainable utility. Once this limit is reached, further increases to batch size will cause this value to become a smaller and smaller fraction of the total offered.

Figure 8.3 also shows greater numbers of QoS levels with lower utility ratios. This also makes sense as the higher QoS levels increase the total utility offered, while any corresponding gain in achieved utility is not as high. We can also infer that considering more paths improves the utility attained. This is directly visible on the graph as considering more

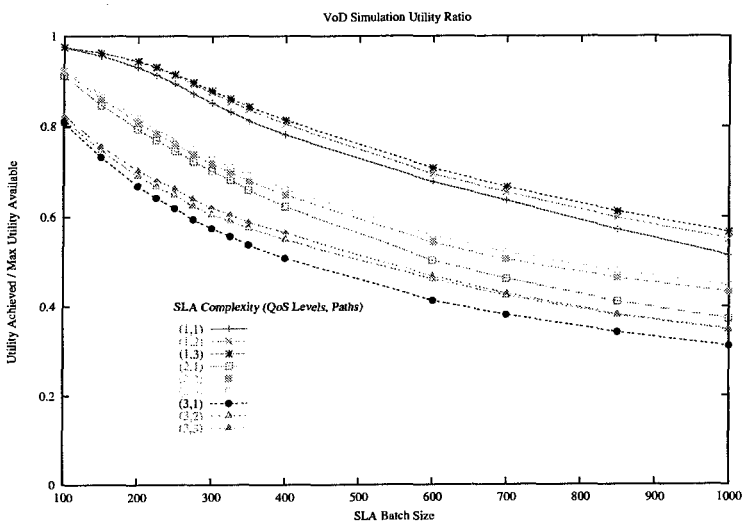

Figure 8.3. *Simulation - Attained Utility.*

paths does not affect the maximum utility offered by a batch.

The network resource utilization shown in Figure 8.4 indicates that increasing batch size and problem complexity will increase the amount of network resources used, up to a limit. This behaviour is especially visible with the higher QoS level and path count experiments.

The limits of resource utilization is determined by the topology of the network and the relative amount of resource SLAs are requesting. As higher QoS levels are added, SLAs ask for a larger percentage of link bandwidth. This will result in resources being fully committed sooner, especially for choke points in the network.

Figure 8.5 is perhaps the most surprising of the results for this sequence of experiments.

This shows very little difference in the admission rate when SLA complexity is increased. One would expect a decrease in admission rate as the number of QoS levels is increased, and an improvement when more paths are considered. These trends are present, but they are very slight.

The downward trend of the entire graph is again due to a relatively constant number of SLAs being admitted while batch size continues to increase. This is due to SLAs having a constant average bandwidth requirement, as opposed to previous experiments where this

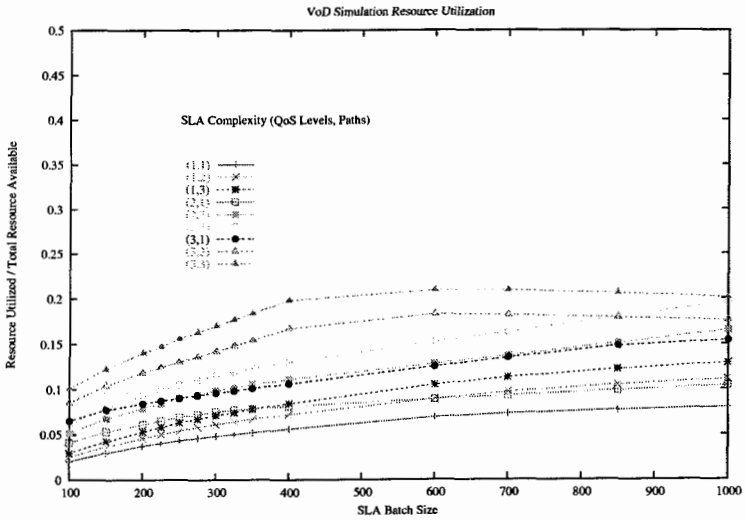

Figure 8.4. *Simulation - Resource Utilization.*

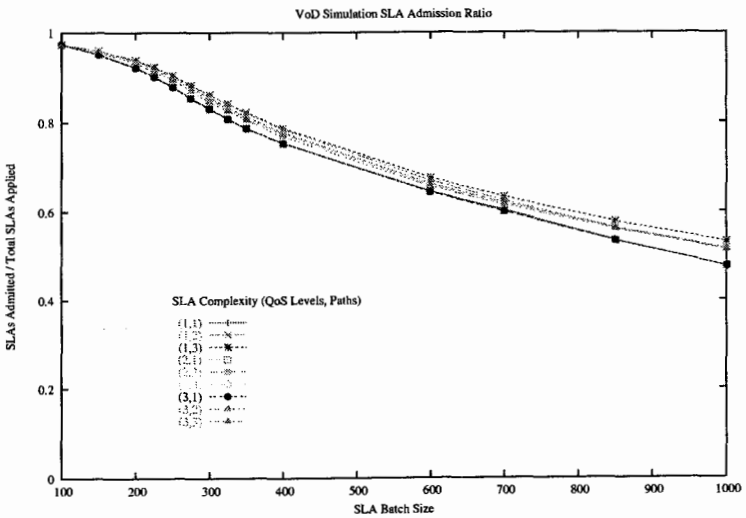

Figure 8.5. *Simulation - SLA Admission Rate.*

value was decreased as batch size increased, to keep contention constant.

Chapter 9

Conclusions

Here we discuss the conclusions that can be drawn from this work, along with the major contributions made. Our conclusions can be divided into two broad categories: Those specific to the evaluation of the Model Implementation of the Utility Model based Network Admission Controller, and general conclusions regarding Network Admission Control

9.1 Conclusions - Model Implementation of a Utility Model based Network Admission Controller

Is the MHEU heuristic suitable for performing Network Admission Control?

It is difficult to give a simple answer to this question. The experiments performed characterize the performance of the heuristic in responce to a wide range of variables. The quality of the solution generated was fairly consistent across all the experiments, while the runtime varied significantly. In particular, it was heavily affected by the number of subscribers, the number of SLAs and the number of paths considered per SLA.

These are all factors that will depend upon the application for which users are requesting service, and upon the deployment environment.

The question of what is an acceptable runtime for a controller is not easily answered either, as this depends upon the way SLA requests are gathered and how quickly a response must be returned. This is, again, application dependant.

Thus, we cannot explicitly say whether the MHEU heuristic is generically suitable

for admission control. This is acceptable, however, as we have identified the parameters required to determine whether it is suitable for a specific application. The detailed characterization presented in our results can be used to evaluate the heuristic's suitability for a given application by fixing the appropriate variables and evaluation criteria accordingly.

The constrained evaluation presented in Chapter 8 is an example of just this, using parameters from MPEG streams and a real world network for the parameters of the problem. From this the expected performance of the controller can be determined.

Two issues specific to MHEU were investigated.

The first was the value of the Iterative version of the MHEU heuristic. This was investigated in the experiments which submitted partially pre-admitted batches. This demonstrated that the iterative approach could provide significant runtime reductions at minimal expense to solution quality. However, this approach required that the application environment be suitable. SLA requests would have to arrive and apply in such a way that saving a previous solution made sense.

For example, in a Video Conferencing application with sessions lasting an arbitrary length of time and admission control performed at regular intervals, this approach makes sense. At each admission control interval, session which have ended are removed from the solution, existing sessions are kept at their current level and newly applying sessions are added.

In a Video-on-Demand application where sessions all start and end at fixed times and admission control is performed once at the start of each session period (i.e. movie showtimes), an iterative approach would provide no benefit.

This result would likely apply to other heuristics. An iterative approach can greatly reduce problem size where the application is suitable.

The second MHEU specific issue investigated the value of a specific Local Maxima Escape technique presented in the actual heuristic by Akbar. It can be concluded from the experimental observations that this step neither incurred a significant runtime penalty nor provided improved solution quality.

9.2 Conclusions - Evaluating a Network Admission Controller

The approach used in performing this evaluation of MHEU can be easily adapted to evaluate the performance of other heuristics. Doing so could be as simple as substituting a different Utility Model heuristic into the presented Model Implementation and repeating the same sequence of experiments. The Model Implementation was designed with this concept in mind, and is sufficiently modular.

It would also be possible to utilize a different network and subscriber interface model. There are many different approaches to representing and specifying user QoS parameters. In this case, the Model Implementation might not be as useful, but the experimental approach shown is useful. Namely, we identify the precise User Request parameters, Network parameters and Controller parameters of the implementation being evaluated, and design experiments manipulating these as the independent variables while observing the solution quality and runtime performance.

One of the major insights provided by this work is that the parameters that define an admission control problem will vary greatly among applications. The requirements of user requests and the network topology all depend upon what it is the specific application is designed to do. As most conceivable Network Admission Controllers will also depend upon these parameters to define the problem they solve, any meaningful evaluation of their performance will need to be placed in the context of a specific application.

9.3 Contributions

The major contributions of this work can be summarized as follows:

- Development of an efficient and reusable Model Implementation of an Utility Model based Network Admission Controller.

- Development of systematic methodology for evaluating the performance of a Network Admission Controller.
- Characterization of the performance of a Model Implementation of Utility Model based Network Admission Controller using this approach.
- Insight into the performance of the MHEU heuristic.

Chapter 10

Future Work

This chapter introduces some topics related to the Utility Model based Network Admission Controller which deserve further research.

10.1 Application Models

In this work, one application based simulation of the Model Implementation was presented. This characterized the performance of the controller with parameters based on a Video-on-Demand (VoD) application on an existing backbone network.

It would be useful to develop other application and network models, and perform similar simulations. This would serve to both refine the modelling process and provide more specific evaluations of our Model Implementations performance.

Suggested applications could include Video Teaching or Conferencing, Interactive VoD, High Performance Multiplayer Games, Virtual Reality Environments and others. These applications could be simulated on network models based on a variety of deployed or theoretical network topologies.

Different SLA batching schemes could also be tested, as the suitability of these depend upon the application model under consideration.

Performing such evaluations would help to determine the maximum problem size that could be solved in a reasonable amount of time, thus determining limits such as the maximum supportable subscriber base.

10.2 Multicast Resource Allocation

A potential criticism of the Model Implementation presented in this work is its use of a single source and destination specification in the SLAs. Depending upon the application, it may be far more efficient to provide a multiple-destination or even multiple-source model.

This ties in closely with multicast, which has been an area of substantial research. Modification of the existing model to include such capabilities would involve incorporating research on efficient multicast tree construction, which can be modeled as a Steiner Tree Problem [7][24]). While this problem is NP-complete, a number of heuristics exist for generating approximate solutions.

In the current model, mapping of QoS specifications into resources is computationally insignificant when compared to the solving of the MMKP itself. The use of a multicast transmission model might affect this, and would have to be investigated.

10.3 Alternate Utility Model and Network Admission Control Heuristics

The Model Implementation and experimental framework presented could be used to evaluate alternative heuristic's for solving the Utility Model MMKP. Some of these heuristics were presented for comparison in Akbar's work, but a more detailed evaluation could be beneficial. Of specific interest were Moser's Heuristic, a Convex Hull approach and a Greedy Heuristic.

Exploring alternate approaches could also be extended to the formulation of the admission control problem itself. The Utility Model utilized the MMKP, but other types of problems and their associated heuristics could be used.

10.4 SLAOpt and Existing QoS Protocols

Further work should be done on establishing the relationship between a Utility Model based Network Admission Controller and existing QoS Protocols such as RSVP, DiffServ or IntServ. Also, protocols such as MPLS could prove important to admission control as explicit routing seems to be a requirement of many models.

The development of a more specific model for the interoperation of a controller such as our Model Implementation would help determine where our model could work with these protocols, and where they may conflict. This is important for the evolution of the Utility Model as a practically useful admission control approach.

Bibliography

[1] A. Mehaoua and R.Boutaba, "The Impacts of Errors and Delays on the Performance of MPEG2 Video Communications," in *In Proceedings of IEEE International Conference On Acoustics, Speech, and Signal Processing (ICASSP'99).* Phoenix AZ.: IEEE, March 1999, pp. 2195–2198.

[2] D. Awduche et al., "Requirements for Traffic Engineering Over MPLS," IETF, Network Working Group, RFC 2702, September 1999.

[3] Daniel Veillard et al. (2004, June) The XML C parser and toolkit of Gnome. [Online]. Available: http://www.xmlsoft.org/index.html

[4] E. Rosen et al., "Multiprotocol Label Switching Architecture," IETF, Network Working Group, RFC 3031, January 2001.

[5] Ernesto Q. Vieira Martins, Marta Margarida B. Pascoal and Jos Luis E. Santos, "A New Algorithm for Ranking Loopless Paths," CISUC, Tech. Rep., May 1997.

[6] Fayez El Gebali, *Computer Communication Networks, Analysis and Design*, First ed. NorthStar Digital Design, Inc., 2001.

[7] F.K. Hwang & D.S. Richards, "Steiner Tree Problems," *Networks*, vol. 22, pp. 55–89, 1992.

[8] G. Apostopoulos et al., "QoS Routing Mechanisms and OSPF Extensions," IETF, Network Working Group, RFC 2676, August 1999.

[9] Grenville Armitage, *Quality of Service in IP Networks, Foundations for a Multi-Service Network*, First ed. MacMillan Technical Publishing, 2000.

[10] Herbert Schildt, *C The Complete Reference*, Third ed. Osborne McGraw-Hill, Inc., 1995.

[11] J. Moy, "OSPF Version 2," IETF, Network Working Group, RFC 2328, July 1997.

[12] J. Moy, *OSPF Anatomy of a Routing Protocol.* Addison & Wesley, 1998.

[13] J. Postel, "Internet Protocol - DARPA Internet Program Protocol Sepcification," IETF, Network Working Group, RFC 791, September 1981.

[14] Jay Fenlason & Richard Stallman. (2004, June) GNU gprof Documentation. [Online]. Available: http://sources.redhat.com/binutils/docs-2.15/gprof/index.html

[15] Jay L. Devore, *Probability and Statistics for Engineering and the Sciences*, Fourth ed. International Thomson Publishing, Inc., 1995.

[16] Jian Pu et al., "SLA Admission Controller for Reliable MPLS Networks," in *Proceedings of IASTED International Conference on Applied Informatics (AI 2003)*. Innsbruck, Austria: IASTED, February 2003.

[17] Merit Networks Inc. (2004, June) MicnNet Backbone with Planned Upgrades. [Online]. Available: http://www.merit.edu/mn/resources/network/maps.html

[18] Mostafa Akbar, "The Distributed Utility Model Applied to Optimal Admission Control & QoS Adaption in Multimedia Systems and Enterprise Networks." Ph.D. dissertation, University of Victoria, 2002.

[19] Mostofa Akbar et al., "Admission Control and QoS Adaptation in Distributed Multimedia Server System," in *ITCom*, Denver, USA, August 2001.

[20] Mostofa Akbar et al., "Heuristic Solutions for the Multiple-Choice Multi-Dimension Knapsack Problem," in *International Conference on Computational Science*, San Francisco, USA, May 2001.

[21] Mostofa Akbar et al., "Application of UM-D to Optimal Server Selection for Content Routing," in *Proceedings of ICECE 02*. Dhaka, Bangladesh: ICECE, December 2002.

[22] Mostofa Akbar et al., "Distributed Utility Model for Distributed Multimedia Server System," in *Proceedings of ICCIT 02*. Dhaka, Bangladesh: ICCIT, December 2002.

[23] Mostofa Akbar et al., "Optimal Admission Controllers for Service Level Agreements in Enterprise Networks," in *6th SCI Conference*, Orlando, FL, USA, July 2002.

[24] P. Winter, "Steiner Problem in Networks: A Survey," *Networks*, vol. 17, pp. 129–167, 1987.

[25] P.J. Plauger & Jim Brodie. (2004, June) Standard C. [Online]. Available: http://www-ccs.ucsd.edu/c/

[26] R. Braden et al., "Integrated Services in the Internet Architecture: An Overview," IETF, Network Working Group, RFC 1633, June 1994.

[27] R. Braden et al., "Resrouce ReSerVation Protocol (RSVP) – Version 1 Functional Specification," IETF, Network Working Group, RFC 2205, September 1997.

[28] Randal L. Schwarz and Tom Christiansen, *Learning Perl*, Second ed. O'Reilly & Associates INC., 1997.

[29] Robert K. Watson, "Applying the Utility Model to IP Networks." Master's thesis, University of Victoria, Department of Electrical and Computer Engineering, 2001.

[30] S. Blake et al., "An Architecture for Differentiated Services," IETF, Network Working Group, RFC 2475, December 1998.

[31] Shahadat Khan, "Quality Adaption in a Multisession Multimedia System: Model, Algorithms and Architecture." Ph.D. dissertation, University of Victoria, Department of Computer Science, 1998.

[32] Shahadat Khan et al., "Solving the Knapsack Problem for Adaptive Multimedia Systems," *Studia Informatica*, 2002.

[33] Shahadat Khan et al., "Optimal Quality of Service Routing and Admission Control Using the Utility Model," *Future Generation Computer Systems (FGCS)*, vol. 19, pp. 1063–1073, October 2003.

[34] Steven J.R. Shelford, "Implementation, Evaluation and Application of Distributed Heuristics for Optimizing SLA Admission," Master's thesis, University of Victoria, Department of Computer Science, 2003.

[35] Steven Shelford et al., "Distributed Optimal Admission Controllers for Service Level Agreements in Interconnected Networks," in *Proceedings of IASTED International Conference on Applied Informatics (AI 2003)*. Innsbruck, Austria: IASTED, February 2003, pp. 565–570.

[36] The GCC Team. (2004, May) GCC Online Documentation. [Online]. Available: http://gcc.gnu.org/onlinedocs/

Appendix A

Experiment Equipment Specification

The following table specifies the equipment used to run all the Model Implementation experiments.

Table A.1. *Experimental Equipment Details.*

Parameter	Description
Processor	AMD Athlon 1.3 GHz 256kB Cache
Primary Memory (RAM)	512 MB
Operating System	Linux (Fedora Core 1) Kernel 2.4.22
Compiler	GNU C Compiler (gcc) version 2.3.2 (no compiler optimizations)
Profiler	GNU Profiler (gprof) 2.14.9

Appendix B

SLA Admission State Document Specification

Below is the XML Document Type Definition (DTD) for the Admission State Document used as the primary input for the Model Implementation of a Utility Model Based Admission Controller presented in this work.

B.1 admission_state_doc.dtd

```
<?xml version="1.0"?>
<!ELEMENT admission_state (sla+, network+)>
<!ELEMENT sla (qos+,path?)>
   <!ATTLIST sla id CDATA #REQUIRED>
   <!ATTLIST sla source CDATA "0">
   <!ATTLIST sla destination CDATA "0">
   <!ATTLIST sla active_profile CDATA "-1">
<!ELEMENT qos (EMPTY)>
   <!ATTLIST qos capacity CDATA "0">
   <!ATTLIST qos delay CDATA "0">
   <!ATTLIST qos utility CDATA "0">
<!ELEMENT path (node+, link*)>
<!ELEMENT network (node+, link*)>
<!ELEMENT node (EMPTY)>
   <!ATTLIST node name CDATA #IMPLIED>
   <!ATTLIST node id CDATA #REQUIRED>
<!ELEMENT link (EMPTY)>
   <!ATTLIST link id CDATA #REQUIRED>
   <!ATTLIST link source CDATA "0">
   <!ATTLIST link destination CDATA "0">
   <!ATTLIST link capacity CDATA "0">
   <!ATTLIST link delay CDATA "0">
```

Appendix C

Network Models

This Appendix contains XML specification of the two networks used in the evaluation experiments presented.

Theses are a 29 node augmented ring network, and a topology based on the MichNet backbone.

C.1 augmented_ring.xml

```
<?xml version = '1.0'?>
<admission_state>
<network bidirectional="TRUE">
   <node name="Node_1" id = "0"></node>
   <node name="Node_2" id = "1"></node>
   <node name="Node_3" id = "2"></node>
   <node name="Node_4" id = "3"></node>
   <node name="Node_5" id = "4"></node>
   <node name="Node_6" id = "5"></node>
   <node name="Node_7" id = "6"></node>
   <node name="Node_8" id = "7"></node>
   <node name="Node_9" id = "8"></node>
   <node name="Node_10" id = "9"></node>
   <node name="Node_11" id = "10"></node>
   <node name="Node_12" id = "11"></node>
   <node name="Node_13" id = "12"></node>
   <node name="Node_14" id = "13"></node>
   <node name="Node_15" id = "14"></node>
   <node name="Node_16" id = "15"></node>
   <node name="Node_17" id = "16"></node>
   <node name="Node_18" id = "17"></node>
   <node name="Node_19" id = "18"></node>
   <node name="Node_20" id = "19"></node>
   <node name="Node_21" id = "20"></node>
   <node name="Node_22" id = "21"></node>
   <node name="Node_23" id = "22"></node>
   <node name="Node_24" id = "23"></node>
```

```
<node name="Node_25" id = "24"></node>
<node name="Node_26" id = "25"></node>
<node name="Node_27" id = "26"></node>
<node name="Node_28" id = "27"></node>
<node name="Node_29" id = "28"></node>
<link id="0" source="0" destination="1"  capacity="100000" delay="0.1"></link>
<link id="1" source="1"  destination="2"  capacity="100000" delay="0.1"></link>
<link id="2" source="2"  destination="3"  capacity="100000" delay="0.1"></link>
<link id="3" source="3"  destination="4"  capacity="100000" delay="0.1"></link>
<link id="4" source="4"  destination="5"  capacity="100000" delay="0.1"></link>
<link id="5" source="5" destination="6"  capacity="100000" delay="0.1"></link>
<link id="6" source="6"  destination="7"  capacity="100000" delay="0.1"></link>
<link id="7" source="7"  destination="8" capacity="100000" delay="0.1"></link>
<link id="8" source="8"  destination="9"  capacity="100000" delay="0.1"></link>
<link id="9" source="9"  destination="10" capacity="100000" delay="0.1"></link>
<link id="10" source="10" destination="11" capacity="100000" delay="0.1"></link>
<link id="11" source="11"  destination="12" capacity="100000" delay="0.1"></link>
<link id="12" source="12"  destination="13" capacity="100000" delay="0.1"></link>
<link id="13" source="13"  destination="14" capacity="100000" delay="0.1"></link>
<link id="14" source="14"  destination="15" capacity="100000" delay="0.1"></link>
<link id="15" source="15" destination="16" capacity="100000" delay="0.1"></link>
<link id="16" source="16"  destination="17" capacity="100000" delay="0.1"></link>
<link id="17" source="17"  destination="18" capacity="100000" delay="0.1"></link>
<link id="18" source="18"  destination="19" capacity="100000" delay="0.1"></link>
<link id="19" source="19"  destination="20" capacity="100000" delay="0.1"></link>
<link id="20" source="20" destination="21" capacity="100000" delay="0.1"></link>
<link id="21" source="21"  destination="22" capacity="100000" delay="0.1"></link>
<link id="22" source="22"  destination="23" capacity="100000" delay="0.1"></link>
<link id="23" source="23"  destination="24" capacity="100000" delay="0.1"></link>
<link id="24" source="24"  destination="25" capacity="100000" delay="0.1"></link>
<link id="25" source="25" destination="26" capacity="100000" delay="0.1"></link>
<link id="26" source="26"  destination="27" capacity="100000" delay="0.1"></link>
<link id="27" source="27"  destination="28" capacity="100000" delay="0.1"></link>
<link id="28" source="28"  destination="0" capacity="100000" delay="0.1"></link>
<link id="29" source="0" destination="3" capacity="100000" delay="0.1"></link>
<link id="30" source="3"  destination="6" capacity="100000" delay="0.1"></link>
<link id="31" source="6"  destination="9"  capacity="100000" delay="0.1"></link>
<link id="32" source="9"  destination="12" capacity="100000" delay="0.1"></link>
<link id="33" source="12"  destination="15" capacity="100000" delay="0.1"></link>
<link id="34" source="15" destination="18" capacity="100000" delay="0.1"></link>
<link id="35" source="18"  destination="21" capacity="100000" delay="0.1"></link>
<link id="36" source="21"  destination="24" capacity="100000" delay="0.1"></link>
<link id="37" source="24"  destination="27" capacity="100000" delay="0.1"></link>
<link id="38" source="27"  destination="1" capacity="100000" delay="0.1"></link>
<link id="39" source="1" destination="4" capacity="100000" delay="0.1"></link>
<link id="40" source="4"  destination="7" capacity="100000" delay="0.1"></link>
<link id="41" source="7"  destination="10"  capacity="100000" delay="0.1"></link>
<link id="42" source="10"  destination="13" capacity="100000" delay="0.1"></link>
<link id="43" source="13"  destination="16" capacity="100000" delay="0.1"></link>
<link id="44" source="16" destination="19" capacity="100000" delay="0.1"></link>
<link id="45" source="19"  destination="22" capacity="100000" delay="0.1"></link>
<link id="46" source="22"  destination="25" capacity="100000" delay="0.1"></link>
<link id="47" source="25"  destination="28" capacity="100000" delay="0.1"></link>
<link id="48" source="28"  destination="2" capacity="100000" delay="0.1"></link>
<link id="49" source="2" destination="5" capacity="100000" delay="0.1"></link>
<link id="50" source="5"  destination="8" capacity="100000" delay="0.1"></link>
<link id="51" source="8"  destination="11"  capacity="100000" delay="0.1"></link>
```

```
<link id="52" source="11"  destination="14" capacity="100000" delay="0.1"></link>
<link id="53" source="14"  destination="17" capacity="100000" delay="0.1"></link>
<link id="54" source="17" destination="20" capacity="100000" delay="0.1"></link>
<link id="55" source="20"  destination="23" capacity="100000" delay="0.1"></link>
<link id="56" source="23"  destination="26" capacity="100000" delay="0.1"></link>
<link id="57" source="26"  destination="0" capacity="100000" delay="0.1"></link>

<link id="58" destination="0" source="1"  capacity="100000" delay="0.1"></link>
<link id="59" destination="1" source="2"  capacity="100000" delay="0.1"></link>
<link id="60" destination="2" source="3"  capacity="100000" delay="0.1"></link>
<link id="61" destination="3" source="4"  capacity="100000" delay="0.1"></link>
<link id="62" destination="4" source="5"  capacity="100000" delay="0.1"></link>
<link id="63" destination="5" source="6"  capacity="100000" delay="0.1"></link>
<link id="64" destination="6" source="7"  capacity="100000" delay="0.1"></link>
<link id="65" destination="7" source="8" capacity="100000" delay="0.1"></link>
<link id="66" destination="8" source="9"  capacity="100000" delay="0.1"></link>
<link id="67" destination="9" source="10" capacity="100000" delay="0.1"></link>
<link id="68" destination="10" source="11" capacity="100000" delay="0.1"></link>
<link id="69" destination="11" source="12" capacity="100000" delay="0.1"></link>
<link id="70" destination="12" source="13" capacity="100000" delay="0.1"></link>
<link id="71" destination="13" source="14" capacity="100000" delay="0.1"></link>
<link id="72" destination="14" source="15" capacity="100000" delay="0.1"></link>
<link id="73" destination="15" source="16" capacity="100000" delay="0.1"></link>
<link id="74" destination="16" source="17" capacity="100000" delay="0.1"></link>
<link id="75" destination="17" source="18" capacity="100000" delay="0.1"></link>
<link id="76" destination="18" source="19" capacity="100000" delay="0.1"></link>
<link id="77" destination="19" source="20" capacity="100000" delay="0.1"></link>
<link id="78" destination="20" source="21" capacity="100000" delay="0.1"></link>
<link id="79" destination="21" source="22" capacity="100000" delay="0.1"></link>
<link id="80" destination="22" source="23" capacity="100000" delay="0.1"></link>
<link id="81" destination="23" source="24" capacity="100000" delay="0.1"></link>
<link id="82" destination="24" source="25" capacity="100000" delay="0.1"></link>
<link id="83" destination="25" source="26" capacity="100000" delay="0.1"></link>
<link id="84" destination="26" source="27" capacity="100000" delay="0.1"></link>
<link id="85" destination="27" source="28" capacity="100000" delay="0.1"></link>
<link id="86" destination="28" source="0" capacity="100000" delay="0.1"></link>
<link id="87" destination="0" source="3" capacity="100000" delay="0.1"></link>
<link id="88" destination="3"  source="6" capacity="100000" delay="0.1"></link>
<link id="89" destination="6"  source="9"  capacity="100000" delay="0.1"></link>
<link id="90" destination="9"  source="12" capacity="100000" delay="0.1"></link>
<link id="91" destination="12" source="15" capacity="100000" delay="0.1"></link>
<link id="92" destination="15" source="18" capacity="100000" delay="0.1"></link>
<link id="93" destination="18" source="21" capacity="100000" delay="0.1"></link>
<link id="94" destination="21" source="24" capacity="100000" delay="0.1"></link>
<link id="95" destination="24" source="27" capacity="100000" delay="0.1"></link>
<link id="96" destination="27" source="1" capacity="100000" delay="0.1"></link>
<link id="97" destination="1" source="4" capacity="100000" delay="0.1"></link>
<link id="98" destination="4"  source="7" capacity="100000" delay="0.1"></link>
<link id="99" destination="7"  source="10"  capacity="100000" delay="0.1"></link>
<link id="100" destination="10" source="13" capacity="100000" delay="0.1"></link>
<link id="101" destination="13" source="16" capacity="100000" delay="0.1"></link>
<link id="102" destination="16" source="19" capacity="100000" delay="0.1"></link>
<link id="103" destination="19" source="22" capacity="100000" delay="0.1"></link>
<link id="104" destination="22" source="25" capacity="100000" delay="0.1"></link>
<link id="105" destination="25" source="28" capacity="100000" delay="0.1"></link>
<link id="106" destination="28" source="2" capacity="100000" delay="0.1"></link>
<link id="107" destination="2" source="5" capacity="100000" delay="0.1"></link>
```

```
    <link id="108" destination="5" source="8" capacity="100000" delay="0.1"></link>
    <link id="109" destination="8" source="11"   capacity="100000" delay="0.1"></link>
    <link id="110" destination="11" source="14" capacity="100000" delay="0.1"></link>
    <link id="111" destination="14" source="17" capacity="100000" delay="0.1"></link>
    <link id="112" destination="17" source="20" capacity="100000" delay="0.1"></link>
    <link id="113" destination="20" source="23" capacity="100000" delay="0.1"></link>
    <link id="114" destination="23" source="26" capacity="100000" delay="0.1"></link>
    <link id="115" destination="26" source="0" capacity="100000" delay="0.1"></link>

</network>
</admission_state>
```

C.2 mich_net.xml

```
<?xml version= '1.0'?>
<admission_state>
<network>
    <node id = "0" name = "External_Network"></node>
    <node id = "1" name = "Norlight_Chicago"></node>
    <node id = "2" name = "Kalamazod/Protage"></node>
    <node id = "3" name = "Ann_Arbor"></node>
    <node id = "4" name = "Detroit"></node>
    <node id = "5" name = "Houghton"></node>
    <node id = "6" name = "Marquette"></node>
    <node id = "7" name = "Muskegon"></node>
    <node id = "8" name = "Frame_Relay_8"></node>
    <node id = "9" name = "Grand_Rapids/Byron_Center"></node>
    <node id = "10" name = "Lansing/East_Lansing"></node>
    <node id = "11" name = "Flint"></node>
    <node id = "12" name = "Frame_Relay_12"></node>
    <node id = "13" name = "Frame_Relay_13"></node>
    <node id = "14" name = "Southfield"></node>
    <node id = "15" name = "Ypsilanti"></node>
    <node id = "16" name = "Rochester"></node>
    <node id = "17" name = "Frame_Relay_17"></node>
    <node id = "18" name = "Frame_Relay_18"></node>
    <node id = "19" name = "Traverse_City"></node>
    <node id = "20" name = "Big_Rapids"></node>
    <node id = "21" name = "Mount_Pleasant"></node>
    <node id = "22" name = "Saginaw"></node>
    <node id = "23" name = "Dearborn"></node>
    <node id = "24" name = "Sault_St._Marie"></node>
    <node id = "25" name = "Frame_Relay_25"></node>
    <node id = "26" name = "Frame_Relay_26"></node>
    <node id = "27" name = "Berrien_Springs"></node>
    <node id = "28" name = "Marshall"></node>
    <node id = "29" name = "Frame_Relay_29"></node>
    <node id = "30" name = "Jackson"></node>

    <link id="0" source="0" destination="1"    capacity="3000" delay="0.100000"></link>
    <link id="1" source="0" destination="3"    capacity="822" delay="0.100000"></link>
    <link id="2" source="1" destination="5"    capacity="155" delay="0.100000"></link>
    <link id="3" source="1" destination="6"    capacity="155" delay="0.100000"></link>
    <link id="4" source="1" destination="2"    capacity="2500" delay="0.100000"></link>
    <link id="5" source="1" destination="3"    capacity="2500" delay="0.100000"></link>
    <link id="6" source="2" destination="7"    capacity="45" delay="0.100000"></link>
```

```
<link id="7" source="2" destination="8"     capacity="135" delay="0.100000"></link>
<link id="8" source="2" destination="9"     capacity="2500" delay="0.100000"></link>
<link id="9" source="2" destination="10"    capacity="2500" delay="0.100000"></link>
<link id="10" source="2" destination="3"    capacity="2500" delay="0.100000"></link>
<link id="11" source="3" destination="12"     capacity="6" delay="0.100000"></link>
<link id="12" source="3" destination="10"    capacity="622" delay="0.100000"></link>
<link id="13" source="3" destination="11"    capacity="155" delay="0.100000"></link>
<link id="14" source="3" destination="13"     capacity="45" delay="0.100000"></link>
<link id="15" source="3" destination="14"   capacity="5000" delay="0.100000"></link>
<link id="16" source="3" destination="15"    capacity="100" delay="0.100000"></link>
<link id="17" source="4" destination="16"   capacity="1000" delay="0.100000"></link>
<link id="18" source="5" destination="17"     capacity="45" delay="0.100000"></link>
<link id="19" source="6" destination="17"     capacity="45" delay="0.100000"></link>
<link id="20" source="17" destination="24"    capacity="45" delay="0.100000"></link>
<link id="21" source="7" destination="18"     capacity="45" delay="0.100000"></link>
<link id="22" source="8" destination="27"     capacity="45" delay="0.100000"></link>
<link id="23" source="8" destination="19"     capacity="45" delay="0.100000"></link>
<link id="24" source="8" destination="28"     capacity="45" delay="0.100000"></link>
<link id="25" source="9" destination="20"     capacity="45" delay="0.100000"></link>
<link id="26" source="9" destination="10"   capacity="2000" delay="0.100000"></link>
<link id="27" source="10" destination="29"    capacity="45" delay="0.100000"></link>
<link id="28" source="29" destination="30"    capacity="45" delay="0.100000"></link>
<link id="29" source="10" destination="21"   capacity="155" delay="0.100000"></link>
<link id="30" source="21" destination="25"    capacity="45" delay="0.100000"></link>
<link id="31" source="10" destination="11"   capacity="155" delay="0.100000"></link>
<link id="32" source="11" destination="22"    capacity="45" delay="0.100000"></link>
<link id="33" source="22" destination="26"    capacity="45" delay="0.100000"></link>
<link id="34" source="13" destination="16"    capacity="45" delay="0.100000"></link>
<link id="35" source="13" destination="14"    capacity="45" delay="0.100000"></link>
<link id="36" source="14" destination="23"    capacity="45" delay="0.100000"></link>
<link id="37" source="0" destination="4"     capacity="155" delay="0.100000"></link>


<link id="38" destination="0" source="1"    capacity="3000" delay="0.100000"></link>
<link id="39" destination="0" source="3"     capacity="822" delay="0.100000"></link>
<link id="40" destination="1" source="5"     capacity="155" delay="0.100000"></link>
<link id="41" destination="1" source="6"     capacity="155" delay="0.100000"></link>
<link id="42" destination="1" source="2"    capacity="2500" delay="0.100000"></link>
<link id="43" destination="1" source="3"    capacity="2500" delay="0.100000"></link>
<link id="44" destination="2" source="7"      capacity="45" delay="0.100000"></link>
<link id="45" destination="2" source="8"     capacity="135" delay="0.100000"></link>
<link id="46" destination="2" source="9"    capacity="2500" delay="0.100000"></link>
<link id="47" destination="2" source="10"   capacity="2500" delay="0.100000"></link>
<link id="48" destination="2" source="3"    capacity="2500" delay="0.100000"></link>
<link id="49" destination="3" source="12"     capacity="6" delay="0.100000"></link>
<link id="50" destination="3" source="10"    capacity="622" delay="0.100000"></link>
<link id="51" destination="3" source="11"    capacity="155" delay="0.100000"></link>
<link id="52" destination="3" source="13"     capacity="45" delay="0.100000"></link>
<link id="53" destination="3" source="14"   capacity="5000" delay="0.100000"></link>
<link id="54" destination="3" source="15"    capacity="100" delay="0.100000"></link>
<link id="55" destination="4" source="16"   capacity="1000" delay="0.100000"></link>
<link id="56" destination="5" source="17"     capacity="45" delay="0.100000"></link>
<link id="57" destination="6" source="17"     capacity="45" delay="0.100000"></link>
<link id="58" destination="17" source="24"    capacity="45" delay="0.100000"></link>
<link id="59" destination="7" source="18"     capacity="45" delay="0.100000"></link>
<link id="60" destination="8" source="27"     capacity="45" delay="0.100000"></link>
<link id="61" destination="8" source="19"     capacity="45" delay="0.100000"></link>
```

```
    <link id="62" destination="8" source="28"     capacity="45" delay="0.100000" ></link>
    <link id="63" destination="9" source="20"     capacity="45" delay="0.100000" ></link>
    <link id="64" destination="9" source="10"   capacity="2000" delay="0.100000" ></link>
    <link id="65" destination="10" source="29"    capacity="45" delay="0.100000" ></link>
    <link id="66" destination="29" source="30"    capacity="45" delay="0.100000" ></link>
    <link id="67" destination="10" source="21"   capacity="155" delay="0.100000" ></link>
    <link id="68" destination="21" source="25"    capacity="45" delay="0.100000" ></link>
    <link id="69" destination="10" source="11"   capacity="155" delay="0.100000" ></link>
    <link id="70" destination="11" source="22"    capacity="45" delay="0.100000" ></link>
    <link id="71" destination="22" source="26"    capacity="45" delay="0.100000" ></link>
    <link id="72" destination="13" source="16"    capacity="45" delay="0.100000" ></link>
    <link id="73" destination="13" source="14"    capacity="45" delay="0.100000" ></link>
    <link id="74" destination="14" source="23"    capacity="45" delay="0.100000" ></link>
    <link id="75" destination="0" source="4"     capacity="155" delay="0.100000" ></link>

</network>
</admission_state>
```

Appendix D

Model Implementation Source Code

This Appendix contains the source code for the Model Implementation of the Utility Model based Network Admission Controller used in the evaluation experiments. The supporting framework for performing the experiments is not included. Refer to inline comments for explanation and details on operation.

D.1 network_model.h

```
#ifndef __network_model_h__
#define __network_model_h__

/*
 * This file defines the structures used in the network model for
 * our admission controller.
 *
 * Created 04/08/2003 Eric Gowland
 */



/* Structure Names */
struct network;
struct link;
struct node;
struct path_node;

//This Represents an entire network.
typedef struct network {
   struct link *links; //The links in the network.
   struct node *nodes; //The nodes in the network.
   int link_count; //The number of links.
   int node_count; //The number of nodes.
   //int bidirectional; //Boolean indicating whether network has bi links.
} NETWORK;
```

```
//This is a node in the network.
typedef struct node {
   int node_id; //Unique ID for this node.
   char* name;
   int *ingress_link_ids; //IDs of incoming links.
   int *egress_link_ids; //IDs of outgoing links
   int ingress_link_count; //Number of incoming links.
   int egress_link_count; //Number of outgoing links.
} NODE;

//A uni-directional link in the network...
typedef struct link {
   int link_id; //Unique ID of this link.
   int start_node_index; //Originating node for this link.
   int finish_node_index; //Terminating node for this link.
   long int capacity; //The capacity of this link.
   long int used_capacity; //The capacity of this link currently in use.
   double cost; //The 'cost' ('delay') of this link.
} LINK;

//This represents a path through the network.
typedef struct path {
   int *node_ids; //ID's of the nodes on the path, preferably in order.
   int *link_ids; //ID's of the links on the path, preferrably in order.
   int node_count; //Number of nodes on the path.
   int link_count; //Number of links on the path - should be node_count - 1.
} PATH;


#endif /* __netmodel_h__ */
```

D.2 sla_model.h

```
#ifndef __sla_model_h__
#define __sla_model_h__

#include "network_model.h"

/**
 * Defines data structures used for primary
 * SLA controller interface.
 *
 * Created 21/10/2003 Eric Gowland
 */

//Structs
struct sla;
struct qos;

//Defines an SLA request to the system.
typedef struct sla {
   int sla_id; //Unique request id.
   int start_node_id; //Unique id of the start node.
   int finish_node_id; //Unique id of the end node.
   int active_profile_index; //Index of currently active QoS profile.
   struct qos_profile *profiles; //The QoS profiles for this SLA.
```

```
    int profile_count; //The number of qos profiles for this sla.
    PATH *active_path;
} SLA;

typedef struct qos_profile {
    int profile_id; //ID of this profile, unique to a single SLA.
    int utility; //Amount of utility offered for this SLA
    int capacity_requirement; //Required capacity on each link for this SLA.
    double max_cost_constraint; //Maximum cost of any path chosen for this SLA.
} QOS_PROFILE;

#endif
```

D.3 xml_parser.h

```
#ifndef __xml_sla_parser_h__
#define __xml_sla_parser_h__

#include "sla_model.h"
#include "network_model.h"
#include <stdio.h>
#include <libxml/parser.h>
#include <libxml/xpath.h>

/**
 * Header for the SLA parser
 * Defines methods for parsing an XML file containing SLA info.
 *
 * Created 09/09/2003 Eric Gowland
 */

//Struct definition:
typedef struct admission_state{
    NETWORK* network;
    SLA* sla_list;
    int sla_count;
} ADMISSION_STATE;

//Function definitions...
//
//Parses the admission state file (SLAs and network)
//Parsed data returned in pointers.
//void parse_xml_admission_state(char *file_name, SLA* sla_list, int* sla_count, NETWORK* network);
ADMISSION_STATE* parse_xml_admission_state(char *file_name, ADMISSION_STATE* state);

//Writes given data to xml format. Sprintf to given target array...
void generate_xml_admission_state(FILE *target, ADMISSION_STATE* state);
#endif
```

D.4 sla_admission_controller.h

```
#ifndef __sla_admission_controller_h__
#define __sla_admission_controller_h__
```

```
#include "network_model.h"
#include "sla_model.h"

/**
 * Defines functions used for primary
 * SLA controller interface.
 *
 * Created 04/08/2003 Eric Gowland
 */

//The main admission control function...
//Updates status of slas & network.
SLA* admit_slas(SLA slas[], int sla_count, NETWORK network, int path_count);

#endif
```

D.5 network_path_finder.h

```
#ifndef __network_path_finder_h__
#define __network_path_finder_h__

#include "network_model.h"

/*
 * This header defines the interface for a network path finder.
 * For example, it could be implemented as Martin's K shortest
 * loopless paths algorithm.
 *
 * Created 04/08/2003 Eric Gowland
 */

//This function returns an array containing the specified number of paths.
PATH* find_paths(NETWORK network, int number_of_paths, int source_node_id, int destination_node_id);
//Method for printing a path - useful for debugging.
void print_path(PATH *path);

#endif
```

D.6 sla_session_builder.h

```
#ifndef __sla_session_builder_h__
#define __sla_session_builder_h__

#include "utility_model.h"
#include "network_model.h"
#include "sla_model.h"

/**
 * This defines the interface for the module that translates
 * SLA requests into Utility model sessions.network_path_finder
 *
 * Created 04/08/2003 Eric Gowland
 */
```

```
//Builds session (group) structures based on the SLAs.
SESSION* build_sessions(SLA slas[], int request_count, NETWORK network, int path_count);

//Builds resource structures for the utility model.
RESOURCE* build_resources(NETWORK network);

#endif
```

D.7 utility_model.h

```
#ifndef __utility_model_h__
#define __utility_model_h__

/**
 * Header for the utility model module.
 * Defines data strucutres used in the Utility Model.
 * Defines methods accesible in the Utility Model
 *
 * Created 04/08/2003 Eric Gowland, ECE Department, University of Victoria
 */

// Structure names...
struct session;
struct profile;
struct resource;
struct resource_requirement;


//Defines a single session for the utility model.
//These correspond to MMKP groups
typedef struct session {
   int session_id; //Unique session id.
   int active_profile_index; //Currently active profile. 0 is an inactive profile
   struct profile *profiles; //Available profiles. 0 is an inactive profile
   int profile_count; //Number of available profiles.
} SESSION;

//Defines a single resource requirement within a
//profile (group) for the utility model.
typedef struct resource_requirement{
   int resource_id; //The ID of the resource required.
   int requirement; //The amount of the resource required.
} RESOURCE_REQUIREMENT;


//Defines a single profile within a session for the utility model.
//These correspond to MMKP Items within a group.
typedef struct profile {
   int profile_id; //ID, unique within session.
   int utility; //Utility offered for this profile.
   int resource_count;    //Number of resource requirements.
   struct resource_requirement *requirements; //Resource requirments for this profile.
} PROFILE;
```

```
//Defines a Resource for the Utility Model.
typedef struct resource{
   int resource_id; //Unique ID for this resource.
   int committed_resource; //Amount of this resource that is currently committed.
   int maximum_resource; //Maximum amount of this resource that can be committed.
} RESOURCE;

//Function definitions...

//Applies the utility model solution to the problem specified.
//Upon completion, the session structs will have their current profiles set
//appropriately, and the resources will have there committed values adjusted.
int solve_knapsack(SESSION sessions[], int session_count, RESOURCE available_resources[], int resource_count);

//Debugging function which prints a UM problem state
void print_sessions(SESSION* sesions, int session_count);

//Debugging function which prints a UM problem state
void print_resources(RESOURCE* sesions, int resource_count);
#endif
```

D.8 start_controller.c

```
#include <stdlib.h>
#include <stdio.h>
#include "sla_admission_controller.h"
#include "xml_parser.h"


/**
 * Main executable for the Model Implementation
 * Admission Controller
 *
 * Created by Eric Gowland
 */
int main(int argc, char **argv) {
   char *admission_docname; //Admission State Document name.
   //Misc. variables
   int i, path_count,admission_count, j, utility, m_utility;
   int used_resource, max_resource;

   ADMISSION_STATE *state;//Admission State Variable.

   if (argc <= 2) {
      fprintf(stderr, "Usage: %s admission_docname path_count\n", argv[0]);
      return(1);
   }
  admission_docname = argv[1];
   path_count = atoi(argv[2]);

   if(path_count == 0){
      fprintf(stderr, "Invalid path_count parameter.\n");
      fprintf(stderr, "Usage: %s admission_docname path_count\n", argv[0]);
      return(1);
   }
```

```
    //Parse admission state...
    state = parse_xml_admission_state(admission_docname, state);

    //Perform admission control...
    state->sla_list = admit_slas(state->sla_list, state->sla_count, *(state->network), path_count);


    utility = 0;
    m_utility = 0;
    used_resource = 0;
    max_resource = 0;
    admission_count = 0;

    //Output admission state...
    generate_xml_admission_state(stdout, state);

    //Provide additional output reporting information...
    for(i = 0; i < state->sla_count; i++){
       if(state->sla_list[i].active_profile_index > -1){
          utility += state->sla_list[i].profiles[state->sla_list[i].active_profile_index].utility;
          admission_count++;
       }
       m_utility += state->sla_list[i].profiles[state->sla_list[i].profile_count - 1].utility;
    }

    //Additional output...
    for(i = 0; i < state->network->link_count; i++){
       if(state->network->links[i].used_capacity > state->network->links[i].capacity){
          printf("<!--_Exceeded_Capacity_on_link_%d_-->\n", state->network->links[i].link_id);
       }
       used_resource += state->network->links[i].used_capacity - 1;
       max_resource += state->network->links[i].capacity - 1;
    }

    free(state->sla_list);
    free(state->network);
    free(state);
    return(0);
}
```

D.9 xml_parser.c

```
#include "xml_parser.h"

//This file defines functions for reading and generating
//XML Admission State Documents.

//Parses the given SLA doc. Returns an array containing the slas. The integer
//pointer parameter is written with the size of the array.
SLA* parse_sla_list(xmlDocPtr doc, SLA* sla_list, int* sla_count);
//Parses the given NETWORK doc. Returns the network data stucture (pointer to).
NETWORK* parse_network(xmlDocPtr doc, NETWORK* network);
//generates xml text for given sla. Writes to given target array.
void sla_to_xml(FILE* target, SLA sla);
//generates xml text for given network. Writes to given target array.
void network_to_xml(FILE* target, NETWORK* network);
```

```
//Generates xml text for complete admission state info.
void generate_xml_admission_state(FILE* target, ADMISSION_STATE* state){
   int i;
   fprintf(target, "<?xml version='1.0' ?>\n");
   fprintf(target, "<admission_state>\n");
   //Print each SLA
   for(i = 0; i < state->sla_count; i++){
      sla_to_xml(target, state->sla_list[i]);
   }
   //Print the network
   network_to_xml(target, state->network);
   fprintf(target, "</admission_state>\n");
}


//Ouputs a single SLA as XML
void sla_to_xml(FILE* target, SLA sla){
   int i;
   fprintf(target, "<sla_id = \"%d\"\n", sla.sla_id);
   fprintf(target, "      source = \"%d\"\n",sla.start_node_id);
   fprintf(target, "      destination = \"%d\"\n", sla.finish_node_id);
   fprintf(target, "      active_profile = \"%d\">\n", sla.active_profile_index);
   if(sla.active_profile_index > -1 && sla.active_path != NULL){
      fprintf(target, "  <path>\n");
      for(i = 0; i < sla.active_path->node_count; i++){
         fprintf(target, "    <node_id = \"%d\"></node>\n", sla.active_path->node_ids[i]);
      }
      for(i = 0; i < sla.active_path->link_count; i++){
         fprintf(target, "    <link_id = \"%d\"></link>\n", sla.active_path->link_ids[i]);
      }
      fprintf(target, "  </path>\n");
   }
   //Output each QoS level
   for (i = 0; i < sla.profile_count; i++){
      fprintf(target, "  <qos  capacity = \"%d\"\n",sla.profiles[i].capacity_requirement);
      fprintf(target, "        delay = \"%f\"\n", sla.profiles[i].max_cost_constraint);
      fprintf(target, "        utility = \"%d\"></qos>\n", sla.profiles[i].utility);
   }
   fprintf(target, "</sla>\n");
}

//Outputs the given network as xml
void network_to_xml(FILE* target, NETWORK* network){
   int i;
   fprintf(target, "<network>\n");
   //Nodes...
   for(i = 0; i < network->node_count; i++){
         fprintf(target, "  <node_id = \"%d\"", network->nodes[i].node_id);
         if(network->nodes[i].name != NULL){
            fprintf(target, " name = \"%s\"", network->nodes[i].name);
         }
         fprintf(target, "></node>\n");
   }
   //Links...
   for(i = 0; i < network->link_count; i++){
```

```
        fprintf(target, "  <link_id=\"%d\" source=\"%d\" destination=\"%d\"  capacity=\"%d\" delay=\"%f\"></link>\n",
            network->links[i].link_id, network->links[i].start_node_index, network->links[i].finish_node_index, network
            ->links[i].capacity - 1, network->links[i].cost);
    }
    fprintf(target, "</network>\n");
}


//Retrieves Document for given file name...
xmlDocPtr getdoc (char *docname) {
    xmlDocPtr doc;
    doc = xmlParseFile(docname);

    if (doc == NULL ) {
        fprintf(stderr,"Document not parsed successfully. \n");
        return NULL;
    }
    return doc;
}

//void parse_xml_admission_state(char *file_name, SLA* sla_list, int* sla_count, NETWORK* network){
ADMISSION_STATE* parse_xml_admission_state(char *file_name, ADMISSION_STATE *state){
    xmlDocPtr doc;

    doc = getdoc(file_name);

    //Read network and SLA states from file...
    state = malloc(sizeof(ADMISSION_STATE));
    state->network = parse_network(doc, state->network);
    state->sla_list = parse_sla_list(doc, state->sla_list, &state->sla_count);

  xmlFreeDoc(doc);
  xmlCleanupParser();

    return state;
}

//Parses the named file, building Network data structures. Returns the array of structures.
NETWORK* parse_network(xmlDocPtr doc, NETWORK *network){
        xmlNodeSetPtr nodeset;
        xmlXPathObjectPtr result;
            xmlNodePtr cur;
        xmlXPathContextPtr context;
        int i,j;
            int *id_array;

            //Parse XML
            network = malloc(sizeof(NETWORK));
        context = xmlXPathNewContext(doc);

            result = xmlXPathEvalExpression("//network", context);
        if(xmlXPathNodeSetIsEmpty(result->nodesetval)){
                fprintf(stderr, "No Network Info Found.\n");
                return NULL;
        }
            if(result) {
               nodeset = result->nodesetval;
```

```
        cur = nodeset->nodeTab[0];
        context->node = cur;
        xmlXPathFreeObject(result);
    }


    //First, do the nodes...
result = xmlXPathEvalExpression("node", context);
if(xmlXPathNodeSetIsEmpty(result->nodesetval)){
        fprintf(stderr, "No Node Info Found.\n");
        return NULL;
}
if (result) {
  nodeset = result->nodesetval;

       //Allocate for nodes...
       network->node_count = nodeset->nodeNr;
       network->nodes = malloc(network->node_count * sizeof(NODE));

       for (i=0; i < network->node_count; i++) {
          cur = nodeset->nodeTab[i];
          network->nodes[i].node_id = atoi(xmlGetProp(cur,"id"));
          network->nodes[i].name = (xmlGetProp(cur,"name"));
          //xmlFree(cur);
       }
   xmlXPathFreeObject (result);
}

    //Now, do the links (and finish the nodes)...
result = xmlXPathEvalExpression("link", context);
if(xmlXPathNodeSetIsEmpty(result->nodesetval)){
            xmlXPathFreeContext(context);
            xmlXPathFreeObject (result);
                xmlFreeDoc(doc);
        fprintf(stderr, "No Link Info Found.\n");
        return NULL;
}
if (result) {
  nodeset = result->nodesetval;

       //Allocate for links...
       network->link_count = nodeset->nodeNr;
       network->links = malloc(network->link_count * sizeof(LINK));

       //Allocate node in/out counters stuff...
       for(i = 0; i < network->node_count; i++){
          network->nodes[i].egress_link_count = 0;
          network->nodes[i].ingress_link_count = 0;
          network->nodes[i].egress_link_ids = malloc(network->link_count * sizeof(int));
          network->nodes[i].ingress_link_ids = malloc(network->link_count * sizeof(int));
       }

       for (i=0; i < network->link_count; i++) {
          cur = nodeset->nodeTab[i];
          network->links[i].link_id = atoi(xmlGetProp(cur,"id"));
          network->links[i].start_node_index = atoi(xmlGetProp(cur,"source"));
          //Update egress
```

```
                for (j = 0; j < network->node_count; j++){
                    if(network->nodes[j].node_id == network->links[i].start_node_index){
                        network->nodes[j].egress_link_ids[network->nodes[j].egress_link_count] = network->links[i].
                            link_id;
                        network->nodes[j].egress_link_count++;
                    }
                }
                network->links[i].finish_node_index = atoi(xmlGetProp(cur,"destination"));
                //update ingress
                for (j = 0; j < network->node_count; j++){
                    if(network->nodes[j].node_id == network->links[i].finish_node_index){
                        network->nodes[j].ingress_link_ids[network->nodes[j].ingress_link_count] = network->links[i].
                            link_id;
                        network->nodes[j].ingress_link_count++;
                    }
                }
                //Add 1 for mathematical fudging this is to avoid 0 divides later on...
                network->links[i].capacity = atoi(xmlGetProp(cur,"capacity")) + 1;
                network->links[i].used_capacity = 1;
                network->links[i].cost = atof(xmlGetProp(cur,"delay"));
            }
        }

            //Resize egres/ingress arrays...
            for(i = 0; i < network->node_count; i++){
                network->nodes[i].egress_link_ids = realloc((network->nodes[i]).egress_link_ids , network->nodes[i].
                    egress_link_count * sizeof(int));
                network->nodes[i].ingress_link_ids = realloc((network->nodes[i]).ingress_link_ids , network->nodes[i].
                    ingress_link_count * sizeof(int));
            }

            //Cleanup
        xmlXPathFreeContext(context);
        xmlXPathFreeObject (result);
        return (network);
}


//Parses the named file, building sla data structures. Returns the array of structures. size stored
//in sla_count
SLA* parse_sla_list(xmlDocPtr doc, SLA* sla_list, int *sla_count){
        xmlNodeSetPtr sla_nodeset, qos_nodeset, path_nodeset;
        xmlXPathObjectPtr sla_result, qos_result, path_result;
        int i,j;
            xmlNodePtr cur_sla, cur_path;
        xmlXPathContextPtr context;

        context = xmlXPathNewContext(doc);

            //Parse XML...
        sla_result = xmlXPathEvalExpression("//sla", context);
        if(xmlXPathNodeSetIsEmpty(sla_result->nodesetval)){
                //fprintf(stderr, "No SLA's Found.\n");
                    xmlXPathFreeContext(context);
                    xmlXPathFreeObject (sla_result);
                return NULL;
```

```
}

if (sla_result) {
  sla_nodeset = sla_result->nodesetval;

      //Allocate space for slas...
      *sla_count = sla_nodeset->nodeNr;
      sla_list = malloc(*sla_count * sizeof(SLA));

      for (i=0; i < *sla_count; i++) {
         cur_sla = sla_nodeset->nodeTab[i];
         context->node = cur_sla;
         sla_list[i].sla_id = atoi(xmlGetProp(cur_sla,"id"));
         sla_list[i].start_node_id = atoi(xmlGetProp(cur_sla,"source"));
         sla_list[i].finish_node_id = atoi(xmlGetProp(cur_sla,"destination"));
         sla_list[i].active_profile_index = atoi(xmlGetProp(cur_sla,"active_profile"));

         //Store path if present
         path_result = xmlXPathEvalExpression("path", context);
         if(!xmlXPathNodeSetIsEmpty(path_result->nodesetval) && path_result){
            sla_list[i].active_path = malloc(sizeof(PATH));
            context->node = path_result->nodesetval->nodeTab[0];
            xmlXPathFreeObject(path_result);
            //Find nodes..
            path_result = xmlXPathEvalExpression("node", context);
            if(!xmlXPathNodeSetIsEmpty(path_result->nodesetval) && path_result){
               path_nodeset = path_result->nodesetval;
               //Allocate space for node arrays...
               sla_list[i].active_path->node_count = path_nodeset->nodeNr;
               sla_list[i].active_path->node_ids = malloc(sizeof(int) * path_nodeset->nodeNr);
               for(j = 0; j < path_nodeset->nodeNr; j++){
                  cur_path = path_nodeset->nodeTab[j];
                  sla_list[i].active_path->node_ids[j] = atoi(xmlGetProp(cur_path, "id"));
               }
            } else {
               sla_list[i].active_path->node_count = 0;
            }
            xmlXPathFreeObject(path_result);
            //Find links...
            path_result = xmlXPathEvalExpression("link", context);
            if(!xmlXPathNodeSetIsEmpty(path_result->nodesetval) && path_result){
               path_nodeset = path_result->nodesetval;
               //Allocate space for link arrays...
               sla_list[i].active_path->link_count = path_nodeset->nodeNr;
               sla_list[i].active_path->link_ids = malloc(sizeof(int) * path_nodeset->nodeNr);
               for(j = 0; j < path_nodeset->nodeNr; j++){
                  cur_path = path_nodeset->nodeTab[j];
                  sla_list[i].active_path->link_ids[j] = atoi(xmlGetProp(cur_path, "id"));
               }
            } else {
               sla_list[i].active_path->link_count = 0;
            }
            xmlXPathFreeObject(path_result);

         }

         //Build QoS profiles...
```

```
                context->node = cur_sla;
                qos_result = xmlXPathEvalExpression("qos", context);
                if(!xmlXPathNodeSetIsEmpty(qos_result->nodesetval) && qos_result){
                   qos_nodeset = qos_result->nodesetval;
                   sla_list[i].profile_count = qos_nodeset->nodeNr;
                   sla_list[i].profiles = malloc(qos_nodeset->nodeNr * sizeof(QOS_PROFILE));
                   for(j = 0; j < qos_nodeset->nodeNr; j++){
                      cur_sla = qos_nodeset->nodeTab[j];
                      sla_list[i].profiles[j].profile_id = j;
                      sla_list[i].profiles[j].utility = atoi(xmlGetProp(cur_sla,"utility"));
                      sla_list[i].profiles[j].capacity_requirement = atoi(xmlGetProp(cur_sla,"capacity"));
                      sla_list[i].profiles[j].max_cost_constraint = atof(xmlGetProp(cur_sla,"delay"));
                   }
                } else {
                   sla_list[i].profile_count = 0;
                   sla_list[i].profiles = NULL;
                }
                xmlXPathFreeObject(qos_result);

             }
          }

             //Cleanup
          xmlXPathFreeContext(context);
          xmlXPathFreeObject (sla_result);
          return (sla_list);
}
```

D.10 sla_admission_controller.c

```
#include "sla_admission_controller.h"
#include "sla_session_builder.h"
#include <stdlib.h>
#include <stdio.h>

//Number of paths through network that are considered.
/*
 * This is the primary function for an sla admission controller.
 *
 * Created 04/08/2003 Eric Gowland
 */

//Function called to admit some SLA's. Admitted slas data structure is modified.
SLA* admit_slas(SLA slas[], int sla_count, NETWORK network, int path_count){

   int i,j,k;

   SESSION *sessions;
   RESOURCE * resources;

   //Prep utility model data structures...
   //Indices of resources are assumed to be matched to indices of links.
   //Ditto with sessions/slas.

   sessions = build_sessions(slas, sla_count, network, path_count);
   resources = build_resources(network);
```

```
//Solve the knapsack problem...
solve_knapsack(sessions, sla_count, resources, network.link_count);

//Update SLAs:
//Determine admission levels...
for (i = 0; i < sla_count; i++){
   //default to not active...
   slas[i].active_profile_index = -1;
   //If session active...
   if(sessions[i].active_profile_index > 0){
      //Active... must find corresponding SLA qos level...
      //Compare each SLA qos level to session profiles
      //capacity use and utility...
      for (j = 0; j < slas[i].profile_count; j++){
         //If utility & resource requirement match...
         if(slas[i].profiles[j].utility == sessions[i].profiles[sessions[i].active_profile_index].utility && slas[i].
               profiles[j].capacity_requirement == sessions[i].profiles[sessions[i].active_profile_index].
               requirements[0].requirement){

            //Assume this is the level..
            slas[i].active_profile_index = j;

            //Allocate space for the path info...
            slas[i].active_path = malloc(sizeof(PATH));
            slas[i].active_path->link_count = sessions[i].profiles[sessions[i].active_profile_index].resource_count;
            slas[i].active_path->link_ids = malloc(slas[i].active_path->link_count * sizeof(int));
            slas[i].active_path->node_count = slas[i].active_path->link_count + 1;
            slas[i].active_path->node_ids = malloc(slas[i].active_path->node_count * sizeof(int));

            //Copy the path info...
            //Links
            for(k = 0; k < slas[i].active_path->link_count; k++){
               slas[i].active_path->link_ids[k] = sessions[i].profiles[sessions[i].active_profile_index].requirements
                     [k].resource_id;
            }
            //Nodes
            for(k = 0; k < slas[i].active_path->link_count; k++){
               slas[i].active_path->node_ids[k] = network.links[slas[i].active_path->link_ids[k]].start_node_index;
            }
            slas[i].active_path->node_ids[k] = slas[i].finish_node_id;

            break;
         }
      }
   } else {
   //IF we didn't find one, sla will remain listed as not admitted (active index < 0).
   //and the active path will be null.
      slas[i].active_path = NULL;
   }
}

//Update Network:
for (i = 0; i < network.link_count; i++){
   network.links[i].used_capacity = resources[i].committed_resource;
}
```

```
    //Free session and resource memory...
    for(i = 0; i < sla_count; i++){
       for(j = 0; j < sessions[i].profile_count; j++){
          free(sessions[i].profiles[j].requirements);
       }
       free(sessions[i].profiles);
    }
    free(sessions);
    free(resources);

    return slas;
}
```

D.11 sla_session_builder.c

```
#include "sla_session_builder.h"
#include "network_path_finder.h"
#include <stdlib.h>
#include <stdio.h>
#include <float.h>

//This function is used in sorting resource requirements.
int resource_index_compare(const RESOURCE_REQUIREMENT *r1, const RESOURCE_REQUIREMENT *r2);

/**
 * This implements the translation layer to convert
 * SLA requests to a format the
 * Utility Model engine can understand.
 *
 * Currently for a uni-directional network model.
 */
//Builds session structures based on the SLAs.
//path_count indicates how many alternate paths to consider for a given SLA
SESSION* build_sessions(SLA slas[], int request_count, NETWORK network, int path_count){

    int i,j,k,m,n;
    int profile_counter;
    SESSION *sessions;
    NODE *current_node;
    LINK *current_link;
    PATH *paths;
    double *path_costs;
    int session_found;

    sessions = malloc(request_count * sizeof(SESSION));

    for (i = 0; i < request_count; i++){
       //Assign session IDs.
       sessions[i].session_id = slas[i].sla_id;

       //Set active profile to the base (0), non active profile...
       sessions[i].active_profile_index = 0;

       //Find possible paths...
       paths = find_paths(network, path_count, slas[i].start_node_id, slas[i].finish_node_id);
```

```
//Find path costs.
path_costs = malloc(path_count * sizeof(double));
for(j = 0; j < path_count; j++){
   if (paths[j].node_count > 0) {
      path_costs[j] = 0;
      for(k = 0; k < paths[j].link_count; k++){
         path_costs[j] += network.links[paths[j].link_ids[k]].cost;
      }
   } else {
      //Empty path...
      path_costs[j] = DBL_MAX;
   }
}

//Now determine the total number of possible profiles.
sessions[i].profile_count = 1;
for(j = 0; j < slas[i].profile_count; j++){
   for(k = 0; k < path_count; k++){
      if(path_costs[k] <= slas[i].profiles[j].max_cost_constraint) sessions[i].profile_count++;
   }
}

//Create Sessions...
sessions[i].profiles = malloc(sessions[i].profile_count * sizeof(PROFILE));
//Create the base case...
sessions[i].profiles[0].profile_id = 0;
sessions[i].profiles[0].utility = 0;
sessions[i].profiles[0].resource_count = 0;
sessions[i].profiles[0].requirements = NULL;
profile_counter = 1;
//For each sla qos profile...
for(j = 0; j < slas[i].profile_count; j++){
   //for each path...
   for(k = 0; k < path_count; k++){
      //if it's a valid path...
      if(path_costs[k] <= slas[i].profiles[j].max_cost_constraint){
         //create a session
         sessions[i].profiles[profile_counter].profile_id = profile_counter;
         sessions[i].profiles[profile_counter].utility = slas[i].profiles[j].utility;
         sessions[i].profiles[profile_counter].resource_count = paths[k].link_count;
         sessions[i].profiles[profile_counter].requirements = malloc(sessions[i].profiles[profile_counter].
               resource_count * sizeof(RESOURCE_REQUIREMENT));
         //Build requirement requests, one for each link on path...
         for(m = 0; m < paths[k].link_count; m++){
            sessions[i].profiles[profile_counter].requirements[m].resource_id = paths[k].link_ids[m];
            sessions[i].profiles[profile_counter].requirements[m].requirement = slas[i].profiles[j].
                  capacity_requirement;
         }
         qsort(sessions[i].profiles[profile_counter].requirements, paths[k].link_count, sizeof(
               RESOURCE_REQUIREMENT), resource_index_compare);
         //Next profile...
         profile_counter++;
      }
   }
}// built profiles...

//Select current profile if appropriate...
```

```
      if(slas[i].active_profile_index > -1){

         //First we'll consume the appropriate resources in the network...
         for (j = 0; j < slas[i].active_path->link_count; j++){
            network.links[slas[i].active_path->link_ids[j]].used_capacity += slas[i].profiles[slas[i].
                  active_profile_index].capacity_requirement;
         }

         session_found = 0;
         //For each session..
         for(j = 0; j < sessions[i].profile_count; j++){
            //check if the utility matches...
            if(sessions[i].profiles[j].utility == slas[i].profiles[slas[i].active_profile_index].utility){
               //then check that the number of resources matches...
               if(slas[i].active_path->link_count == sessions[i].profiles[j].resource_count){
                  //NOw check the resource/link ids...
                  session_found = 1; //maybe...
                  for(k = 0; k < sessions[i].profiles[j].resource_count; k++){
                     if(slas[i].active_path->link_ids[k] != sessions[i].profiles[j].requirements[k].resource_id) {
                        session_found = 0; //not this one...
                        break;
                     }
                  }
                  //If we went through all of them and found no un-included links, we found it...
                  if(session_found) {
                     sessions[i].active_profile_index = j;
                     //break out of the loop...
                     break;
                  }
               }
            }
         }//checked 'em all..
      }//Finished session search..

   }

   //Free temporary memory.
   free(paths);
   free(path_costs);

   return sessions;
}

int resource_index_compare(const RESOURCE_REQUIREMENT *r1, const RESOURCE_REQUIREMENT *r2){
   return(r1->resource_id - r2->resource_id);
}

//Builds resource structures for the utility model, from the network admission model.
RESOURCE* build_resources(NETWORK network){
   int i;

   RESOURCE *resources;

   resources = malloc(network.link_count * sizeof(RESOURCE));
   for (i = 0; i < network.link_count; i++){
      resources[i].resource_id = network.links[i].link_id;
      resources[i].committed_resource = network.links[i].used_capacity;
```

```
        resources[i].maximum_resource = network.links[i].capacity;
    }

    return resources;
}
```

D.12 network_path_finder.c

```
#include "network_path_finder.h"
#include <stdlib.h>
#include <float.h>

/**
 * This contains the implementation of the network path finder.
 * It will implement Martin's K Shortest Loopless paths algorithm.
 *
 * Uses Uni-directional network model!
 *
 * Created 04/08/2003 Eric Gowland
 */


//Constant used in weighting for sorting links.
//HARD CODING IS BAD - a better way to do this should be developed.
#define ON_SHORTEST_PATH_WEIGHT 10

//Data structure to track candidate paths.
typedef struct candidate_node{
    PATH* path;
    int deviation_node_index;
    double path_cost;
    struct candidate_node* next;
} CANDIDATE_NODE;

//Internal Function definitions...
//Runs quicksort algorithm on links in given network - sorts indices array and costvalues. Recursive.
void quick_sort_links(NETWORK network, int indices[], double cost_values[], int left_bound, int right_bound);
//Returns 0 if no cycle exists in the given path.
int find_cycle(PATH* path);
//Returns 0 if the given link does NOT form a cycle with the given path.
int forms_cycle(PATH* path, LINK* link);
//Builds a Path from the node in path_trails with the given index to the destination node.
//Path_trails is assumed to be in the same order as the nodes in the given network.
PATH* build_shortest_path(NETWORK network, int path_trails[], int source_node_index);
//Run's Djikstra's Shortest path algorithm. path_trails and path_cost must be the same size as the node
//array in the given network. Populates these arrays with appropriate values.
void find_shortest_path(NETWORK network, int destination_node_id, int path_trails[], double path_cost[]);
//Copys the given path from between the given node indices of this paths node array.
PATH* copy_path(PATH* path, int start_index, int finish_index);
//Creates a path consisting of the two given paths appended together,using the given link.
PATH* join_paths(PATH* path_1, int joining_link_index, double joining_link_cost, PATH* path_2);
//Checks if two given paths contain the same links. return 1 if they do, 0 otherwise.
int check_duplicate_paths(PATH* path_1, PATH* path_2);
```

```
//Main function
//Returns the specified number of paths. These are the shortest paths from the specified source
//to destination in the given network. Indices given are for node array in network.
PATH* find_paths(NETWORK network, int number_of_paths, int source_node_index, int destination_node_index){

   int sorted_link_indices[network.link_count]; //Sorted indices of links.
   double link_costs[network.link_count]; //Link costs
   double shortest_path_costs[network.node_count]; //Djikstra costs
   int path_trails[network.node_count]; //Djikstra path trails
   PATH* paths; //The paths to be returned...
   int i,j; //A loop index variable.
   int path_counter = 0; //Number of paths currently found...
   PATH* current_path; //Current working path...
   PATH* sub_path; //Partial path used in algorithm...
   PATH *new_path, *new_path_end; //Used in building new ths.
   CANDIDATE_NODE* candidate_list; //The candidate list...
   CANDIDATE_NODE* new_candidate; //The candidate list...
   CANDIDATE_NODE* candidate_insertion_point; //Used in insertion sort of candidate list..
   int deviation_node_index; //Index in network node array of current deviation node.
   int deviation_link_index; //Index used in choosing link to create new path candidate.
   int current_path_deviation_node_index; //Index on current path for deviation..

   //Allocate space for paths.
   paths = malloc(number_of_paths * sizeof(PATH));
   for(i = 0; i < number_of_paths; i++){
      paths[i].node_ids = NULL;
      paths[i].link_ids = NULL;
      paths[i].node_count = 0;
      paths[i].link_count = 0;
   }


   //Run Djikstra's.
   find_shortest_path(network, destination_node_index, path_trails, shortest_path_costs);


   //Calculate cost of each link.
   for (i = 0; i < network.link_count; i++){
      link_costs[i] = shortest_path_costs[network.links[i].finish_node_index] - shortest_path_costs[network.links[i].
            start_node_index] + network.links[i].cost;
      //Give a -10 bonus to the path chosen as the 'shortest' - This breaks ties...
      if(path_trails[network.links[i].start_node_index] == network.links[i].finish_node_index) link_costs[i] -=
            ON_SHORTEST_PATH_WEIGHT;
      sorted_link_indices[i] = i;
   }

   //Sort the list of links...
   quick_sort_links(network, sorted_link_indices, link_costs, 0, network.link_count - 1);

   //Restore costs...
   for(i = 0; i < network.link_count; i++){
      if(link_costs[i] <= -1*ON_SHORTEST_PATH_WEIGHT) link_costs[i] += ON_SHORTEST_PATH_WEIGHT;
   }

   //Initialize X (vector of candidate paths).
   candidate_list = malloc(sizeof(CANDIDATE_NODE));
   candidate_list->path = build_shortest_path(network, path_trails, source_node_index);
```

```
//candidate_list->deviation_node_index = candidate_list->path->node_ids[0];
candidate_list->deviation_node_index = 0;
candidate_list->path_cost = shortest_path_costs[source_node_index];
candidate_list->next = NULL;


//Main loop of algorithm...
while(candidate_list != NULL && path_counter < number_of_paths){

   //Assume x list is sorted by increasing path_cost...
   current_path = candidate_list->path;
   //current_base_path_cost = candidate_list->path_cost;
   current_path_deviation_node_index = candidate_list->deviation_node_index;

   //Dispose of head of list...
   new_candidate = candidate_list;
   candidate_list = candidate_list->next;
   free(new_candidate);

   //If candidate has no cycles, add it to the solution...
   if (!(find_cycle(current_path))){
      //copy the path
      paths[path_counter].node_count = current_path->node_count;
      paths[path_counter].link_count = current_path->link_count;
      paths[path_counter].node_ids = malloc(current_path->node_count * sizeof(int));
      paths[path_counter].link_ids = malloc(current_path->link_count * sizeof(int));
      for(i = 0; i < paths[path_counter].link_count; i++){
         paths[path_counter].node_ids[i] = current_path->node_ids[i];
         paths[path_counter].link_ids[i] = current_path->link_ids[i];
      }
      paths[path_counter].node_ids[current_path->node_count - 1] = current_path->node_ids[current_path->node_count
            - 1];

      path_counter++;
      if(path_counter >= number_of_paths) break;
   }



   //Build subpath from Source to deviation_node_index...
   deviation_node_index = current_path->node_ids[current_path_deviation_node_index];
   sub_path = copy_path(current_path, 0, current_path_deviation_node_index);
   do{
      deviation_link_index = current_path->link_ids[current_path_deviation_node_index];
      for(i = 0; i < network.link_count; i++){
         if(sorted_link_indices[i] == deviation_link_index){
            deviation_link_index = i;
            break;
         }
      }

      //Find New Link...
      while(deviation_link_index+1 < network.link_count && network.links[sorted_link_indices[deviation_link_index
            +1]].start_node_index == deviation_node_index && forms_cycle(sub_path, &network.links[sorted_link_indices
            [deviation_link_index+1]])){
         deviation_link_index++;
```

```
}

if (deviation_link_index+1 < network.link_count && network.links[sorted_link_indices[deviation_link_index
      + 1]].start_node_index == deviation_node_index){

   //Build path starting at subpath and taking shortest path from
   //deviation point...
   new_path_end = build_shortest_path(network, path_trails, network.links[sorted_link_indices[
         deviation_link_index+1]].finish_node_index);
   //Join this path to the end of the existing path, at the given end point.
   new_path = join_paths(sub_path, sorted_link_indices[deviation_link_index+1], network.links[
         sorted_link_indices[deviation_link_index+1]].cost, new_path_end);

   //Create new candidate node...
   new_candidate = malloc(sizeof(CANDIDATE_NODE));
   new_candidate->path = new_path;

   //Determine deviation node... last node of longest common subpath between
   //This path and any of the found shortest paths...
   new_candidate->deviation_node_index = current_path_deviation_node_index;

   //Determine Cost...(Sum of c_bar for each link...)
   new_candidate->path_cost = 0;
   for(i = 0; i < new_candidate->path->link_count; i++){
      //new_candidate->path_cost += network.links[new_candidate->path->link_ids[i]].cost;
      new_candidate->path_cost += link_costs[new_candidate->path->link_ids[i]];
   }


   //Insertion sort new candidate into list...(Also checks if candidate exists already)
   if (candidate_list == NULL || new_candidate->path_cost < candidate_list->path_cost){
      new_candidate->next = candidate_list;
      candidate_list = new_candidate;
   } else {
      candidate_insertion_point = candidate_list;
      //Use i as a sentinal - if 1, might have a dupe, if 0 it isn't
      i = 0;
      //Check if first is dupe
      if(check_duplicate_paths(candidate_insertion_point->path, new_candidate->path)){
         i = 1;
      } else {
         //Check others...
         while(candidate_insertion_point->next != NULL && candidate_insertion_point->next->path_cost <=
               new_candidate->path_cost){
            //If they have the same cost, we must check for a dupe.
            if(check_duplicate_paths(candidate_insertion_point->next->path, new_candidate->path)){
               i = 1;
               break;
            }
            candidate_insertion_point = candidate_insertion_point->next;
         }
         //If we didn't find a dupe, insert it.
         if(i == 0){
            new_candidate->next = candidate_insertion_point->next;
            candidate_insertion_point->next = new_candidate;
         } else {
         }
```

```
            }
          }

          //Free temp paths...
          free(new_path_end);
          free(sub_path);

        }
        //Build subpath from S to deviation_node_index...
        current_path_deviation_node_index++;
        deviation_node_index = current_path->node_ids[current_path_deviation_node_index];

        //Get new sub_path
        sub_path = copy_path(current_path, 0, current_path_deviation_node_index);

      }while(!(deviation_node_index == destination_node_index || find_cycle(sub_path)));

      //Free current path when finished deviating from it...
      free(current_path);
    }// main loop


    //Free candidate memory...
    while(candidate_list != NULL){
      candidate_insertion_point = candidate_list->next;
      free(candidate_list->path);
      free(candidate_list);
      candidate_list = candidate_insertion_point;
    }

    return paths;
}

//Checks if given paths contain the same links...
int check_duplicate_paths(PATH* path_1, PATH* path_2){
    int i;
    if(path_1->link_count == path_2->link_count){
      for (i = 0; i < path_1->link_count; i++){
        if(path_1->link_ids[i] != path_2->link_ids[i]) return 0;
      }
      return 1;
    } else {
      return 0;
    }
}

//Checks given path for any cycles...
int find_cycle(PATH* path){
    int i, j;
    for (i = 0; i < path->node_count; i++){
      for (j = i+1; j < path->node_count; j++) {
        if(path->node_ids[i] == path->node_ids[j]) return 1;
      }
    }
    return 0;
}
```

```
//Checks if given link forms a cycle when added to given path.
//We assume the links is being added to the end of the path,
//and that the path doesn't currently have any cycles...
//i.e., we only check if the finish of the link is coincident
//with any nodes in the path...
int forms_cycle(PATH* path, LINK* link){
   int i;
   for (i = 0; i < path->node_count; i++){
      if(link->finish_node_index == path->node_ids[i]) return 1;
   }
   return 0;
}

//Copys given path until the node of the given index. Returns pointer to start of
//path
PATH* copy_path(PATH* path, int start_index, int finish_index){
   PATH* new_path;
   int i;
   if(path != NULL){
      new_path = malloc(sizeof(PATH));
      new_path->node_count = finish_index + 1 - start_index;
      new_path->link_count = new_path->node_count - 1;
      new_path->node_ids = malloc(new_path->node_count*sizeof(int));
      new_path->link_ids = malloc(new_path->link_count*sizeof(int));
      for(i = 0; i < new_path->node_count; i++) new_path->node_ids[i] = path->node_ids[i + start_index];
      for(i = 0; i < new_path->link_count; i++){
         new_path->link_ids[i] = path->link_ids[i + start_index];
      }
      return new_path;
   } else {
      return NULL;
   }
}


//Builds a path data structure from the given starting node to the destination
//indicated in the provided data structures
PATH* build_shortest_path(NETWORK network, int path_trails[], int source_index){
   PATH* new_path;
   int path_length = 0;
   int current_node_index = source_index;
   int i,j;

   while(current_node_index > -1){
      path_length++;
      current_node_index = path_trails[current_node_index];
   }

   new_path = malloc(sizeof(PATH));
   new_path->node_count = path_length;
   new_path->link_count = path_length-1;
   new_path->node_ids = malloc(sizeof(int) * new_path->node_count);
   new_path->link_ids = malloc(sizeof(int) * new_path->link_count);

   current_node_index = source_index;
   new_path->node_ids[0] = current_node_index;
```

```
    for(i = 0; i < new_path->link_count; i++){
        for(j = 0; j < network.nodes[current_node_index].egress_link_count; j++){
            if(network.links[network.nodes[current_node_index].egress_link_ids[j]].finish_node_index == path_trails[
                current_node_index]){
                new_path->link_ids[i] = network.nodes[current_node_index].egress_link_ids[j];
                break;
            }
        }
        //move to next node on path.
        current_node_index = path_trails[current_node_index];
        new_path->node_ids[i+1] = current_node_index;
    }


    return new_path;
}



//Prints given path
void print_path(PATH *path){
    int i;
    if (path->node_count == 0){
        printf("Empty path.\n");
        return;
    }
    for (i = 0; i < path->node_count-1; i++){
        printf("%d->", path->node_ids[i]);
    }
    printf("%d  %d nodes, %d links\n", path->node_ids[path->node_count-1], path->node_count, path->link_count);
}

//Quick sort algorithm to sort list of links on given array of values. Also sorts on node_index.
//on return, indexes & are sorted.
void quick_sort_links(NETWORK network, int indexes[], double values[], int left_bound, int right_bound){
    int pivot;
    int temp_link_index;
    double temp_value;
    int left_index = left_bound;
    int right_index = right_bound - 1;


    //If range is 0 or 1, or bounds are reversed, return.
    if (right_bound - left_bound <= 0) return;
    pivot = right_bound;

    //Main loop...
    while(left_index <= right_index){
        //Scan right until we find element larger than pivot or cross indexes...
        while((left_index <= right_index) && (network.links[indexes[left_index]].start_node_index < network.links[indexes[
            pivot]].start_node_index || (network.links[indexes[left_index]].start_node_index == network.links[indexes[
            pivot]].start_node_index && values[indexes[left_index]] <= values[indexes[pivot]])))
            left_index++;
        //Scan left until we find element larger than pivot or cross indexes...
        while((right_index >= left_index) && (network.links[indexes[right_index]].start_node_index > network.links[indexes
            [pivot]].start_node_index || (network.links[indexes[right_index]].start_node_index == network.links[indexes[
```

```
            pivot]].start_node_index && values[indexes[right_index]] >= values[indexes[pivot]])))
            right_index--;
        if (left_index < right_index){
            //Swap elements...
            temp_link_index = indexes[left_index];
            temp_value = values[indexes[left_index]];
            indexes[left_index] = indexes[right_index];
            //values[indexes[left_index]] = values[indexes[right_index]];
            indexes[right_index] = temp_link_index;
            //values[indexes[right_index]] = temp_value;
        }
    }
    //Place pivot in correct place...
    temp_link_index = indexes[left_index];
    temp_value = values[indexes[left_index]];
    indexes[left_index] = indexes[pivot];
    //values[indexes[left_index]] = values[indexes[pivot]];
    indexes[pivot] = temp_link_index;
    //values[indexes[pivot]] = temp_value;
    pivot = left_index;
    //Pivot has moved -- recur on both sides...
    quick_sort_links(network, indexes, values, left_bound, pivot - 1);
    quick_sort_links(network, indexes, values, pivot + 1, right_bound);

}//End quicksort.



//Runs Djikstra's Shortest path algorithm on the network model.
void find_shortest_path(NETWORK network, int destination_index, int path_trail[], double path_cost[]){
    int current_link_index;
    int i, j;
    double calculatedDelay;
    int current_node_index = destination_index;
    NODE *workList[network.node_count];

    //Reset path costs and trails..
    for (i = 0; i < network.node_count; i++){
        workList[i] = &network.nodes[i];
        path_cost[i] = DBL_MAX;
        path_trail[i] = -1;
    }
    //Start at the destination node...
    path_cost[current_node_index] = 0;
    for ( i = 0; i < network.node_count; i++){
        //remove current node from working list...
        workList[current_node_index] = NULL;
        //Update neighboring nodes paths if this node yields a shorter route.
        for (j = 0; j < (network.nodes[current_node_index]).ingress_link_count; j++){
            //Assumption here that id of link equal to index of link, and same for nodes.
            current_link_index = (network.nodes[current_node_index]).ingress_link_ids[j];

            calculatedDelay = network.links[current_link_index].cost + path_cost[current_node_index];

            if (path_cost[network.links[current_link_index].start_node_index] > calculatedDelay){
```

```
            //printf("Updated %d!\n", network.links[current_link_index].start_node_index);
            path_cost[network.links[current_link_index].start_node_index] = calculatedDelay;
            path_trail[network.links[current_link_index].start_node_index] = current_node_index;
         }
      }//Finish updating other nodes

      //Find next node and expand the "cloud"
      current_node_index = -1;
      for (j = 0; j < network.node_count; j++){
         if(workList[j] != NULL){
            if(path_cost[j] < DBL_MAX && (current_node_index == -1 || path_cost[j] < path_cost[current_node_index])){
               current_node_index = j;
            }
         }
      }//finish choosing next node.
   }//Loop back and process next selected node...
}//End Dijskstra's


//Creates a path consisting of the two given paths appended together
PATH* join_paths(PATH* path_1, int joining_link_id, double joining_link_cost, PATH* path_2){

   PATH* new_path;
   int i,j;

   new_path = malloc(sizeof(PATH));
   new_path->node_count = path_1->node_count + path_2->node_count;
   new_path->link_count = new_path->node_count - 1;
   new_path->node_ids = malloc(new_path->node_count * sizeof(int));
   new_path->link_ids = malloc(new_path->link_count * sizeof(int));

   //First part of new path is path 1...
   for (i = 0; i < path_1->node_count - 1; i++){
      new_path->node_ids[i] = path_1->node_ids[i];
      new_path->link_ids[i] = path_1->link_ids[i];
   }
   //Join with deviation node and specified link...
   new_path->node_ids[path_1->node_count - 1] = path_1->node_ids[path_1->node_count - 1];
   new_path->link_ids[path_1->node_count - 1] = joining_link_id;
   //Second part is path 2...
   for (i = 0; i < path_2->node_count-1; i++){
      new_path->node_ids[i + path_1->node_count] = path_2->node_ids[i];
      new_path->link_ids[i + path_1->node_count] = path_2->link_ids[i];
   }
   //The last node...
   new_path->node_ids[new_path->node_count - 1] = path_2->node_ids[path_2->node_count - 1];
   return new_path;
}
```

D.13 utility_model.c

```
#include "utility_model.h"
#include <math.h>

//Constants for utility model conditions.
#define DOWNGRADE_SOLUTION_FOUND -1
```

```
#define DOWNGRADE_SOLUTION_FAILED 0
#define FEASIBLE 1
#define NOT_FEASIBLE 0
#define MINIMUM_RESOURCE_CHANGE 0.00001
#define MINIMUM_PROFILE_INDEX 0

/*
 * This is the implementation of the utility model solution.
 * It uses Mostofa Akbar's MHEU Algorithm
 *
 * Created 04/08/2003 Eric Gowland, ECE Department, University of Victoria
 */


//Local Function prototypes
//
//Performs the best feasible upgrade to the solution. Returns utility gain.
int do_feasible_upgrade(SESSION sessions[], int session_count, RESOURCE resources[], int resource_count);
//Does the best non-feasible upgrade to the solution. Returns utility gain.
int do_non_feasible_upgrade(SESSION sessions[], int session_count, RESOURCE resources[], int resource_count);
//Does the best possible (lowest impact) downgrade to the solution.
int do_downgrade(SESSION sessions[], int session_count, RESOURCE resources[], int resource_count, int
    current_solution_gain);
//Caches the sessions current profile selection.
void cache_session_state(SESSION sessions[], int session_count, int profile_index_cache[]);
//Revives the sessions current profile selection from the given cache.
void restore_cached_session_state(SESSION sessions[], int session_count, int profile_index_cache[], RESOURCE resources
    []);
//Calculate change in aggregate resource consumption for an upgrade...
double calculate_change_in_aggregate_resource_consumption(PROFILE* old_profile, PROFILE* new_profile, RESOURCE resources
    [], int resource_change);
//Calculate scaled change in aggregate resource consumption for an upgrade...
double calculate_scaled_change_in_resource_consumption_available(PROFILE* old_profile, PROFILE* new_profile, RESOURCE
    resources[], int resource_change);
//Calculate scaled change in aggregate resource consumption for an upgrade...(overconsumed case)
double calculate_scaled_change_in_resource_consumption_overconsumed(PROFILE* old_profile, PROFILE* new_profile, RESOURCE
     resources[], int resource_change);
//Consume resources for the given profile.
void consume_resources(PROFILE* profile, RESOURCE resources[]);
//Free resources from the given profile.
void free_resources(PROFILE* profile, RESOURCE resources[]);
//Check if resources are available to accomodate the given profile
int check_upgrade_feasibility(PROFILE* old_profile, PROFILE* new_profile, RESOURCE resources[]);
//Check if given resource consumption is feasible.
int check_solution_feasibility(RESOURCE resources[], int resource_count);
//Finds an initially feasible solution.
int find_feasible_solution(SESSION sessions[], int session_count, RESOURCE resources[], int resource_count);


//Applies the utility model solution to the problem specified.
//Upon completion, the session structs will have their current profiles set
//appropriately, and the resources will have there committed values adjusted.
solve_knapsack(SESSION sessions[], int session_count, RESOURCE resources[], int resource_count){

   int cached_profile_indices[session_count];
   int upgrade_gain, downgrade_adjusted_gain;
```

```
    //If current solution is not feasible, return...
    if(find_feasible_solution(sessions, session_count, resources, resource_count) != FEASIBLE ) return;

    //Do feasible upgrades...
    while (do_feasible_upgrade(sessions, session_count, resources, resource_count) > 0){
    }
    //Try infeasible upgrade followed by downgrades...
    do{
       //Save current admission state...
       cache_session_state(sessions, session_count, cached_profile_indices);
       //Do a non feasible upgrade..
       upgrade_gain = do_non_feasible_upgrade(sessions, session_count, resources, resource_count);
       //Try to make solution feasible but still better with downgrades...
       if (upgrade_gain > 0) {
          downgrade_adjusted_gain = upgrade_gain;
          do{
             downgrade_adjusted_gain = do_downgrade(sessions, session_count, resources, resource_count,
                  downgrade_adjusted_gain);
          } while (downgrade_adjusted_gain > DOWNGRADE_SOLUTION_FAILED);
       }
       //If we failed to find a better solution, restore previous state.
       if (downgrade_adjusted_gain == DOWNGRADE_SOLUTION_FAILED) restore_cached_session_state(sessions, session_count,
            cached_profile_indices, resources);

       //Do any other upgrades that are now feasible...
       while (do_feasible_upgrade(sessions, session_count, resources, resource_count) > 0){}

       //try again, while we havn't failed...
    } while (upgrade_gain > 0 && downgrade_adjusted_gain == DOWNGRADE_SOLUTION_FOUND);
}

//Prints current session state. for debugging.
void print_sessions(SESSION* sessions, int session_count){
    int solution_utility = 0;
    int i;
    for(i = 0; i < session_count; i++){
       printf("Session: %d\n", sessions[i].session_id);
       printf("  Active_Profile: %d\n", sessions[i].active_profile_index);
       printf("  Utility_Yield_: %d\n", sessions[i].profiles[sessions[i].active_profile_index].utility);
       solution_utility += sessions[i].profiles[sessions[i].active_profile_index].utility;
    }
    printf("Solution_Utility: %d\n", solution_utility);
}

//Prints current resource state. for debugging.
void print_resources(RESOURCE* resources, int resource_count){
    int i;
    for(i = 0; i < resource_count; i++){
       printf("Resource_%d: %d/%d\n", resources[i].resource_id, resources[i].committed_resource, resources[i].
            maximum_resource);
    }
}


//Performs the best feasible upgrade to the solution. Returns 1 if successful -- 0 otherwise (?).
int do_feasible_upgrade(SESSION sessions[], int session_count, RESOURCE resources[], int resource_count){
```

```
//Indices for best aggregate resource consumption change
int best_delta_a_profile = -1;
int best_delta_a_session = -1;
//Indices for best value/aggregate resource consumption change
int best_delta_v_profile = -1;
int best_delta_v_session = -1;

//Variable for Change in Aggregate Resource Consumption
double delta_a = -1, best_delta_a = -1;
//Variable for Change in Value per Change in Aggregate Resource Consumption
double delta_v_per_a = -1, best_delta_v_per_a = -1;

int i,j;

PROFILE *old_profile, *new_profile;

//Consider each session...
for (i = 0; i < session_count; i++){
   //Check all possible upgrades from the current profile state for this session...
   old_profile = &sessions[i].profiles[sessions[i].active_profile_index];
   for (j = sessions[i].active_profile_index+1; j < sessions[i].profile_count; j++){
      new_profile = &sessions[i].profiles[j];
      if(check_upgrade_feasibility(old_profile, new_profile, resources) == FEASIBLE){
         //Find delta a
         delta_a = calculate_change_in_aggregate_resource_consumption(old_profile, new_profile, resources,
               resource_count);
         //Find delta v over delta a
         //delta_v_per_a = delta_a / (old_profile->utility - new_profile->utility);
         //Watch for division by 0...
         if(delta_a != 0){
            delta_v_per_a = (old_profile->utility - new_profile->utility) / delta_a;
         } else {
            delta_v_per_a = -1;
         }

         //Update best delta a
         if(delta_a > best_delta_a){
            best_delta_a = delta_a;
            best_delta_a_session = i;
            best_delta_a_profile = j;
         }

         //Update best delta a over delta v
         if(delta_v_per_a > best_delta_v_per_a){
            best_delta_v_per_a = delta_v_per_a;
            best_delta_v_session = i;
            best_delta_v_profile = j;
         }
      } else {
      }
   }
}

//If an upgrade is feasible..
if (best_delta_a_session > -1 && best_delta_a >= 0){
   //If there is a positive delta a value, choose that upgrade...
```

```
        free_resources(&sessions[best_delta_a_session].profiles[sessions[best_delta_a_session].active_profile_index],
            resources);
        sessions[best_delta_a_session].active_profile_index = best_delta_a_profile;
        consume_resources(&sessions[best_delta_a_session].profiles[best_delta_a_profile], resources);
        return 1;
    //If an upgrade is feasible..
    } else if(best_delta_v_session > -1 && best_delta_v_per_a >= 0){
        //Otherwise, choose the best delta V
        free_resources(&sessions[best_delta_v_session].profiles[sessions[best_delta_v_session].active_profile_index],
            resources);
        sessions[best_delta_v_session].active_profile_index = best_delta_v_profile;
        consume_resources(&sessions[best_delta_v_session].profiles[best_delta_v_profile], resources);
        return 1;
    //No upgrade is feasible...
    } else {
        return 0;
    }
}


//Does the best non-feasible upgrade to the solution. Returns utility gain. 0 for no upgrade found...
int do_non_feasible_upgrade(SESSION sessions[], int session_count, RESOURCE resources[], int resource_count){

    //Indices for best non_feasible upgrade
    int best_delta_profile = -1;
    int best_delta_session = -1;

    //Variable for Change in utility over scaled change in resource consumption w.r.t. available resource...
    double delta = 0, best_delta = 0, delta_v = 0, best_delta_v = 0;

    int i,j;

    PROFILE *old_profile, *new_profile;

    //Consider each session...
    for (i = 0; i < session_count; i++){
        //Check all possible upgrades from the current profile state for this session...
        old_profile = &sessions[i].profiles[sessions[i].active_profile_index];
        for (j = sessions[i].active_profile_index+1; j < sessions[i].profile_count; j++){
            new_profile = &sessions[i].profiles[j];
            //Find scaled change in resource consumption w.r.t. available resource...
            delta = calculate_scaled_change_in_resource_consumption_available(old_profile, new_profile, resources,
                resource_count);
            //Find change in utility...
            delta_v = old_profile->utility - new_profile->utility;
            //Find change in utilty over this...
            delta = delta_v / delta;

            //Update best delta
            if(delta > best_delta){
                best_delta = delta;
                best_delta_v = -1 * delta_v;
                best_delta_session = i;
                best_delta_profile = j;
            }
        }
    }
```

```
    //If an improvement was found...
    if (best_delta_session > -1 && best_delta_v > 0){
        //If there is a positive delta a value, choose that upgrade...
        free_resources(&sessions[best_delta_session].profiles[sessions[best_delta_session].active_profile_index],
            resources);
        sessions[best_delta_session].active_profile_index = best_delta_profile;
        consume_resources(&sessions[best_delta_session].profiles[best_delta_profile], resources);
        return best_delta_v;
    //If there was no possible improvement...
    } else {
        return 0;
    }

}


//Does the best possible (lowest impact) downgrade to the solution. Returns new aggregate utility gain if
//the resulting state is still not feasible. 0 if no solution resulting in a possitive aggregate gain can
//be found. -1 if a feasible state has been reached with a positive aggregate utility gain.
int do_downgrade(SESSION sessions[], int session_count, RESOURCE resources[], int resource_count, int
    current_solution_gain){

    //Indices for best downgrade
    int best_delta_profile = -1;
    int best_delta_session = -1;

    //Variable for Change in utility over scaled change in resource consumption w.r.t. available resource...
    double delta = 0, delta_v = 0, best_delta_v = 0, best_delta = 0;

    int i,j;

    PROFILE *old_profile, *new_profile;

    //Consider each session...
    for (i = 0; i < session_count; i++){
        //Check all possible downgrades from the current profile state for this session...
        old_profile = &sessions[i].profiles[sessions[i].active_profile_index];
        for (j = sessions[i].active_profile_index - 1; j >= MINIMUM_PROFILE_INDEX; j--){
            new_profile = &sessions[i].profiles[j];
            //Find change in utility...
            delta_v = old_profile->utility - new_profile->utility;
            //If this downgrade would still result in an aggregate utility gain, consider it...
            if(current_solution_gain - delta_v > 0){
                //Find scaled change in resource consumption w.r.t. available resource...
                delta = calculate_scaled_change_in_resource_consumption_overconsumed(old_profile, new_profile, resources,
                    resource_count);
                //Find change in utilty over this...
                delta = delta_v / delta;

                //Update best delta
                if(delta > best_delta){
                    best_delta = delta;
                    best_delta_v = delta_v;
                    best_delta_session = i;
                    best_delta_profile = j;
                }
```

```
            }
        }
    }

    //If an improvement was found...
    if (best_delta_session > -1){
        //If there is a positive delta value, choose that downgrade...
        free_resources(&sessions[best_delta_session].profiles[sessions[best_delta_session].active_profile_index],
            resources);
        sessions[best_delta_session].active_profile_index = best_delta_profile;
        consume_resources(&sessions[best_delta_session].profiles[best_delta_profile], resources);
        if(check_solution_feasibility(resources, resource_count) == FEASIBLE){
            return DOWNGRADE_SOLUTION_FOUND;
        } else {
            //Return adjusted gain...
            return current_solution_gain - best_delta_v;
        }
    //No downgrade that preserved gain...
    } else {
        return DOWNGRADE_SOLUTION_FAILED;
    }

}

//Takes initial solution state and ensures it is feasible, by performing downgrades if necessary.
//Returns NOT_FEASIBLE if no feasible solution can be found, FEASIBLE if succesful.
int find_feasible_solution(SESSION sessions[], int session_count, RESOURCE resources[], int resource_count){

    //Indices for best downgrade
    int best_delta_profile = -1;
    int best_delta_session = -1;

    //Variable for Change in utility over scaled change in resource consumption w.r.t. available resource...
    double delta = 0, delta_v = 0, best_delta = 0;

    int i,j;
    PROFILE *old_profile, *new_profile;

    while(check_solution_feasibility(resources, resource_count) != FEASIBLE){

        //Consider each session...
        for (i = 0; i < session_count; i++){
            //Check all possible downgrade from the current profile state for this session...
            old_profile = &sessions[i].profiles[sessions[i].active_profile_index];
            for (j = sessions[i].active_profile_index - 1; j >= MINIMUM_PROFILE_INDEX; j--){
                new_profile = &sessions[i].profiles[j];
                //Find change in utility...
                delta_v = old_profile->utility - new_profile->utility;
                //Find scaled change in resource consumption w.r.t. available resource...
                delta = calculate_scaled_change_in_resource_consumption_overconsumed(old_profile, new_profile, resources,
                    resource_count);
                //Find change in utilty over this...
                delta = delta_v / delta;

                //Update best delta
                if(best_delta_session == -1 || delta > best_delta){
                    best_delta = delta;
```

```
                best_delta_session = i;
                best_delta_profile = j;
            }
          }
        }
        if (best_delta_session > -1){
          //If there is a downgrade, implement it...
          free_resources(&sessions[best_delta_session].profiles[sessions[best_delta_session].active_profile_index],
              resources);
        sessions[best_delta_session].active_profile_index = best_delta_profile;
          consume_resources(&sessions[best_delta_session].profiles[best_delta_profile], resources);
        } else {
          return NOT_FEASIBLE;
        }
    }
    return FEASIBLE;
}

//Caches the sessions current profile selection.
void cache_session_state(SESSION sessions[], int session_count, int profile_index_cache[]){
    int i;
    for (i = 0; i < session_count; i++){
      profile_index_cache[i] = sessions[i].active_profile_index;
    }
    return;
}

//Revives the sessions current profile selection from the given cache.
void restore_cached_session_state(SESSION sessions[], int session_count, int profile_index_cache[], RESOURCE resources
      []){
    int i;
    for (i = 0; i < session_count; i++){
      free_resources(&sessions[i].profiles[sessions[i].active_profile_index], resources);
      sessions[i].active_profile_index = profile_index_cache[i];
      consume_resources(&sessions[i].profiles[sessions[i].active_profile_index], resources);
    }
    return;
}


//Calculate scaled change in aggregate resource consumption for an upgrade...
//Assumes resource requirements arrays are sorted by index!!!
//Includes '+1 fudge factor' to avoid introducing division by 0.
double calculate_scaled_change_in_resource_consumption_available(PROFILE* old_profile, PROFILE* new_profile, RESOURCE
      resources[], int resource_count){
    int i, j;
    double delta_a = 0;
    //Compare old and new resources...
    //Step through array intelligently until we're off the end of one or the other...
    i = 0; j = 0;
    while(i < old_profile->resource_count && j < new_profile->resource_count){
      if(old_profile->requirements[i].resource_id == new_profile->requirements[j].resource_id){
        delta_a += (((double)(old_profile->requirements[i].requirement - new_profile->requirements[j].requirement)) / (
            resources[old_profile->requirements[i].resource_id].maximum_resource - resources[old_profile->
            requirements[i].resource_id].committed_resource + 1));
        i++; j++;
      } else if(old_profile->requirements[i].resource_id < new_profile->requirements[j].resource_id){
```

```
            delta_a += (((double)(old_profile->requirements[i].requirement)) / (resources[old_profile->requirements[i].
                resource_id].maximum_resource - resources[old_profile->requirements[i].resource_id].committed_resource
                + 1));
            i++;
        } else if(new_profile->requirements[j].resource_id < old_profile->requirements[i].resource_id){
            delta_a -= (((double)(new_profile->requirements[j].requirement)) / (resources[new_profile->requirements[j].
                resource_id].maximum_resource - resources[new_profile->requirements[j].resource_id].committed_resource
                + 1));
            j++;
        }
    }
    //Finish off remaining requirements...
    while(i < old_profile->resource_count || j < new_profile->resource_count){
        if(i < old_profile->resource_count){
            delta_a += (((double)(old_profile->requirements[i].requirement)) / (resources[old_profile->requirements[i].
                resource_id].maximum_resource - resources[old_profile->requirements[i].resource_id].committed_resource
                + 1));
            i++;
        } else if(j < new_profile->resource_count){
            delta_a -= (((double)(new_profile->requirements[j].requirement)) / (resources[new_profile->requirements[j].
                resource_id].maximum_resource - resources[new_profile->requirements[j].resource_id].committed_resource
                + 1));
            j++;
        }
    }
    return delta_a;

}

//Calculate scaled change in aggregate resource consumption for an upgrade (overconsuemd case)...
//Assumes resource requirements arrays are sorted by index!!!
//Includes '-1 fudge factor' to avoid introducing division by 0.
double calculate_scaled_change_in_resource_consumption_overconsumed(PROFILE* old_profile, PROFILE* new_profile, RESOURCE
        resources[], int resource_count){
    int i, j;
    double delta_a = 0;

    //Compare old and new resources...
    //Step through array intelligently until we're off the end of one or the other...
    i = 0; j = 0;
    while(i < old_profile->resource_count && j < new_profile->resource_count){
        if(old_profile->requirements[i].resource_id == new_profile->requirements[j].resource_id){
            delta_a += (((double)(old_profile->requirements[i].requirement - new_profile->requirements[j].requirement)) / (
                resources[old_profile->requirements[i].resource_id].committed_resource - resources[old_profile->
                requirements[i].resource_id].maximum_resource - 1));
            i++; j++;
        } else if(old_profile->requirements[i].resource_id < new_profile->requirements[j].resource_id){
            delta_a += (((double)(old_profile->requirements[i].requirement)) / (resources[old_profile->requirements[i].
                resource_id].committed_resource - resources[old_profile->requirements[i].resource_id].maximum_resource
                - 1));
            i++;
        } else if(new_profile->requirements[j].resource_id < old_profile->requirements[i].resource_id){
            delta_a -= (((double)(new_profile->requirements[j].requirement)) / (resources[new_profile->requirements[j].
                resource_id].committed_resource - resources[new_profile->requirements[j].resource_id].maximum_resource
                - 1));
            j++;
        }
```

```
    }
    //Finish off remaining requirements...
    while(i < old_profile->resource_count || j < new_profile->resource_count){
      if(i < old_profile->resource_count){
        delta_a += (((double)(old_profile->requirements[i].requirement)) / (resources[old_profile->requirements[i].
            resource_id].committed_resource - resources[old_profile->requirements[i].resource_id].maximum_resource
            - 1));
        i++;
      } else if(j < new_profile->resource_count){
        delta_a -= (((double)(new_profile->requirements[j].requirement)) / (resources[new_profile->requirements[j].
            resource_id].committed_resource - resources[new_profile->requirements[j].resource_id].maximum_resource
            - 1));
        j++;
      }
    }
    return delta_a;
}

//Calculate change in resource usage for a profile...
//Assumes resource requirements arrays are sorted by index!!!
double calculate_change_in_aggregate_resource_consumption(PROFILE* old_profile, PROFILE* new_profile, RESOURCE resources
    [], int resource_count){

    int i, j;
    double delta_a = 0;

    //Compare old and new resources...
    //Step through array intelligently until we're off the end of one or the other...
    i = 0; j = 0;
    while(i < old_profile->resource_count && j < new_profile->resource_count){
      if(old_profile->requirements[i].resource_id == new_profile->requirements[j].resource_id){
        delta_a += ((double)(old_profile->requirements[i].requirement - new_profile->requirements[j].requirement)) *
            resources[old_profile->requirements[i].resource_id].committed_resource;
        i++; j++;
      } else if(old_profile->requirements[i].resource_id < new_profile->requirements[j].resource_id){
        delta_a += ((double)(old_profile->requirements[i].requirement)) * resources[old_profile->requirements[i].
            resource_id].committed_resource;
        i++;
      } else if(new_profile->requirements[j].resource_id < old_profile->requirements[i].resource_id){
        delta_a -= ((double)(new_profile->requirements[j].requirement)) * resources[new_profile->requirements[j].
            resource_id].committed_resource;
        j++;
      }
    }
    //Finish off remaining requirements...
    while(i < old_profile->resource_count || j < new_profile->resource_count){
      if(i < old_profile->resource_count){
        delta_a += ((double)(old_profile->requirements[i].requirement)) * resources[old_profile->requirements[i].
            resource_id].committed_resource;
        i++;
      } else if(j < new_profile->resource_count){
        delta_a -= ((double)(new_profile->requirements[j].requirement)) * resources[new_profile->requirements[j].
            resource_id].committed_resource;
        j++;
      }
    }
```

```
    return delta_a;

}

//Consumes resources for the given profile
void consume_resources(PROFILE* profile , RESOURCE resources[]){
    int i;
    for (i = 0; i < profile->resource_count; i++){
        resources[profile->requirements[i].resource_id].committed_resource += profile->requirements[i].requirement;
    }
    return;
}

//Frees resource from the given profile
void free_resources(PROFILE* profile , RESOURCE resources[]){
    int i;
    for (i = 0; i < profile->resource_count; i++){
        resources[profile->requirements[i].resource_id].committed_resource -= profile->requirements[i].requirement;
    }
    return;
}


//Checks if resources are available to accomodate the indicated profile
//Assumes resource requirements arrays are sorted and id corresponds to index in resource array.
int check_upgrade_feasibility(PROFILE* old_profile , PROFILE* new_profile , RESOURCE resources[]){
    int i, j, current_resource_id;
    int feasible = FEASIBLE;

    i = 0;
    for (j = 0; j < new_profile->resource_count; j++){
        current_resource_id = new_profile->requirements[j].resource_id;
        while(i < old_profile->resource_count && old_profile->requirements[i].resource_id < current_resource_id){
            i++;
        }
        if(i < old_profile->resource_count && old_profile->requirements[i].resource_id == current_resource_id){
            if(resources[current_resource_id].committed_resource - old_profile->requirements[i].requirement + new_profile->
                requirements[j].requirement > resources[current_resource_id].maximum_resource){
                feasible = NOT_FEASIBLE;
                break;
            }
        } else if(resources[current_resource_id].committed_resource + new_profile->requirements[j].requirement > resources
            [current_resource_id].maximum_resource){
            feasible = NOT_FEASIBLE;
            break;
        }
    }

    return feasible;
}


//Check if given resource consumption is feasible. Returns NOT_FEASIBLE if any overcommitted resource found, FEASIBLE
    otherwise.
int check_solution_feasibility(RESOURCE resources[], int resource_count){
    int i;
    for (i = 0; i < resource_count; i++){
```

```
        if ( resources[i].committed_resource > resources[i].maximum_resource) return NOT_FEASIBLE;
    }
    return FEASIBLE;
}
```