Dynamic Bandwidth Allocation for Requests with Classes and Priorities in Preemptive Distributed Networks

Amit D. Naik  
Purdue University School of Electrical and Computer Engineering

Howard Jay Siegel  
Purdue University School of Electrical and Computer Engineering

Edwin K. P. Chong  
Purdue University School of Electrical and Computer Engineering

Follow this and additional works at: http://docs.lib.purdue.edu/ecetr

http://docs.lib.purdue.edu/ecetr/26

This document has been made available through Purdue e-Pubs, a service of the Purdue University Libraries. Please contact epubs@purdue.edu for additional information.
DYNAMIC BANDWIDTH ALLOCATION FOR REQUESTS WITH CLASSES AND PRIORITIES IN PREEMPTIVE DISTRIBUTED NETWORKS

AMIT D. NAIK
HOWARD JAY SIEGEL
EDWIN K. P. CHONG

TR-ECE 00-10
JULY 2000

SCHOOL OF ELECTRICAL AND COMPUTER ENGINEERING
PURDUE UNIVERSITY
WEST LAFAYETTE, INDIANA 47907-1285
Dynamic Bandwidth Allocation for Requests with Classes and Priorities in Preemptive Distributed Networks

Amit D. Naik
Howard Jay Siegel
Edwin K. P. Chong

Purdue University
School of Electrical and Computer Engineering
1285 Electrical Engineering Building
West Lafayette, IN 47907-1285 USA
(naika, hj, echong)@ecn.purdue.edu

July 2000
ACKNOWLEDGMENTS

We thank Professor A. A. Maciejewski for his insightful comments and suggestions regarding this research. We are grateful to Shoukat Ali, Jong-Kook Kim, and Pranav Dharwadkar, who read the manuscript carefully and offered a number of useful comments and suggestions.

This research was supported by the DARPA/ISO BADD Program and the Office of Naval Research under ONR grant number N00014-97-1-0804, and by the DARPNITO AICE program under contract numbers DABT63-99-C-0010 and DABT63-99-C-0012. Intel and Microsoft donated some of the equipment used in this research.
# Table of Contents

LIST OF TABLES ........................................................................................................... v
LIST OF FIGURES ........................................................................................................ vi
ABSTRACT .................................................................................................................. x

1. INTRODUCTION ........................................................................................................ 1
   1.1. Overview .............................................................................................................. 1
   1.2. Motivation ............................................................................................................ 2
   1.3. Report Structure .................................................................................................. 5

2. REQUEST MODEL .................................................................................................... 7
   2.1. Overview .............................................................................................................. 7
   2.2. Paradigms for Grouping Requests ........................................................................ 7
   2.3. Notion of Class and Priority ............................................................................... 9
   2.4. Request Types ..................................................................................................... 12

3. NETWORK TOPOLOGY AND HEURISTIC OVERVIEW ........................................ 15
   3.1. Network Topology .............................................................................................. 15
   3.2. Overview of Heuristics Developed .................................................................... 17
   3.3. Computational Complexity of the Scheduling Problem .................................... 19

4. RELATED WORK ....................................................................................................... 23

5. PROPOSED HEURISTIC .......................................................................................... 31
   5.1. Overview ............................................................................................................. 31
   5.2. Scheduling Heuristic ......................................................................................... 31
   5.3. Session Type Request Scheduling .................................................................... 32
   5.4. Data Type Request Scheduling ......................................................................... 39

6. PERFORMANCE COMPARISONS ............................................................................ 43
   6.1. Overview ............................................................................................................. 43
   6.2. Simple Scheduling Technique .......................................................................... 44
6.3. Upper Bounds

6.3.1. Loose upper bound

6.3.2. Ingress upper bound

6.3.3. Egress upper bound

7. SIMULATION EXPERIMENTS

7.1. Overview

7.2. Request Generation Details

7.3. Request Size and Duration

7.4. Priority Weights

7.5. Probability of Preemption

7.6. Versions of the Heuristic Simulated

8. RESULTS OF THE SIMULATION EXPERIMENTS

8.1. Overview

8.2. Simulation Results for the Eight Scenarios

8.3. Priority Level and Number of Preemption Related Results

8.4. Execution Times

8.5. Summary

9. CONCLUSIONS AND FUTURE WORK

LIST OF REFERENCES

APPENDIX 1: PSEUDO CODE FOR FULL HEURISTIC

APPENDIX 2: PSEUDO CODE FOR SORT + PREEMPT

APPENDIX 3: PSEUDO CODE FOR SORT
LIST OF TABLES

Table | Page |
--- | --- |
1 | Parameter details chosen for simulation experiments | 54 |
2 | Values of parameters varied to generate eight different simulation scenarios. For a given value of one of the parameters, the other two are varied to generate $2^3 = 8$ simulation scenarios | 57 |
3 | Average number of preemptions by class for the sort + preempt version and the **full** versions of the heuristic for different values of the loading factor (l.f), $a$, and preemption probability (p.p.), each averaged over 20 simulation runs | 76 |
4 | Example of a resource–status–unified table depicting the minimum of the unused bandwidths available at each point in time at the ingress and egress links of request R. The order values are based on duration values | 93 |
LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Topology of the underlying communication network.</td>
<td>15</td>
</tr>
<tr>
<td>2</td>
<td>Situation when the request ( r ) is being blocked by requests of class ( c ). The bandwidth occupied by requests of class ( &gt; c ) is not shown.</td>
<td>35</td>
</tr>
<tr>
<td>3</td>
<td>Situation where request 1 is needlessly preempted by request 2, as request 2 ends up getting preempted by request 3, where ( k &lt; i ).</td>
<td>36</td>
</tr>
<tr>
<td>4</td>
<td>The confidence intervals for loading factor = 1.2, ( \omega = 10 ), and preemption probability = 0.9 with the <strong>full</strong> heuristic (sorting + preemption + reposition).</td>
<td>60</td>
</tr>
<tr>
<td>5</td>
<td>The relative performance for loading factor = 0.7, ( \omega = 2 ), and preemption probability = 1.0 for complete sharing policy, upper bounds, and the three variations of the heuristic averaged over 20 experiments. Note: <strong>c.s.</strong> = complete sharing performance, ingress = ingress upper bound, egress = egress upper bound and C1 = class 1, C2 = class 2, C3 = class 3. The loose upper bound for the simulation experiment averaged over 20 experiments is 7633.</td>
<td>62</td>
</tr>
<tr>
<td>6</td>
<td>The relative performance for loading factor = 0.7, ( \omega = 2 ), and preemption probability = 0.9 for complete sharing policy, upper bounds, and the three variations of the heuristic averaged over 20 experiments. Note: <strong>c.s.</strong> = complete sharing performance, ingress =</td>
<td></td>
</tr>
</tbody>
</table>
ingress upper bound, egress = egress upper bound and C1 = class 1, C2 = class 2, C3 = class 3. The loose upper bound for the simulation experiment averaged over 20 experiments is 7603. 

The relative performance for loading factor = 0.7, \( \omega = 10 \), and preemption probability = 1.0 for complete sharing policy, upper bounds, and the three variations of the heuristic averaged over 20 experiments. Note: c.s. = complete sharing performance, ingress = ingress upper bound, egress = egress upper bound and C1 = class 1, C2 = class 2, C3 = class 3. The loose upper bound for the simulation experiment averaged over 20 experiments is 583277.

The relative performance for loading factor = 0.7, \( \omega = 10 \), and preemption probability = 0.9 for complete sharing policy, upper bounds, and the three variations of the heuristic averaged over 20 experiments. Note: c.s. = complete sharing performance, ingress = ingress upper bound, egress = egress upper bound and C1 = class 1, C2 = class 2, C3 = class 3. The loose upper bound averaged over 20 experiments is 576222.

The relative performance for loading factor = 1.2, \( \omega = 2 \), and preemption probability = 1.0 for complete sharing policy, upper bounds, and the three variations of the heuristic averaged over 20 experiments. Note: c.s. = complete sharing performance, ingress = ingress upper bound, egress = egress upper bound and C1 = class 1, C2 = class 2, C3 = class 3. The loose upper bound averaged over 20 experiments is 7610.

The relative performance for loading factor = 1.2, \( \omega = 2 \), and preemption probability = 0.9 for complete sharing policy, upper
bounds, and the three variations of the heuristic averaged over 20 experiments. Note: c.s. = complete sharing performance, ingress = ingress upper bound, egress = egress upper bound and C1 = class 1, C2 = class 2, C3 = class 3. The loose upper bound averaged over 20 experiments is 7589.

11 The relative performance for loading factor = 1.2, $\omega = 10$, and preemption probability = 1.0 for complete sharing policy, upper bounds, and the three variations of the heuristic averaged over 20 experiments. Note: c.s. = complete sharing performance, ingress = ingress upper bound, egress = egress upper bound and C1 = class 1, C2 = class 2, C3 = class 3. The loose upper bound averaged over 20 experiments is 569298.

12 The relative performance for loading factor = 1.2, $\omega = 10$, and preemption probability = 0.9 for complete sharing policy, upper bounds, and the three variations of the heuristic averaged over 20 experiments. Note: c.s. = complete sharing performance, ingress = ingress upper bound, egress = egress upper bound and C1 = class 1, C2 = class 2, C3 = class 3. The loose upper bound averaged over 20 experiments was 578567.

13 The number of class 1 requests of each priority level (1, 2, 3, and 4). This is for the scenario with loading factor = 1.2, and preemption probability = 0.9 and for $\omega = 10$ and $\omega = 2$. Each number is averaged over 20 simulation experiments.

14 The number of class 1 requests of each priority level (1, 2, 3, and 4). This is for the scenario with loading factor = 1.2, and preemption
probability = 1.0 and for $\omega = 10$ and $\omega = 2$. Each number is averaged over 20 simulation experiments.

15 The number of class 1 requests of each priority level (1, 2, 3, and 4). This is for the scenario with loading factor = 0.7, and preemption probability = 0.9 and for $\omega = 10$ and $\omega = 2$. Each number is averaged over 20 simulation experiments.

16 The number of class 1 requests of each priority level (1, 2, 3, and 4). This is for the scenario with loading factor = 0.7, and preemption probability = 1.0 and for $\omega = 10$ and $\omega = 2$. Each number is averaged over 20 simulation experiments.
ABSTRACT

In today's general-purpose networked computing environments, both commercial and military, in addition to providing connectivity, providing quality of service (QoS) to users is a major concern. One of the major QoS parameters in this context is bandwidth. This research focuses mainly on the intelligent allocation of bandwidth to requests in oversubscribed networks such that, some measure of worth associated with the satisfied requests is optimized. In this work, heterogeneous networks with preemptive capabilities have been considered. A preemptive network is capable of interrupting a request that is currently in progress to service a request that is considered to be more important to the system by some metric. To provide a sound theoretical foundation for grouping requests into different categories, a class and priority based mechanism is described such that, all requests belonging to a higher class are to be satisfied before any request(s) of a lower class are considered. Within a given class, further differentiation among requests is achieved by assigning different priority levels to the requests depending on their relative importance.

Scheduling heuristics have been developed to determine the bandwidth allocations to the requests depending on their class, relative worths of their priority levels, and the network capacities available for that period. These heuristics allocate network bandwidth to requests of two types: 1) session type – requesting a fixed amount of bandwidth for a given interval of time, and 2) data type – requesting a data item of given size with an earliest available time and fixed deadline. The heuristics used for the intelligent allocation of bandwidth try to maximize the weighted priority (or worth) of the satisfied requests in the most important class, then the second most important class, and so on, down to the least important class. Simulation experiments have been conducted to quantify the performance of the scheduling algorithms in different scenarios.
1. INTRODUCTION

1.1. Overview

As the field of computer networking has begun to mature, there has been a gradual trend to look at not just providing a means to connect different computer systems together, but to try to provide this service bounded by certain performance assurances. Thus, currently in many distributed and networked computing environments, both commercial as well as military, the emphasis of research activities has been shifting from providing just interconnectivity between various systems to providing a means to deliver Quality of Service (QoS) for these environments. In this work, the term QoS is used to signify the collective measure of the level of service delivered to the customer. Several basic performance parameters can be used to characterize QoS, including bandwidth, availability (low downtime), throughput, latency, jitter, and loss.

The critical problem in providing the desired end-to-end QoS to customers is the successful creation and maintenance of the technical infrastructure (hardware and software) involved in providing such performance bounds in systems that are as of now mostly best effort. The work presented here is primarily targeted at developing algorithms that schedule requests in a heterogeneous networked environment where the QoS is significant. A simulation study is provided as a basis for quantifying the performance of the algorithms presented and to aid in further work in this field by clearly outlining the simplifying assumptions and performance criteria used.

This research evolved primarily out of work done for the Agile Information Control Environment (AICE) project [AIC98] sponsored by DARPA/ISO. The main goal of the AICE program was the development of technologies required to manage and control
information flows in support of military operations. The AICE architecture consists of four main functional layers: an Information Policy Management (IPM) layer, an Adaptive Information Control (AIC) layer, a MetaNet layer, and the physical networks layer. The main purpose of the MetaNet layer is to interact with the various heterogeneous physical networks present in the network layer so that the higher layers, AIC and the IPM, would only have to deal with the aggregated abstraction of these networks provided to them by the MetaNet. Thus, the AIC and IPM layers are provided end-to-end QoS decision-making capabilities by the MetaNet layer. The AIC layer makes end-to-end QoS based resource allocations to the requests in the system, based on the information provided to it by the MetaNet to achieve a high "global utility" as defined by the IPM. A more detailed description of the AIC and MetaNet layers can be found in [WaJ99]. This research grew out of an effort to develop efficient resource allocation algorithms for the AIC layer to allocate the end-to-end resources in an AICE-like scenario. In this work, rather than dealing with an aggregated abstraction of the underlying networks provided by the MetaNet, a tighter integration with the underlying network is assumed. Consequently, the AICE-like environment, in which this research was conducted, is assumed to have detailed information regarding capacities at various points in the underlying network.

1.2. Motivation

As noted earlier, the need for providing QoS-aware services to users is increasing day by day. In spite of the fact that more and more network bandwidth is becoming available, due to the massive proliferation of the number of users connected to the Internet and the consequent explosion in the variety of applications needed and supported by these users, resources such as bandwidth continue to be a major bottleneck in the development of today's networks. Consequently, any network where better than best-effort service is desired will have to implement some sort of resource delivery guarantees in such an oversubscribed scenario. This has led to the development of services like the guaranteed service model [BrC94] and differentiated service model
that try to provide some guarantees regarding the bandwidth allocated and delay to the user.

One extreme approach to achieving such guarantees would be to allocate a large amount of bandwidth to each user. Such an approach may, however, lead to low network capacity utilization and may end up costing the user much more than necessary, because pricing may depend not on actual bandwidth consumption but just allocated bandwidth. So both the network service provider (NSP) and the customers need a better approach towards the problem of bandwidth allocation. In such an environment, the intelligent allocation of bandwidth in the network is critical for the NSP to maintain profitability as well as for the customers to get the maximal benefit for their price.

In addition to providing a good utilization of the network, profitability to the NSP, and maximal benefit to the customer, the bandwidth allocation strategy should also ensure fairness. The need for fairness translates into the requirement that a higher paying customer should always get better grade of service (as defined by some parameters) than some other lesser paying customer. At the same time, no single customer should be able to take over the entire network. In a military context, price issues usually correspond to military priorities.

Complicating the development of an intelligent bandwidth allocation policy is the large variety of applications that use the network simultaneously. These applications have very different requirements in terms of bandwidth, duration, delay; etc. The notion of QoS for an application translates to acceptable application-level performance. Hence, with the variety of applications using the network, it is very difficult to accurately estimate the network capacity needed to maintain acceptable application-level QoS, as this depends both on the user-specified performance requirements and the robustness of the application to bandwidth changes and outages.

This research is focussed on developing bandwidth allocation strategies (i.e., bandwidth provisioning algorithms) in networks that attempt to provide some performance guarantees to its users. Typically in networks of even moderate sizes, the communication volumes are much larger than the buffering capabilities in the networks. This implies that decisions regarding current requests have to be made without the
knowledge of future request arrivals. The problem becomes more acute when the available bandwidth is insufficient to satisfy all requests and some have to be rejected due to contention at one or more points in the network. In this context, it is logical to consider "softening" the rigidity of the guaranteed bandwidth policy to allow preemption of requests in progress, to accommodate calls that are "worth more" to the system by some metric.

There are many scenarios in which it is acceptable for a connection in progress to be preempted or aborted. One such example is using some pricing scheme based approach where, the customers are given price discounts proportional to the probability of preemption by other users or by applications of the same user. Another example is in real-time military systems, where a high priority real-time connection is allowed to preempt a low priority non-real-time connection. Indeed, preemption has obvious disadvantages in that the customers may not be willing to tolerate preemptions and also that preempting a connection in progress implies that all the bandwidth utilized by that connection before preemption may be wasted and in some cases, may require retransmission of data. However, the improvement in the overall network performance and the ability to correct past allocation mistakes due to insufficient information, makes preemptive networks an attractive proposition.

This research is consequently targeted towards such networks that have the ability to preempt (i.e., abort) connections already scheduled, to schedule more valuable ones. The phrase "preemptive network" is used throughout this work to imply a network that has the capability of preempting or aborting connections already in progress. We have designed on-line heuristics that perform bandwidth allocation in preemptive networks in such a manner as to attempt to maximize the worth of the requests satisfied. Two types of communication requests are considered: sessions (where a given bandwidth is requested for a given interval of time) and data (where a data item available at a source at a given time must arrive at a destination by a given time). Each request will have an associated priority. A mechanism for grouping requests that is flexible enough to be applicable in a wide range of scenarios has also been proposed. The heuristics developed also include the preemption related rules in such scenarios. It is hoped that this work will
be of use to network service providers in the areas of bandwidth allocation in oversubscribed and preemptive networks where service guarantees are required.

As this research attempts to control the provisioning of an oversubscribed resource (bandwidth) to multiple competing requests, it is clear that in most cases only a subset of these requests will be satisfied. Viewed from this standpoint, this research may also be generically grouped under the category of admission control. However, it needs to be emphasized that in some cases, admission control is defined to mean non-preemptive bandwidth allocation in an on-line scenario, i.e., when there is no knowledge of future request arrivals. This may be too restrictive a definition and a slightly broader approach is needed. Another related category, is that of scheduling heuristics. Depending on the type of the request being processed, the heuristics developed here also implement some elements of scheduling techniques to maximize the net worth obtained.

A final point that needs to be clarified is that, in spite of the fact that this research was carried out in the background of a military environment (AICE), its applicability is in no way restricted only to military systems. All of the important concepts and heuristics developed herein are applicable to both commercial and military settings. The mechanism for grouping the requests proposed here is also flexible enough to be applicable in almost any networked environment that satisfies only a few very modest rules.

1.3. Report Structure

Sections 1 through 4 serve to introduce the research area, and provide background material regarding its scope and applicability. Section 2 specifies the request model, while Section 3 explains the network model and gives an overview of the heuristic techniques developed. A literature survey that explores the related research in this field by other groups is covered in Section 4 and helps to compare and contrast the work presented in this report. Sections 5 and 6 present the details of the heuristics proposed and the bounds that are used to compare the performance of these proposed heuristics, respectively. The simulation experiments that were used to study one implementation of the heuristics proposed are described in Section 7 and the results, that justify the use of
these heuristics, are presented in Section 8. The final chapter, Section 9, summarizes the significance of this research and outlines some possible future work in this area.
2. REQUEST MODEL

2.1. Overview

To better explain the research problem addressed by this work and to aid in any comparisons with the work done by other groups in similar fields, a clearer understanding of some of the terminology involved is helpful. In this work a request is used to refer to a communication need between a given source and one or more destinations. So, a request could be a voice call in a cellular network, a video conferencing session involving multiple participants, an HTML document being requested of a web server, a file transfer over a local area network, etc. The crux of this work involves scheduling these communications requests such that a certain performance criterion is to be optimized, while at the same time, obeying certain predefined policies.

2.2. Paradigms for Grouping Requests

In most communication networks, inherent to each request or call is the idea of the worth associated with that call. (In the remainder of this chapter, the terms call, connection, and request are used to convey the same basic idea.) The concept of a given request being more important or more valuable in some sense as compared to another request, arises naturally in most communication systems. A call from a main office might be more important as compared to a call from a remote branch office. As another example, in cellular systems that employ call-handoff procedures, a call already in progress in a local cell might be of more importance than a handoff call from a remote cell which in turn may be more important than a new call arising in the local cell. In commercial systems, the notion of worth associated with a given request is typically
related to the revenue that will be generated by satisfying the request. Thus, a call that lasts longer might be worth more than a shorter call; or a request that utilizes less network resources but at peak usage times might be worth more than a request that utilizes more resources at an off-peak period.

The notion of worth can also be illustrated in a military setting where similar geographic factors (location in a war zone) or identity of the source and/or destination (call from a general to a battlefield commander) or some other characteristic associated with the request serves to introduce the notion of relative worth of requests. Typically there are three or four value levels (real-time, high, medium, and low) associated with a request in a usual military scenario. In the related literature dealing with the notion of worths in a communication network, the term most commonly used is priority [Pey94].

As this research also considers QoS parameters while providing the schedules for the requests, it is instructive to consider another differentiating aspect while handling the requests. The QoS parameters such as latency, jitter, and loss characteristics might be different for different requests. Consequently, a network that seeks to satisfy these requests in a QoS-aware manner needs some way of differentiating between these requests and, the term most commonly used in this context is request class [MuS98] or class of service [RaQ97] provided to a request. A simple example of actual implementation-level significance of a class of service is the implementation of multiple queues in routers, with packets belonging to requests of different classes being directed to different queues and receiving different handling (treatment) by the routers. A typical example of this may be a digital cellular network that deals with both voice and data type calls. The voice-call packets may need more stringent limits on their latency and jitter parameters while these parameters may not be of much importance to data calls (within reasonable limits). So it would make sense for the network architect to provide different classes of service to voice and data type calls or, put another way, group voice and data type requests into different classes.

Thus, the above two different paradigms for differentiating a set of requests are considered in the literature. For the purposes of this research, an underlying mechanism
of grouping requests is used that borrows from both the above concepts, i.e., relative worths (priority) and differentiated treatment (class).

**2.3. Notion of Class and Priority**

In this work, a request has both a class and a priority. Both factors, the class of a request and its associated priority within that class, play a role in the scheduling of that request as will be explained more in detail in later sections. Class allows differentiated treatment of requests, while priorities allow relative worths to be assigned to requests within a class. As noted earlier, most communication systems have three or four priority values (real-time, high medium, and low) associated with the requests in the system. Each of these may be termed as a priority level associated with the request. In this research four priority levels will be considered, where priority level i represents a more important request than priority level j, if i < j, and 1 < i, j < 4.

Though it is usually possible to determine that a given request is worth more than some other request, often it might not be equally simple to quantify this. Application users and system builders often assign an arbitrary naming or numbering scheme to priority levels that does not meaningfully quantify the relative importance of one priority level to another. Such a scheme, therefore, cannot be used alone in the measure. A more meaningful weight must instead be assigned to each priority level so that the relative importance of requests from different priority levels is properly quantified.

Consider the military situation outlined earlier. In peace time, for example, a high priority request might be worth ten medium priority requests, but in war mode one high priority request might be worth 100 medium priority requests. Thus, the utility or worth of a request is calculated as a weighted priority where the weight associated with a priority level is determined by the situational mode (e.g., war time or peace time).

In a commercial setting, the concept of a weight associated with a given priority might be explained in the following manner. Consider a network service provider (NSP) providing round the clock service to customers; during off-peak periods, the service provider might consider a high priority request to be only worth ten medium priority ones but during peak periods when network bandwidth is much harder to obtain, a high
priority request may be worth a 100 medium priority ones. The NSP might choose to pass on these costs to its customers in the form of tariff plans that explain to the customer the relative benefit of utilizing bandwidth at off-peak time.

The weight of a priority level \( i \) request is given by \( \omega^{i-1} \) where \( \omega \) is a weighting constant such that its value could be 2 or 10. The value of \( \omega \) could depend on the situational mode, e.g., war time or peace time. This type of weighting scheme was used in [ThB00, ThS00].

The class hierarchy in processing and satisfying requests is defined so that any request that belongs to a more important class will be satisfied (or attempted to be satisfied) before requests belonging to a less important class are processed. One advantage of this scheme is that, especially in a military context, there may be requests that are of an extremely critical nature that need to be satisfied irrespective of how many other less important messages need to be dropped. This implies that such requests effectively have a priority weighting of almost infinity. The class-based hierarchy is perfect for such scenarios as it enables us to place such requests in the most important class and helps avoid dealing with unreasonably large (or small) priority values. Thus, the concept of a class captures the notion of one very critical request being worth more than any number of less critical ones. In this research, three classes are considered, where class \( i \) is more important than class \( j \), if \( i < j \), and \( 1 < i, j < 3 \).

In a commercial system, both the customer and the NSP could have a role in determining what class and priority within that class a particular request should have in the system. To the customer, these concepts (class and priority) would ultimately have an impact on the tariff to be paid to the NSP. This leads naturally to the idea of incorporating pricing in network communication systems. Over the past few years, the concept of incorporating pricing into networking decisions has been gaining interest. Some of these studies (e.g., [CoS93]) have concentrated on best-effort networks like the Internet while others (e.g., [LoV93], [Ke194], [MiM96]) focus on ATM-based networks. The chief contribution of such work is however in the area where all requests receive the same type of service, and innovative economics-based mechanisms like bidding are used.
to maximize the network revenue. There has also been some work in this area where multiple request classes have been considered for pricing, as in [SaF95] and [OrS97].

With the increasing need for providing QoS to requests, the need to provide pricing incentives to the NSPs to develop communication networks capable of QoS-aware resource provisioning is also increasing. In such systems, the price to be paid by the customer for a particular class of service would be an important parameter determining the profitability of the NSP. Recently there has been an enormous increase in interest in research related to this area and some fairly complex routines have been proposed to achieve this objective (e.g., [MaT99]).

Some of the basic ideas incorporated in these aforementioned works are that, if the principal objective of the customer is to obtain maximum benefit for the price paid and that of the NSP is to obtain maximum revenue for its services, various mechanisms exist to provide optimal or near optimal schemes of assigning service classes and priorities to each request in a given system. As clarified earlier, both the customer and the NSP have a role in determining these parameters.

Much of this discussion also holds for military networks with the slight modification that instead of real revenue (in terms of financial gain) the military perspective would be to consider some virtual revenue (in terms of advancement of some overall military objective). Also in the military paradigm, security considerations may play a much greater role in all decisions as compared to commercial systems. So for example, the entity generating the request (e.g., a war-fighter, a battlefield commander, some application running at a military data processing center) may have much less of a say in determining the priority assigned to the request to minimize the effect of malicious users flooding the system with large numbers of high-priority messages.

As stated earlier, the weight associated with a particular priority level in a given class is generally a function of factors such as the situational mode or temporal conditions (e.g., peak period or off-peak period). Hence, the weight to be assigned to a given priority is more or less outside the control of the customer in a network. It is generally determined by the NSP or by the network designers to reflect the relative weightings of
various priority levels depending on various circumstances. In a military situation, the relative priority weightings may reflect the policies.

The concept of classes and priorities used in this research is very flexible, and can be and can be adapted to group requests in almost any communication setting, commercial or military. The actual values for (i) the number of classes, (ii) number of priority levels, (iii) weightings of priority levels, (iv) the class and priority assigned to a request are a function of the given communication setting. The way these values are determined in a real system is outside the scope of this research. For the research here, the values for (i) to (iii) have been specified in this subsection. Requests are assigned a class and a priority based on a uniform distribution in the simulation studies conducted (as described in Section 7).

Given such a structure, the goal of this research is to propose scheduling heuristics that maximize the sum of the weighted priorities of the satisfied requests in the most important class, followed by the next most important class, and so on, down to the least important class. A similar performance measure involving requests in only one class is used in [ThT00]. When comparing two scheduling heuristics, the one that results in the higher sum of weights of the satisfied class 1 requests is considered to be the better one. Only when these class 1 sums are equal, are the class 2 sums examined. Similarly, if both the class 1 sums are equal and the class 2 sums are equal, then class 3 sums are evaluated.

2.4. Request Types

For the purposes of this work, requests are considered to be of two fundamental types: session type requests and data type requests. Session type requests are for a specified amount of bandwidth needed for a given duration of time. In this work, it is assumed that the start and the end times of a session type request are known at the time the requests are made. This assumption is based on the fact that depending on the type of user making the request, mechanisms such as profiling and prior agreements can be utilized to get an expected completion time for such requests. To keep the research
problem of manageable size and to focus on the important aspects of the problem, the assumption is made that each request has only one source and one destination. Thus, a session type request is a request for bandwidth between a given source and a given destination for a given duration. Each such request is in the form of the tuple:

\[
\langle \text{request id}, \text{source}, \text{destination}, \text{start time}, \text{finish time}, \text{bandwidth needed}, \text{priority}, \text{class} \rangle.
\]

Data type requests are for transmitting a specific data item resident on some server (i.e., source) that is a part of the network to some destination. Data type requests have an earliest time, which may correspond to the time that the data may become available at the source, latest time or deadline by which the data has to be present at the destination, and the maximum bandwidth at which the data can be sent. Each data type request also consists of a unique data item identifier that identifies the exact piece of data to be fetched. Each such request is in the form of the tuple:

\[
\langle \text{request id}, \text{source}, \text{destination}, \text{time available}, \text{time latest}, \text{data item needed}, \text{maximum bandwidth}, \text{priority}, \text{class} \rangle.
\]

Again, as in the case of the session type requests, the assumption is made that there is only one destination per request. As the bulk of this research focuses on heuristics that attempt to maximize the worth by intelligent scheduling of bandwidth, the data type requests need a further slight simplification. In dealing with data type requests in the simulations performed in this work, typically, the data item requested is replaced by the corresponding data size of that item. Though this may appear to be a trivial change, depending on the type of system being considered, even this step may involve considerable processing. The important idea to appreciate in this context is that the "data item needed" identifier in the tuple above is actually a form of metadata i.e., information about or descriptive of some other information (the data item in this case).

Typically, in large distributed multimedia databases, the data items of interest may be stored in an unstructured manner. Some examples of such storage are video files containing a sequence of events stored in a video database, large geographical databases containing map images, musical archives containing musical recordings of various artists, medical imaging databases. In such cases, the visualization, manipulation and
transfer of such items may not be a trivial task. Consequently, there is considerable research (e.g., [KeP96], [Lah96], [HuL97], and [Fau99]) in this area. Some of this research suggests new conceptual data modeling and handling techniques to better aid in the identification and transmission of data (particularly image and video).

For systems where it is known a priori that a considerable amount of such unstructured multimedia data is to be transferred over the network, the research mentioned above may be a valuable guide in implementing the request and translation mechanisms in an efficient and timely manner. In this research, it is assumed that such a translation, if required, is already in place and that the scheduler deals with only the relevant information, i.e., the size of the data item that needs to be transferred. Consequently, in data type requests, the "data item needed" is replaced by "data size needed" resulting in:

\[
\langle \text{request id, source, destination, time available, time latest, data size needed, maximum bandwidth, priority, class} \rangle,
\]
3. NETWORK TOPOLOGY AND HEURISTIC OVERVIEW

3.1. Network Topology

As shown in Figure 1, each node in the network is associated with one or more applications that generate the communication requests, both of session and data type. Each node has a network ingress link and a network egress link connecting it to the high-speed backbone network for sending and receiving network traffic respectively. The node serves as the network access point for all the applications associated with that node.

Figure 1: Topology of the underlying communication network.
All the requests arriving at these nodes are processed by the admission control heuristic that decides as to which of these requests to admit into the network.

The admission control algorithm operates under the constraint that the sum of the bandwidths of all allocated requests at a given node should not exceed the link capacities available at that node. Once a request has been admitted into the network, the source node for that request allocates a fixed bandwidth to that request on its network ingress link and sends it in to the network cloud (i.e., the high-capacity network backbone). This backbone then routes the request to the appropriate destination node, which has also allocated bandwidth for this request on its network egress link.

The network model is similar to the one proposed in [BrC94] and borrows from the current structuring of the Internet in which the nodes can be assumed to be points-of-presence (PoP) of the major Internet Service Providers (ISP). These ISPs have high bandwidth connections into the very high capacity backbone of the Internet and are able to provide Internet services to many different consumers for many different kinds of applications.

The resource allocation algorithm is assumed to have detailed link level information regarding the capacities of the network ingress and egress links associated with each node. Whenever a request completes transmission or gets preempted because some other more important request needs to be scheduled, the bandwidth allocated to it on the source ingress link and on the destination egress link is freed and becomes available for other requests contending for that resource. The overhead involved in preempting a network request is assumed to be negligible in terms of the network bandwidth utilized and additional delay introduced.

To reduce the level of complexity involved in modeling the network, a simplifying assumption is made here that the network backbone has sufficient bandwidth capacity and speed so that it can satisfy all requests entering into it simultaneously from the network ingress links of all nodes (i.e., appropriately route requests with negligible increase in latency and nearly zero blocking probability). Thus, in effect, the network backbone acts like a very high-capacity switch that has the ability to support simultaneous communication between all the nodes connected to it.
For simplicity, all the network ingress and egress links are assumed to have equal bandwidth capacities; however, all of the research presented in this report could be readily adapted to allow different capacities. In keeping with the general trend of providing increasing bandwidth availability to the customers, each of the ingress and the egress links is assumed to have a capacity of 155 Mbps, which corresponds to the capacity of an OC-3 link. Note that, as the main focus of this research is in the area of admission control, the network is assumed to be reliable and the bandwidth allocation algorithms do not account for link failures. Such an assumption is often made in research work that is not directly related to issues of fault tolerance and/or network reliability.

3.2. Overview of Heuristics Developed

The scheduling heuristic proposed in this research attempts to maximize the sum of the weighted priorities of satisfied requests (both session and data type) in a hierarchical (class-wise) manner. The heuristic works in an on-line fashion using preemption as a mechanism to improve performance. In this subsection, the on-line nature of the scheduling (i.e., bandwidth allocation) problem is examined in greater detail.

The heuristics considered in this work all share the characteristic of being on-line in nature. This implies that the scheduling algorithms receive the requests to be scheduled one-by-one as they enter the system. The algorithms cannot wait until the last request has arrived because by then, the start times of many of the requests that have already arrived might have expired and these requests might have to be dropped without even a consideration of their bandwidths. Another consideration is that, in most networks, the buffering capabilities in the network are much smaller as compared to the communication volumes involved. So, the algorithms need to make scheduling decisions without complete knowledge of future request arrivals. Any such algorithm that must make scheduling decisions about current requests without knowledge of future request arrivals is said to be an on-line algorithm.

An algorithm that has complete knowledge of the entire input sequence, before beginning the scheduling of the first request, is said to be an off-line algorithm. Off-line
algorithms typically tend to be much more time intensive (i.e., take longer to execute) as compared to on-line algorithms and can utilize well-known random search optimization techniques such as genetic algorithms and simulated annealing to accomplish their tasks (e.g., [BrS99]). One simple example of an off-line algorithm is an exhaustive search technique that enumerates through all possible solutions for the scheduling problem defined here before arriving at the optimal solution. Clearly, such a technique is severely limited in application by problem size. In contrast, on-line heuristics tend to be much faster because too long a running time per request might cause some unscheduled requests to be dropped even before the heuristic has had time to process them as their starting times have expired. Thus, implementation issues aside, off-line algorithms typically tend to outperform on-line algorithms for the same input sequence of requests being considered.

In this work, a variation of the on-line scheduling technique, called batch scheduling, is used. In batch scheduling, instead of scheduling each request as soon it arrives in the system, a set of such requests is collected in a batch and all the requests in a batch are then processed by the algorithms at a scheduling event (e.g., [MaA99]). The on-line nature of the heuristic is still preserved in that the heuristic does not have complete knowledge of future request arrivals. Also, this research does not make any assumptions regarding the future arrival of requests and the heuristics proposed could therefore be applied to a variety of scenarios with different arrival statistics.

Two different mechanisms are commonly used to batch requests together, a fixed interval approach and a fixed count approach (e.g., [MaA99]). In the fixed interval approach, the scheduler waits for a predefined interval of time and collects all requests that arrive within that interval in a single batch. In the fixed count approach, the scheduler collects a predefined number of requests in a batch before scheduling for that batch. In this research, a combination of the fixed count method and the fixed interval method is used for batch scheduling. The batching routines in this work are primarily fixed count driven and wait for a pre-determined number of requests to arrive before collecting them in a batch and triggering a scheduling event. However, the waiting time for the predefined number of requests to arrive is bounded by a suitable high value so that,
should the waiting time exceed this high value, the **batching** routines do not wait any more and the number of requests that have arrived at that point in time is considered as a batch and the scheduling event is triggered.

It is obvious that smaller the batch size, less the amount of information available to the scheduler regarding arrivals at that point in time. An upper bound on the batch size arises kom the consideration that the batch size should not be so large that some of the earliest-starting requests in the batch have to be dropped because their start times have expired. Within these two bounds, the actual value of the batch size used is generally determined by experimentation. In this research, the bound value for waiting time is set at 60 seconds and the fixed count number for batch size is selected to be equal to 100 requests.

### 3.3. Computational Complexity of the Scheduling Problem

The general **framework** for the on-line scheduling problem has been outlined in the preceding subsections and this subsection is devoted to an examination of its computational complexity. The important factors that tend to typically influence the computational complexity are the start time (or available time in case of a data type request), the **finish** time (deadline), the size of requested resource (bandwidth or data size) and the priority level associated with each request. It is a well-known result that even greatly simplified versions of the general scheduling problem described herein are known to be NP-complete. Provided below are the details of some of these versions.

1. The version of the scheduling problem where the start time, end time, and the amount of resource requested are all permitted to vary **from** request to request, is known to be NP-complete irrespective of whether the priorities are the same for all requests [GaJ78].

2. Similarly, considering that all requests demand the same amount of resource, if the priority associated with each request is allowed to vary along with their start and end times, then the problem is known to be NP-complete [Kar72].
The proofs of both these results rely on selecting a known NP-complete problem, the Knapsack problem, and showing that for any instance of the Knapsack problem, it is possible to construct a corresponding instance of the scheduling problem such that this instance of the scheduling problem is solvable in polynomial time if and only if the original instance of Knapsack is solvable in polynomial time.

Another source of complexity in the general problem framework is the decision regarding which calls to preempt and how many to preempt. In [GaG92], the authors have described several versions of the call preemption problem and proved the computational intractability of these versions. The call preemption problem is to minimize the "disruption" to the existing calls, such that if all the preempted calls are removed from the system, the network has the capacity to accept the new call under consideration. Several different metrics have been defined to capture the notion of minimum disruption:

1. the minimum number of calls that are to be preempted to schedule a new call (called the CPI problem);
2. The total bandwidth of the calls preempted (called as the CP2 problem); and
3. The total network bandwidth (i.e., bandwidth of call times number of hops in the route of the call) of the calls preempted (called the CP3 problem).

The authors in [GaG92] prove that the call preemption problem considering any of the three criteria mentioned above is NP-complete. This is done by selecting a known NP-complete problem, the Minimal Set Cover (MSC) problem, and showing that for any instance of the MSC, it is possible to construct a corresponding instance of the CPI problem such that this instance of the CPI is solvable in polynomial time if and only if the original instance of MSC is solvable in polynomial time. The proof extends trivially to the other two cases. In [PeK97, PeK98], the authors extend the above set of results and prove that the problem of selecting the best set of calls to be preempted, such that the sum of the priorities of the calls to be preempted is minimized, while minimizing the bandwidth to be preempted, is also NP-complete.

From the above discussion, the need for heuristics (as opposed to provably optimal algorithms) to solve the bandwidth allocation problem should be obvious. Any heuristic
that attempts to solve the scheduling problem outlined in the framework of this research has three main areas of computational complexity that need to be considered:

1. The problem of selecting the "best" set of requests to be satisfied given that three or more of their parameters (start time, end time, bandwidth, and priority) can vary simultaneously is known to be NP-complete.

2. The problem of selecting the "best" set of requests to preempt to schedule a new request is known to be NP-complete.

3. Session type requests have a fixed start and end time and bandwidth need as opposed to the data type requests for which all three parameters are flexible and need to be determined by the scheduling heuristic. So, while scheduling data requests, the heuristic should be able to decide both the amount of bandwidth that needs to be allocated to the data transfer operation and the time of transfer, to somehow minimize the "impact" on session type requests.
4. RELATED WORK

Sections 2 and 3 contain a description of the terminology and the environment in which this research was conducted. They also contain a clear definition of various terms and an illustration of the network model and processing mode of the heuristics. In this chapter, some of the related work in this field is explored to compare and contrast it with the work that has been presented here. To the best of the author's knowledge, there is no work in open published literature that tries to solve the preemptive bandwidth allocation problem where the connection requests are allowed to demand arbitrary bandwidths and data items of arbitrary sizes (in the hierarchical multi-class and multi-priority setting described herein).

There has been considerable work done on preemption algorithms in the context of requests that are allowed to ask for only a certain fraction of the bandwidth available at any node, or where only a single link network model is considered. Research activities have also explored non-preemptive bandwidth allocation both in the single class and multi-class scenario. Some of these activities are examined in detail here, to contrast and compare with the work presented in this report.

A bandwidth allocation scheme for call control and routing of multi-class traffic is developed in [GeK91]. The scheme is based on a load prediction mechanism where the prediction of call durations and new call arrival distributions is used as a mechanism for optimizing the projected network residual capacity, i.e., to maximize projected network revenue. The load predictor is based on the analysis of past data and general trends for the type of applications designed to run on the network. Should the network load exceed the projections, a load balancing technique is used to distribute the blocked calls uniformly among all sources.
Our research resembles [GeK91] in that the heuristics proposed also attempt to maximize network revenue (or net worth of satisfied requests) in a multi-class setting. It differs from the work in [GeK91] in many significant ways. This research utilizes preemption as a tool for improving on-line admission control performance whereas [GeK91] considers strictly non-preemptive call control and uses predictive techniques as a basis for its on-line decision making. Our research does not utilize prediction because in the expected AICE-like environment, the great variety of applications supported makes it very difficult to get an idea of traffic and also equally difficult to get usage data for a AICE-like network that has yet to be constructed.

In [GaG92, GaG93], the authors have presented one of the earliest studies of call preemption in on-line bandwidth allocation scenarios. Some of the basic concepts in our work are based on this work. However, in [GaG92, GaG93] all the calls are assumed to request a bandwidth that is equal to the link bandwidth. This implies that only one call can be served at any given time. Our model is considerably different in many respects because multiple requests of different classes can be simultaneously served. The performance criteria for the two heuristics proposed in [GaG92, GaG93] are the number of calls preempted and the total network bandwidth of the calls preempted, respectively. The performance parameter of our heuristic is the sum of the weighted priorities of the satisfied requests.

The work in [PeK97, PeK98] is an extension of the basic call preemption work presented in [GaG92, GaG93]. The authors in [PeK97, PeK98] also considered a more sophisticated multi-link network model than in [GaG92, GaG93]. A couple of preemption based admission control algorithms for prioritized requests have been proposed. The first algorithm is of exponential complexity and optimizes the criteria of (i) the bandwidth to be preempted, (ii) the priority of the connections to be preempted, (iii) the number of connections to be preempted, in that order. The second algorithm is of polynomial complexity and optimizes the criteria of (i) the number of connections to be preempted, (ii) the bandwidth to be preempted, and (iii) the priority of the connections to be preempted, in that order. A detailed simulation study of these two algorithms is
conducted and it has been demonstrated that the polynomial algorithm is almost as good as the exponential one.

**Our** work could be considered as an extension of the work in [PeK97, PeK98] with some important differences. This report also considers algorithms to perform admission control in an on-line setting. Similar to [PeK97, PeK98] we also use preemption as a tool to schedule requests that are worth more and some of the preemption related rules detailed in [PeK97, PeK98] have also been incorporated in our work to schedule requests in some scenarios.

The algorithms in [PeK97, PeK98] are decentralized in nature with each link manager in the multi-link network model executing these algorithms independent of the other link managers. **Our** algorithms are centralized in nature and, unlike the algorithms of [PeK97, PeK98] do not try to optimize the three different objectives mentioned above. The algorithms presented in this report maximize only one parameter and that is the sum of the weighted priorities of the requests. We also differ from the work in [PeK97, PeK98] by considering data type requests (in addition to session type requests) and classes (in addition to priorities).

A bandwidth allocation mechanism utilizing preemption has been presented in [BaC99]. The authors present a set of algorithms that may seem surprising and counter-intuitive at first sight but are, in reality, simple and efficient in performance. The basic approach used in designing algorithms in [BaC99] is that in deciding the calls (sessions) to be preempted, the algorithms consider only the duration of a call and the time for which that call has been already in service and completely ignore the throughput (i.e., bandwidth times duration) of the call. Thus, a call with a very large throughput could be preempted to make room for a call with a much longer duration and much smaller throughput.

All the algorithms proposed in [BaC99], implement various methods of reconciling between the need to avoid preempting jobs that have been running the longest and the need to schedule jobs that will run for the longest time in the future. The authors use competitiveness (which is the ratio of the throughput achieved by their algorithm to the throughput achieved by the best off-line scheduler) as a measure of performance of their
algorithms. They prove that in the case where each call is allowed to request only a certain fraction of the capacity of a link, their algorithms are optimal in an on-line scenario.

The work presented in this report differs from [BaC99] in that we use the sum of weighted priorities as a measure of the performance of our algorithms instead of competitiveness (there are no priorities or classes in [BaC99]). Also, none of algorithms or results developed in our work depend on the calls being able to request only certain fractions of the link capacities (i.e., all our algorithms and results hold even if some of the calls can demand bandwidths greater than the link bandwidths, for example). Finally, we also consider bandwidth allocation for data type requests in addition to session type requests.

In [HaS98], the authors have presented scheduling heuristics that maximize network throughput in the presence of two call classes with static ordering between the classes in terms of importance. The heuristics presented work at a per-packet level and a couple of greedy approaches have been proposed that have the ability to determine the packets to be dropped before their deadlines. The main goal of the heuristics is to maximize the throughput of one class of packets while maintaining throughput optimality for the total packet stream composed of packets of two or more call classes. The work presented in this report also develops scheduling heuristics that operate in a multi-class or hierarchical scenario in which there is a strict ordering of the classes from the most important to the least important. However, our work attempts to maximize the sum of weighted priorities (worth) of all requests satisfied in the most important class, which may not necessarily translate to network throughput optimization. Also in [HaS98], the concept of a priority within a class does not arise as it does in our work.

Another important difference is that while [HaS98] merges all data packets of all requests and then operates on the single resulting packet stream, this report considers end-to-end behavior on a request-by-request basis and does not consider individual packets at all. Hence, the rules for packet dropping in [HaS98] could roughly be compared to the rules for preemption in this work. A final point of differentiation is that in [HaS98] all the results proposed are for just two classes (high priority class and low
priority class) of requests only. (The authors have presented some suggestions as to how these results could be extended to cover more than two classes.) The heuristics and results presented in this report are independent of the number of classes actually selected in the system. The underlying grouping mechanism ensures that the results remain relevant irrespective of the number of classes selected.

In [MuS98], an analytical study of various admission control mechanisms in a multi-class video-on-demand system has been conducted. There are two classes of movie videos, popular (class 1) and unpopular (class 2) in the system and the authors have studied three types of admission control mechanisms, complete partitioning, complete sharing, and threshold type. In complete partitioning, the resources are completely partitioned and every class of request has access only to a dedicated set of resources. In complete sharing, all classes are always accepted if there is available capacity. The threshold based approach sets thresholds for class 2 requests while always accepting class 1 requests. The authors also conduct a numerical simulation to calculate various probabilities and to prove the usefulness of their threshold-based admission control policy over the other two mechanisms.

The work presented in this report is similar to [MuS98] in that we also propose admission control policies in a multi-class on-line scenario. We have also used simulations to justify the use of our admission control policies and demonstrate its performance under various loading conditions. There are some important differences between [MuS98] and our work. In [MuS98], a non-preemptive network is assumed but in our work preemption is a valuable tool for increasing heuristic performance in the on-line setting. Also in [MuS98], there are no priorities within classes. The main objectives of the admission control policy in [MuS98] is to minimize the blocking rate (the ratio of rejected requests of a class to the total requests of that class) for the two classes of requests whereas in this report the main objective of the admission control policy is to maximize the sum of the weighted priorities of the most important class. Our work also considers data type requests in the admission control and scheduling aspects while [MuS98] considers only session, i.e., bandwidth type requests.
A game theoretic approach for link capacity allocation is presented in [KoL97]. The scenario of non-cooperating users, each setting their own selfish policies for optimizing their performance objectives, is considered in detail. In such a scenario, while allocating the link capacities, the user specified lower bounds and the network-related upper bounds need to be considered. The authors in [KoL97] consider a set of design rules such that the Nash equilibria, corresponding to the stable operating points of the network, exhibit certain "desirable properties" according to various efficiency criteria. Several different efficiency criteria have been considered including the "price" as seen by each user, the cost of each user to the network, network delay, etc.

Our work considers heuristics to perform bandwidth allocation for a given fixed AICE-like network as opposed to developing rules for network design (in terms of capacity needed for a given set of loads) as in [KoL97]. The design mechanisms in [KoL97] are based on game theoretic analysis of the system to maximize network efficiency while the heuristics in our work maximize the sum of weighted priorities of the most important class of requests.

In [MaS99], the authors have proposed a multi-class QoS routing algorithm for bandwidth sharing. The main goals of the algorithm proposed are to improve the performance of best effort traffic, while at the same time admitting the same number of guaranteed QoS sessions as would be admitted by the network if it were dedicated to the guaranteed effort (i.e., QoS) traffic. The algorithm achieves its objective by using a link cost for QoS traffic that is based on the "virtual residual bandwidth" (i.e., the non-reserved bandwidth) instead of the actual residual bandwidth. The virtual residual bandwidth captures the congestion conditions of the best-effort traffic, i.e., lower than actual residual bandwidth on links that, relative to the rest of the network, carry more best effort traffic and higher on links that have little best effort traffic. The algorithm achieves admission control by making some links in the network more or less attractive to the network traffic and thus helps to optimize best-effort traffic performance.

The work presented in this report attempts to maximize the sum of weighted priorities of the satisfied requests instead of the network throughput as in [MaS99]. We also do not consider any best-effort traffic requests. The algorithms in [MaS99] do not
take in to account the scheduling of data requests and treat all requests as either best-effort or bandwidth type.

The research work presented in [PiWOO] may appear, at first reading, to be very similar to the work presented in this report. In [PiWOO] the authors present a research allocation mechanism that performs on-line bandwidth allocation for both volume (i.e., data) and bandwidth (i.e., session) type requests. The heuristics presented also attempt to maximize the sum of weighted priorities of the requests as is done in our work. Similar to our work, [PiWOO] also considers on-line scheduling in batch mode and the heuristics proposed are greedy in nature. However, in spite of the apparent similarities between the work presented in [PiWOO] and this report, there are some significant differences.

In [PiWOO], there is no concept of a class and a priority level within a given class. All requests have only a predefined priority level and the algorithms in [PiWOO] maximize the sum of weighted priorities of all requests. Our system model incorporates the concept of a class and all requests have a priority level within a given class. Hence, our heuristics work on a class-by-class basis, first maximizing the sum of weighted priorities of the most important class, next the second most important class, and so on down to the lowest class as opposed to maximizing the weighted priorities of all requests. In [PiWOO], the requests can be degraded, before they begin transmission, to obtain a corresponding fraction of the associated utility and hence worth. In our work, the requests cannot be so degraded (i.e., there is no partial utility to be gained, corresponding to a smaller bandwidth allocation) and the potential utility (worth) gained from a request is full if the request completes successfully and zero if it does not. Furthermore, in our work any request can be preempted before its end time, while in [PiWOO], a request that has begun transmission cannot be preempted. Thus, our underlying system model differs from the one in [PiWOO] in many significant ways as outlined above.
5. PROPOSED HEURISTIC

5.1. Overview

Having established the computational intractability of the bandwidth allocation problem and hence the need for heuristics in Section 3, in this chapter a scheduling heuristic with all its details is presented. The heuristic presented here attempts to maximize the sum of the weighted priorities of the satisfied requests in the most important class followed by the sum of the weighted priorities in the second most important class and so on down to the least important class. A conceptual description of the heuristic is provided in this chapter along with more detailed pseudo code in Appendix 1. The actual Java source code is in [Nai00]. Section 5.2 presents the details of the batching and sorting routines used in the heuristic, while Sections 5.3 and 5.4 explain the scheduling of session and data requests, respectively.

5.2. Scheduling Heuristic

As explained earlier in Section 2, the heuristic proposed in this work schedules two different kinds of requests: session type requests and data type requests. Session type requests need a given bandwidth for a specified start time and duration while data type requests need a data item of specified size that is available at the source from a specified time (called the available time) and that must reach the destination before a certain time (called the deadline). Data type requests also have a maximum bandwidth associated that indicates the maximum rate at which the source and destination can transfer data. An assumption is made here that this maximum bandwidth is always equal to the maximum capacity of the link, i.e., 155 Mbps.
The heuristic operates on batches of requests and the batch size is determined as explained earlier in Section 3. After a scheduling event has been triggered for a given batch of requests, the heuristic applies three different sorting routines to each batch of requests. First, the requests in each batch are sorted according to class such that all requests belonging to the most important class, class 1, are grouped together first, followed by requests of class 2, and so on down to class $C$, where $C$ indicates the total number of classes present in the system ($C = 3$ in this work). Then, the requests in each class are separated into session type requests and data type requests. Finally, the requests of each type, session and data, are further sorted by their worth-per-bit in descending order such that the request that is the most valuable per-bit is placed first. The worth-per-bit value is calculated for each request as the weighted priority of the request divided by the number of bits requested (i.e., bandwidth times duration for session type and data size for data type). This corresponds to steps 1-6 in function main() in Appendix 1.

After the three sorting routines have been applied to the requests in a batch, the heuristic then begins the actual bandwidth allocation process on a request-by-request basis in the order in which the sorting routines have placed the requests. Two slightly different techniques are used to schedule session and data type requests. First the scheduling of session type requests is explained in detail.

5.3. Session Type Request Scheduling

Assume a session type request $r$ such that:

$b_r =$ begin time or start time of $r$
$\hat{f}_r =$ finish time of $r$
$i_r =$ network ingress link of $r$
$e_r =$ network egress link of $r$
$c_r =$ class of $r$, $1 \leq i \leq C$ ($C =$ total number of classes in the system)
$p_r =$ priority level of $r$, $1 \leq p, P$ ($P =$ total number of priority levels in class $c_r$)
$t_r =$ type of $r$ (session or data; considering only sessions in this subsection)
$bw_r =$ bandwidth requested by $r$
Then \( r \) can be denoted as:

\[
r = (b_r, f_r, i_r, e_r, c_r, p_r, t_r, b_{wr})
\]

The heuristic first checks the network ingress and egress link capacities to determine if there is sufficient unutilized bandwidth at both places (corresponds to function \textit{check\_used\_bw()} in Appendix 1). If the ingress and egress links have the requisite bandwidth available then the request \( r \) gets scheduled and suitable entries are made in the allocation lists maintained by the ingress and egress links (corresponding to function \textit{change\_link\_status()} in Appendix 1).

If such capacity is not found, then the heuristic builds a list at both the ingress link and egress link of the previously scheduled requests that are active during the lifetime of request \( r \). A request \( k \) can be said to be active at a given link during the lifetime of \( r \) if \( b_k \leq f_k \) and \( f_k \geq b_r \), and the request \( k \) has been allocated but not preempted. The heuristic then calculates the sum of the bandwidths used by all session type requests of class \(< c_r \) in each of these lists at each point in time between \( b_r \) and \( f_r \). If the maximum of this used bandwidth is such that the addition of \( b_{wr} \) to it causes the link capacity (155 Mbps) to be exceeded, then the request \( r \) is dropped from the heuristic and put into the unscheduled-requests list. This indicates that the bandwidth occupied by the session type requests of more important class is so much that \( r \) cannot be scheduled without freeing up some (or all) of this bandwidth. However, as all these requests are of session type and of a more important class this cannot be achieved. In this case the request \( r \) is said to be blocked by session type requests of more important classes (steps 1-9 in function \textit{main\_session\_scheduler()} in Appendix 1).

If the link bandwidth is not exceeded as described above, consider the case where the only possible way for the request \( r \) to be scheduled is for a data type request of a more important class to be "repositioned" (function \textit{data\_reposition()} in Appendix 1). A data type request can be repositioned by scheduling it in a time window that does not overlap with the duration of the request \( r \). This scheduling can be done either by using the unused bandwidth in the non-overlapping time window or by allowing the data type request to cause preemption in that time window. Note that, should preemption be needed while repositioning, the data type request being repositioned is only allowed to
preempt requests that are of class $> c_r$. This ensures that the heuristic does not end up preempting one or more requests of class $\leq c_r$ to schedule a request of class $c_r$.

Due to the time required to perform a data repositioning, in this situation, only the blocking data type request occupying the maximum bandwidth is considered for repositioning. Future work could examine other variations. If such a repositioning is possible and successful, both at ingress and at egress if required, then request $r$ is maybe scheduled using the capacity freed by repositioning. If this bandwidth is not enough for scheduling $r$, repositioning of other requests of class $\geq c_r$ may be required for the remaining needed bandwidth. The method for doing this is the same as for the case where no repositioning is done and all of the bandwidth needed by $r$ is attempted to be obtained by preemption. If the repositioning fails, or if no data type request is found at ingress and/or egress link to reposition then the request $r$ is dropped and added to the unscheduled-requests list (steps 10-31 in function main_session_scheduler() in Appendix 1).

If $r$ does not require data repositioning, then the heuristic tries to schedule the request $r$ by using preemption (function preempt-routine() in Appendix 1). To do this, the heuristic first checks to see if the current request is blocked by requests of class $c_r$. It does this by calculating the maximum of the sum of the bandwidths of all simultaneously active requests of class $\leq c_r$, within the duration of $r$ (see Figure 2). Let this bandwidth be $bw_{le}$ (where $le$ stands for less than or equal to). Let the maximum of the sum of the bandwidths of all simultaneously active requests of class $< c_r$ be $bw_I$ (I stands for less than). If these bandwidths are such that:

$$bw_I + bw_{le} \leq bw_{le} + bw_r,$$

then, the request $r$ is said to be blocked by requests of the same class (see Figure 2). This implies that for the request $r$ to be scheduled, some (or all) of the capacity used by the blocking requests of class $c_r$ needs to be freed up. In such a case, the heuristic selects preemption candidates from the set of blocking requests of class $c_r$ active at the (ingress and/or egress) link.
The preemption candidates are selected by first selecting the conflicting request of class $c$, with the lowest priority followed by the request with the next lowest priority and so on until the potential bandwidth preempted of all requests in the preemption candidates list equals or exceeds the needed bandwidth (needed bandwidth = $bw_r + bw_{le} - \text{link capacity}$). If there is more than one request of a given priority level to be selected then the request that occupies the most bandwidth in the duration of $r$ is selected first. This ensures that for the same loss of worth of a preempted request, the maximum bandwidth is obtained from the preemption. (The heuristic bases this selection on bandwidth occupied in the conflict region and not the total amount of bits being preempted as bandwidth is the resource being allocated.)

The heuristic then decides whether or not to preempt these requests by comparing the sum of the weighted priorities of the requests in the preemption candidates list (both at
ingress and egress, not counting any request twice) with the weighted priority of \( r \). Let the total number of such requests selected for preemption at both ingress and egress is be equal to \( q \). Now if:

\[
\text{weighted priority of } r > \left\{ \sum_{u=1}^{q} \text{(weighted priority of request } u) \right\}
\]

then, the heuristic preempts the \( q \) requests selected earlier with a certain probability and schedules \( r \). If the bandwidth freed up by preemption is not sufficient to schedule \( r \), the remaining bandwidth needed is obtained by preempting requests of class \( c_r \). The rules for this preemption are given later in the case where \( r \) is blocked by requests of only less important classes. If the weighted priority of \( r \) is \( \leq \) the sum above, then the request \( r \) is dropped and added to the unscheduled-requests list.

A probabilistic preemption policy is followed to allow requests that might needlessly get preempted some chance of being able to continue in the system. To better illustrate this situation, consider Figure 3.

**Figure 3:** Situation where request 1 is needlessly preempted by request 2, as request 2 ends up getting preempted by request 3, where \( k < i \).
In Figure 3 above, to schedule request 2, request 1 is preempted (because request 2 is worth more), and to schedule request 3 (class k < class i), request 2 is later preempted. If request 1 was already in progress at the time that it was preempted, the weighted priority represented by it is lost once it is preempted and completely dropped from the system. However, this is an undesirable situation because both requests 1 and 3 could have simultaneously existed in the system had the arrival of request 3 been known earlier. Hence, to attempt to account for such cases, a variation of the heuristic that includes a probabilistic preemption scheme is adopted by the heuristic.

Thus, when making preemption decisions amongst requests in the same class (i.e., intra-class (worth-based) decisions as opposed to inter-class (hierarchy-based) decisions), the heuristic adopts a probabilistic approach and generates a random number \( rand \) from 0 to 1. The preemption is carried out only if \( rand > x \) (where \( 1 - x \) is defined to be the preemption probability) and \( r \) is scheduled in place of the preempted requests. This can be interpreted as allowing the lower priority requests to continue with a probability of \( x \) and discarding request \( r \) even though it was worth more to the system at that point in time. The preemption probability \( (1 - x) \) acts as a sort of ”globalization factor” as it tends to make the heuristic less greedy by assuming that instead of losing the worth of the preempted \( request(s) \) to a more valuable preempting request at some given instant in time, the current requests are allowed to continue with the expectation that, for example, the preempting request itself would later be preempted.

The factor \( 1 - x \) depends upon a lot of parameters such as the relative \textbf{worths} of the preempting request and the preempted \( request(s) \), the bandwidth requested by the preempting request as compared to the bandwidths of the preempted \( request(s) \), the relative durations, the time instant at which the preemption occurs. Hence, there is no simple way to determine the factor \( x \) for various cases in which the heuristic may be used. In this research two different values of this factor, \( x = 0 \), and \( x = 0.1 \), have been evaluated. (In addition, lower and higher values have also been tried out apart from the two mentioned above to determine the best performance of the heuristic by performing sample experiments.) Thus, in one case (\( x = 0 \)), a higher priority request always preempts
lower priority ones whereas in the other case \((x = 0.1)\), a higher priority request preempts with 90% probability.

Note that an exactly similar situation could arise in an inter-class scenario when, for example, a class 3 request is needlessly preempted by a class 2 request which itself gets preempted later by a class 1 request. However, when making inter-class decisions, the preemption is always deterministic, i.e., the less important class request always gets preempted in favor of the more important class one. This is in keeping with the strict hierarchical ordering of the class-based policy that implies that a request of a more important class should always be satisfied ahead of any number of requests of less important classes. Hence, even though by scheduling the class 2 request in the example above, the system may needlessly lose out on a class 3 request, the scheduling heuristic will always choose to preempt the request of class 3. Thus, a request of a more important class is given every possible chance to complete relative to a request of a less important class. It is only when comparing requests of a given class that weighted priority considerations arise and hence a probabilistic approach is adopted which, it is hoped, would result in a maximization of the weighted priorities of a given class.

Recall that if the capacity freed by any data repositioning or any intra-class preemption is still not sufficient to schedule \(r\) then the preemption of requests of class \(> c_r\) may be required. The rules for this case are exactly the same as those for the scheduling of \(r\) when it is blocked only by requests of less important classes explained below.

A final case in the scheduling of session type requests that may arise is that the blocking requests are all of classes \(> c_r\). In such a case, the heuristic selects active requests to preempt, starting with requests of the lowest worth within the least important class (and highest bandwidth if more than one request has the same lowest worth in the least important class) and adding to this set until the bandwidth of the preempted requests is equal to or greater than the needed bandwidth. When such a set of requests is located, the heuristic schedules \(r\) by preempting all requests in this list.

Note that every request \(r\) scheduled with preemption triggers a post-preemption scheduler. This post-preemption scheduler attempts to allocate the excess preempted
capacity to requests in the unscheduled-requests list. The post-preemption scheduler runs the same scheduling heuristic described above on requests in the unscheduled-requests list such that each of the requests selected for post-preemption scheduling has a ingress or egress that is identical to the ingress and/or egress of the request(s) just preempted and whose class is \( \geq c \). The requests are kept in the unscheduled-requests list until either their start time expires (for session type requests) or until either the deadline expires or the bandwidth needed in the time left before the deadline exceeds the link bandwidths (for data type requests).

### 5.4. Data Type Request Scheduling

Data type requests differ from session type requests in that they do not have fixed start and end times except for the condition that their transmission start on or after their available time and end on or before their deadlines. This implies that the heuristic needs to determine both the amount of bandwidth to allocate and the time period during which to allocate it. The important conditions that need to be satisfied in selecting this time duration are that: (i) the block of time selected should be contiguous, and (ii) the duration of the time block multiplied by the minimum bandwidth available during all of that period should be \( \geq \) data size requested.

In this work, data type requests are always attempted to be scheduled for the longest possible duration, i.e., with the minimum possible bandwidth. It is expected that this policy will cause minimum interference with session type requests at any given point in time. Consider a data type request \( r \) being scheduled such that:

- \( a_r \) = available time of \( r \)
- \( l_r \) = latest time or deadline of \( r \)
- \( i_r \) = ingress link of \( r \)
- \( e_r \) = egress link of \( r \)
- \( c_r \) = class of \( r \), \( 1 \leq c_r \leq C \) (C is the total number of classes in the system)
- \( p_r \) = priority of \( r \), \( 1 \leq p_r \leq P \) (P is the total number of priority levels within class \( c_r \))
- \( t_r \) = type of \( r \) (session or data; considering only data type request here)
$size_r = \text{size of data item requested by } r$.

Then $r$ can be denoted as:

$$r = (a_r, l_r, i_n, e_r, c_r, p_r, t_r, size_r).$$

The heuristic first tries to schedule the request $r$ by utilizing the unused bandwidth in the network. To achieve this, the heuristic first computes a unified resources picture of the network ingress and network egress links of $r$. This resource picture is computed by taking the maximum of the occupied bandwidths on each of the links at each point in time in the duration of $r$. The heuristic then tries to schedule $r$ in the longest contiguous block of time with sufficient capacity that can be found in this unified resource picture (function check_possible_data_schedule() in the Appendix 1). The algorithm to locate empty capacity on the network ingress and network egress links will always find the required capacity should such capacity exist, i.e., it is not heuristic in nature.

If no such block can be found, then the heuristic attempts to schedule the data type request with preemption by treating it like a session type request with a bandwidth need given by $size_r / (l_r - a_r)$, start time given by $a_r$, and end time given by $l_r$ (function main-data-scheduler() in Appendix 1). This corresponds to scheduling it with the minimum possible bandwidth. (It is assumed that the minimum schedulable bandwidth in the system is 1 Kbps. Thus, if $size_r / (l_r - a_r)$ is less than 1 Kbps, the bandwidth required is 1 Kbps from time $a_r$ until $a_r + size_r / (l_r - a_r)$.)

Should the data type request get scheduled successfully in the steps just described above, then, it represents the most "flexible" schedule of the data item because it may be possible to reschedule it later, if needed, with more bandwidth. The preemption related rules are exactly the same for the data type requests as described in Section 5.3 for the session type requests. The scheduling of the data type request with preemption does not, however, cause any data repositioning algorithms to be initiated. This condition ensures that the heuristic does not spend too much time on scheduling a single request. (The data type request might cause a reposition of another data type request, which may in turn, cause more preemption and/or reposition. To limit the time spent on these, the condition of no repositions has been imposed. Future work may include heuristics without such a constraint.)
After both the data and session type requests belonging to a given class have been processed, the heuristic then considers scheduling requests of lower classes on a class-by-class basis until all the requests in a given batch are processed. This process is then repeated for all batches in the system.
6. PERFORMANCE COMPARISONS

6.1. Overview

In this chapter, a set of comparison heuristics is presented to examine the performance of the heuristic suggested in Section 5. There have been some attempts in the related literature to provide performance bounds for on-line admission control in a preemptive scenario. In [DaS99], the authors have presented a set of upper bounds for the performance of admission control and routing policies for networks with multiple call classes. However, these bounds cannot be applied directly in this work because the bounds in [DaS99] are for the single-link model of the network and also the system model considers only session type (i.e., bandwidth) requests and does not consider data type requests.

Bounds on the competitiveness of preemption algorithms are presented in [CaI98] for any randomized scheduling scheme. Our work is mainly simulation based and a competitiveness analysis is not performed here. Also, our heuristic contains scheduling policies for data type requests too and the applicability of purely preemption-based bounds to the heuristic is unclear. Hence, these bounds are not useful for comparing the performance of the heuristic suggested in this work. In the following subsections, a simple scheduling heuristic and three upper bounds are presented that help to compare the performance of the heuristic suggested in this work. The three upper bounds and the simple scheduler provide us with a value for the sum of the weighted priorities in each request class considered in the system.
6.2. Simple Scheduling Technique

A simple scheduling technique used to compare the performance of the heuristic presented in this work at the lower end is based on the complete sharing policy [BoM98]. Assume that the heuristic is to be run for a set $S$ of requests (i.e., $S$ represents the set of all requests, arriving in an on-line fashion into the system, for which the heuristic will perform scheduling decisions; for details regarding the manner in which the set of requests is generated, see Section 7.1). The complete sharing heuristic gives the sum of the weighted priorities of satisfied requests for the set $S$, assuming that there is no formal admission control mechanism in the system.

Thus, the requests are satisfied on a first-come-first-served basis as they arrive in the system, solely on the basis of available link capacity. As soon as the link capacity is exhausted for a given link for a given point in time, no further requests asking for bandwidth on that link for that time are satisfied. This process continues until all of the requests in $S$ have been processed.

The sum of the weighted priorities of satisfied requests for a given class, gives us the performance of the system for a set $S$ of requests assuming that no admission control type processing is performed and a very simple mechanism (complete sharing) is used to allocate requests. The main goal of the heuristics presented in this work is to maximize the sum of the weighted priorities of the most important class followed by the second most important class and so on. This may imply that while the heuristic outperforms the complete sharing policy for the most important class (class 1), it may not exhibit the same behavior for all classes. In some cases, such as for the least important class, the heuristic may actually provide a lower sum of weighted priorities as compared to the sum of weighted priorities provided by the complete sharing policy for that class. The class 2 values for the sum of weighted of priorities of the complete sharing heuristic would become significant only if the class 1 performances of the complete sharing heuristic and our algorithm were comparable in some scenario. A similar argument holds
for the class 3 values of weighted priorities \textit{(i.e., only when the class 1 and class 2 performances are comparable)}.

That there may be cases when the heuristic suggested here might actually provide a lower performance than complete sharing in even class 1. If such cases exist and are \textit{known} before hand, then it is expected that a complete sharing policy will be followed in such scenarios. (The simulation results presented in Section 8 indicate that for the simulation scenarios selected in this work, the heuristic always outperforms the complete sharing policy by a considerable margin.)

6.3. \textbf{Upper Bounds}

6.3.1. \textbf{Loose upper bound}

Assume that \( S \) is the set of all requests to be scheduled by the heuristic. The loose upper bound is obtained by simply adding the weighted priorities of all requests belonging to a given class in \( S \). The loose upper bound provides a measure of the best performance possible for \textit{any} heuristic (on-line, off-line, exhaustive search, random search, etc.) that may operate on that particular set of requests. Because this bound assumes that all requests are satisfied, in most cases it is very optimistic. Thus, there is a need for tighter upper bounds when oversubscribed networks are considered.

6.3.2. \textbf{Ingress upper bound}

The \textit{ingress upper bound} is obtained by assuming the best possible allocation of bits for every instant that the network is active. Again, assume that \( S \) is the set of all requests that need to be scheduled. Let \( t_{\text{earliest}} \) be the earliest start time (or available time) of any request in \( S \) and let \( t_{\text{latest}} \) be the latest finish time (or deadline) of any request in \( S \) (some requests in \( S \) may be "truncated" as explained in Section 7.2). The network \textit{active time}, \( t_{\text{act}} \), is defined as:

\[ t_{\text{act}} = (t_{\text{latest}} - t_{\text{earliest}}) \]
For every node in the network under consideration, the ingress link capacities are obtained and the maximum allocable bits for each ingress link over the length of the network active time are obtained. In this work, as all network ingress link capacities are equal to 155 Mbps, the maximum allocable bits can be obtained as:

\[ \text{max. allocable bits} = (155 \times 10^6 \times t_{\text{act}}). \]

All requests in \( S \) are then sorted by class, with the most important class first, and by ingress link. The worth-per-bit of each request is calculated as:

\[ \text{worth-per-bit} = \frac{\text{weighted priority}}{\text{bits requested}} \]

where, for a session type request:

\[ \text{bits requested} = (\text{duration} \times \text{bandwidth}), \]

and for a data type request:

\[ \text{bits requested} = \text{data size}. \]

The requests within each class belonging to a particular ingress link are then sorted in a descending order by their worth-per-bit values. The requests in this sorted batch, taken in the sorted order, are then allocated the corresponding bits requested such that the sum of all allocated bits equals the maximum allocable bits value calculated above. These requests are termed the "satisfied" requests for that particular ingress link. Let the sum of the weighted priorities of the satisfied class \( i \) requests for ingress link \( j \) be \( \sigma_i^j \). Then the ingress upper bound for a given class is defined as the sum of the \( \sigma_i^j \)s for all \( j \) (in this work \( 1 \leq j \leq 15 \)).

As the allocation of the bits is bounded by the maximum allocable bits, it is possible that for the last request to be allocated at each node (considered in the sorted order) the bits actually allocated might exceed the maximum allocable bits. Hence, for this last request, the full worth cannot be considered, as this would correspond to an allocation of bits greater than the maximum possible. The worth corresponding to this last request to be satisfied at a given ingress is taken to be a fraction of its total worth. This fraction corresponds to the fraction of bits of the request that make up the difference in bits between the net allocated bits up to the second-to-last allocated request and the maximum allocable bits per ingress link. The ingress upper bound represents the best
possible utilization of the bits available at a given ingress link, in terms of obtaining the maximum weighted priority per bit, summed over all ingress links in the network.

By considering only the bits needed for every request, the ingress upper bound ignores the actual arrival, start (available), and finish (deadline) times of the requests. By considering all the requests concurrently, it also ignores the on-line nature of the scheduling problem and assumes that all requests to be scheduled are known in advance. Finally, by considering only the ingress link and not the egress link of each request, an assumption is automatically made that the corresponding capacity is available even at the egress link for all requests satisfied at the ingress link. Thus, the ingress upper bound may not be realized. The ingress upper bound is a tighter upper bound than the upper bound in Section 6.3.1, as it considers the maximum allocable bits available at a given ingress link as a constraint on the number of requests that can be satisfied, instead of simply summing the weighted priorities over all requests in the set of requests for scheduling.

6.3.3. Egress upper bound

The egress upper bound is calculated in exactly the same way as the ingress bound just by considering the egress links of all requests instead of the ingress link. The previous subsections present two different tighter upper bounds, the ingress upper bound and the egress upper bound. The ingress upper bound provides us with a better estimate of the relative possible performance in the case where most of the requests have only one or two ingress links as sources but are uniformly distributed among all egress links for their destinations. In such a case, the egress upper bound would tend to provide a very optimistic estimate in terms of the sum of weighted priorities that can be satisfied. The ingress upper bound is expected to provide a much tighter estimate of the requests that could possibly be satisfied. The egress upper bound would similarly provide a better estimate of the performance when the requests in the systems originate from many different ingress links but have only one or two links as their egress points. In the average case, assuming that both ingress links and egress links occur with equal
probabilities among all the available links in the network, both bounds should provide comparable estimates of performance.
7. SIMULATION EXPERIMENTS

7.1. Overview

In this chapter, a detailed description of the simulation experiments implementing the proposed heuristic has been provided. The network model used in the simulation experiments has been detailed in Section 3. The network is assumed to consist of fifteen nodes, where each node can act either as a source or as a destination for a given request. The requests are assumed to be uniformly distributed among all fifteen nodes as sources and destinations. Each node has a network ingress and egress link connecting it to the high-capacity backbone network. Without loss of generality, the capacities of all the links are assumed to be the same and equal to 155 Mbps per link. Note that most of the parameter values chosen in these experiments are assumed to be representative of one type of AICE-like environment and are summarized in Table 1.

7.2. Request Generation Details

The requests in the system are assumed to have equal probability of being either of data type or session type and to be uniformly distributed among three classes and four priority levels within each class. Each run of the simulation is carried out for a time interval corresponding to the time difference between the earliest start time or available time of a request in the set of requests to be scheduled to the 2000th finish time (or deadline) in the set. Thus, the simulation time interval corresponds to the time taken by
the "earliest finishing" two thousand requests. All requests with start times (or available times) on or after the 2000th finish time (or deadline) are ignored.

If $S$ is the set of all requests to be scheduled by the system, then the set $S$ contains all those requests that have start times (or available times in case of data) before the simulation end time. Thus, $|S|$ may be greater than 2000 with some requests beginning before the 2000th end time but not ending before this time. All such requests are considered to be "truncated requests and for such requests, only those bits that lie before the simulation end time are considered. The worths of such requests are then prorated based on the ratio of total bits in the request to truncated bits. For the simulation experiments conducted, these truncated requests numbered between 70 and 105 for each set of the 2000 normal requests.

The heuristic considers only the worth of those truncated requests that are active (allocated and not preempted) at the simulation end time. The "truncated worth" included in the calculations can be considered to be an "edge effect" similar to the edge effect encountered at the start of the simulation ("warm up") when the system is mostly empty and gradually gets loaded as more and more requests arrive. The number 2000 has been selected to be large enough so that the initial "warm up" transient and the "edge truncation" effects have relatively minor effects on the performance calculations and hence do not affect the validity of the results significantly.

The request arrival process is modeled as a Poisson process (i.e., exponential distribution for the request inter-arrival times) with two different arrival rates corresponding to two different loading levels. To characterize the load on the system, the concept of loading factor is defined to be a ratio of the offered load to the maximum load supported by the system.

If $S$ is the set of all requests to be scheduled by the system, where $s$ is a session type request and $d$ is a data type request,

$$\text{offered load} = \sum_{s \in S} (\text{duration}, \times \text{bandwidth}_s) + \sum_{d \in S} \text{sized}$$
The **maximum load** on the system is defined as the maximum number of allocable bits available in the network for the simulation period. Hence, as there are 15 ingress links,

\[
\text{maximum load} = \max \text{ allocable bits} \times 15.
\]

(where max. allocable bits was defined in Section 6.3.2.)

The simulations have been conducted for arrival rates corresponding to loading factors of 0.7 and 1.2 to simulate the performance of the heuristic under different loading conditions. The arrival rates can be estimated from the offered load as follows.

The average number of bits per second (bps) the system needs to schedule is given by:

\[
\text{average bps} = \frac{\text{offered load}}{\text{simulation time}},
\]

Next define

\[
\text{average requests per second} = \frac{\text{avg. bps}}{\text{avg. bandwidth needed per request}}
\]

where the average number of request to be handled by the system per second could be approximated as the number of new arrivals in the system per second, i.e., the arrival rate of the requests.

The average bandwidth needed per request is calculated as:

\[
(0.5 \times \text{average bandwidth of session type request}) + (0.5 \times \text{average data size of all data type requests} / \text{avg. of } (\text{deadline} - \text{available time})).
\]

The arrival rates obtained in the following manner are rough estimates of the actual arrival rates needed to obtain the loading factors of 0.7 and 1.2. The actual arrival rates are determined by trial-and-error, starting with the estimates obtained above and "tuning" the arrival rates to attain the desired loading factors.

The time difference between the request arrival time and the request start time (the *lead time*) is assumed to be uniformly distributed from 2 minutes to 2 hours for both session and data. The batch size for processing is set at 100 and the maximum waiting time for this size of 100 requests to be reached is set at 60 seconds (i.e., after 60 seconds a scheduling is performed even if the batch size is less than 100). Because of the relatively high arrival rates imposed by the loading factors of 0.7 and 1.2, the maximum waiting time is never exceeded in the simulation studies and scheduling events are always triggered with a batch size of 100.
7.3. Request Size and Duration

The size and duration parameters selected for the session type and data type requests are presented in this subsection. Typically, an exponential distribution is used to model the file sizes and request durations in communications and queuing theory related research activities. However, recently there has been considerable research into the use of heavy-tailed functions to model request duration in communication networks. In [LeT94], the authors have demonstrated that Ethernet traffic is bursty or "self-similar" in nature across a wide range of time scales. Thus, Ethernet LAN traffic measured over microseconds and seconds exhibits the same second-order statistics as Ethernet LAN traffic measured over minutes and hours or even larger time scales. Intuitively, this scale invariance of measured Ethernet LAN traffic manifests itself in the absence of a characteristic burst length; Ethernet traffic is bursty on all (i.e., wide range of) time scales and plotting it over different time scales results in "similar-looking" pictures, all of which feature a distinctive burst-within-burst structure. In [PaF95], the authors have extended the study for Ethernet traffic to WAN and Internet traffic. One of the ways in which a self-similar process can be modeled is by the use of heavy-tailed distributions for some of the random variables involved in the models. In the field of network communications, the heavy-tailed model has been typically used to model call holding times as in [DuM94], and the frame sizes for variable-bit-rate video as in [GaW94].

In [PaK96], the authors have presented one possible explanation for the heavy-tailed nature of the network traffic in terms of file sizes. The file sizes for files typically transferred over the network tend to be heavy-tailed in nature giving rise to the observed self-similarity or long range dependence of the network traffic. Consequently, the property of self-similarity of the network traffic is usually modeled in simulation experiments via the heavy-tailed distribution of file sizes. So in our work, for session type requests this translates to a heavy-tailed probability distribution for the request durations, and for data type requests to a heavy-tailed distribution of the data item sizes.
A distribution is **heavy-tailed** if:

\[ P[X > x] \sim x^{-a} \text{ as } x \to \infty \]

where the parameter \( a \), called the shape parameter, satisfies \( 0 < a < 2 \). The approximation relation "\( \sim \)" means \( P[X > x]/x^{-a} \) approaches 1 as \( x \to \infty \).

One of the simplest heavy-tailed distributions is the **Pareto distribution** (also referred to as the power-law distribution, the double-exponential distribution, and the hyperbolic distribution) that has a cumulative distribution function (cdf) given by:

\[ F(x) = P[X \leq x] = 1 - (k/x)^a, \quad x \geq k \]

where \( k > 0 \) is the smallest possible value of the random variable. Its probability density function (pdf) is given by:

\[ f(x) = k^a / x^{a+1}. \]

The function given by \( 1 - F(x) \), corresponding to \( P[X > x] \), is sometimes called the "tail" of the distribution and for the Pareto distribution it is given by:

\[ P[X > x] = (k/x)^a. \]

Heavy-tailed distributions differ from the distributions that have been traditionally used to model request durations in communication networks in a number of significant ways. If the random variable \( X \) in the above functions is interpreted as the waiting time of a request in some queuing system, then for a light-tailed distribution (such as a uniform distribution) of waiting times, the longer the request has waited, the sooner the request is likely to be served. If the waiting times have a medium-tailed distribution (such as the memoryless or exponential distribution), the expected future waiting time of the request is independent of the waiting time so far. For heavy-tailed distributions (such as the Pareto distribution or Weibull distribution), the longer the request has waited, the longer is its expected future waiting time.

For a value of \( a < 1 \), the Pareto distribution has an infinite mean, and if \( 1 < a < 2 \), then the distribution has an infinite variance and a finite mean given by:

\[ \text{mean} = (k \times a)/(a + 1). \]
As $\alpha$ decreases, the portion of the probability mass in the pdf present in the tail of the distribution increases and for real-world scenarios this translates to higher probabilities of having requests with longer and longer durations.

One of the problems in applying this heavy-tailed modeling technique to the request durations and data item sizes generated in this simulation is the choice of a "good $\alpha$ value. In [Fel00], the author has indicated that for typical networking applications, $\alpha$ values in the range of 1.4 to 1.6 are commonly acceptable. In this research, we use the Pareto distribution with an $\alpha$ value of 1.5 whenever a heavy-tailed distribution is needed.

The data type requests are assumed to be between 1 megabyte (MB) to 100 MB in size. The distribution of file sizes is based on a heavy-tailed (Pareto) distribution. The session type requests are assumed to have a minimum size of 500 Kbps and a maximum of 10 Mbps and correspond roughly to the use of a high-quality video stream at the lower end, as in [MiM96], and to typical bandwidths requested in video-on-demand systems, as in [MuS98], at the upper end. The requests are assumed to have a uniform distribution for bandwidths between the minimum and the maximum.

The duration of the session type requests is assumed to be from a minimum of 3 minutes to a maximum of 120 minutes. The probability distribution for the request durations is based on the heavy-tailed (Pareto) distribution. While not being true heavy-tailed distributions (as they are being truncated for practical reasons), the distribution function of the random variables representing the request duration and data file sizes resembles a heavy-tailed distribution.

Hence, for session request durations, the cdf is given by:

$$ F(x) = 1 - (180/x)^{1.5} $$

and the mean duration $= (180 \times 1.510.5) = 540$ seconds. The data type request sizes have a cdf given by:

$$ F(x) = 1 - (10^6/x)^{1.5} $$

and mean data size of 3 MB. The average bandwidth that can be requested by a data type request is given by:

avg. bandwidth = mean size/ mean duration = $(3 \times 8 \times 10^6) / 1315$ bps = 76.2 Kbps.
Table 1: Parameter details chosen for simulation experiments.

<table>
<thead>
<tr>
<th>parameter</th>
<th>minimum</th>
<th>maximum</th>
<th>distribution</th>
<th>mean</th>
</tr>
</thead>
<tbody>
<tr>
<td>session bandwidth</td>
<td>500 Kbps</td>
<td>10 Mbps</td>
<td>uniform</td>
<td>5.25 Mbps</td>
</tr>
<tr>
<td>session duration</td>
<td>3 mins.</td>
<td>120 mins.</td>
<td>heavy-tailed</td>
<td>9 mins.</td>
</tr>
<tr>
<td>data size</td>
<td>1 MB</td>
<td>100 MB</td>
<td>heavy-tailed</td>
<td>3 MB</td>
</tr>
<tr>
<td>data duration (available)</td>
<td>30 secs.</td>
<td>600 secs.</td>
<td>uniform</td>
<td>315 secs.</td>
</tr>
<tr>
<td>lead time (for both types)</td>
<td>2 mins.</td>
<td>120 mins.</td>
<td>uniform</td>
<td>61 mins.</td>
</tr>
</tbody>
</table>

7.4. Priority Weights

As stated earlier, the priority level \( p \) for each request has an associated weight, which provides the worth of that request:

\[
\text{worth} = \omega^{4-p}
\]

For the simulation experiments performed in this work, the mode value \( \omega \) is set at either 2 or 10. Consequently, the requests can have worths of \( (1, 2, 4, 8) \) (corresponding to \( \omega = 2 \)) or \( (1, 10, 100, 1000) \) (corresponding to \( \omega = 10 \)). These values could approximate peace time and war time situations, where the most important priority level is worth twice the second most important level or worth ten times the next most important level, respectively.

7.5. Probability of Preemption

The heuristics proposed in Section 5 utilize a probability-based preemption technique when deciding between preempting request(s) with lower priority and one with higher priority in the same class. This technique, it is hoped, would make the heuristic more "global," i.e., more likely to select a choice of requests that is worth more in the final
analysis and less greedy in operation. For the purposes of the simulation experiments, two different values of the preemption probability have been chosen as 1.0 and 0.90.

Thus, in one set of experiments, the heuristic always preempts a lower priority request in favor of a higher priority one (when both are in the same class) and in another set of experiments, the heuristic preempts a lower priority request with a probability of 0.90 in favor of a higher priority one. The value of 0.90 has been chosen somewhat arbitrarily and in future work it is expected that a more accurate estimation of this probability, based on several factors (such as the relative worth of the preempting request as compared to the request(s) being preempted, the relative probability of completion of the preempting request as compared to the requests being preempted) will be used.

### 7.6. Versions of the Heuristic Simulated

The heuristic proposed in Section 5, is composed of three main operations:
(1) sorting operation – sort requests in each batch by their worth-per-bit values after grouping them together on the basis of class and type;
(2) preemption operation – use preemption as a tool to schedule more important requests that arrive later than less important requests scheduled earlier; and
(3) data reposition operation – use the repositioning of data type requests to schedule session requests of less important classes that would have to be otherwise dropped;

In the simulation experiments, three different versions of the heuristics have been implemented: sort which uses only the sorting routine, sort + preempt which utilizes sorting and preemption, and full which implements the full heuristic (sorting + preemption + data reposition). (For more details on deriving one version of the heuristic from the other, see Appendices 1, 2, and 3.)

These three versions of the heuristic have been simulated for a total of eight different combinations of the three parameters: Loading factor, preemption probability, and the priority weightings as explained in Sections 7.2, 7.4, and 7.5, respectively. These combinations are shown in Table 2. For each of these 24 simulation cases (eight combinations of parameter values x three versions), the sum of the weighted priorities of
the satisfied requests belonging to each class is calculated along with several other quantities of interest. Similar runs of the simulation were also conducted for the complete sharing policy and the upper bounds as explained in Section 6.

Table 2: Values of parameters varied to generate eight different simulation scenarios.

For a given value of one of the parameters, the other two are varied to generate $2^3 = 8$ simulation scenarios.

<table>
<thead>
<tr>
<th>parameters varied</th>
<th>values</th>
</tr>
</thead>
<tbody>
<tr>
<td>loading factor (l.f.)</td>
<td>0.7</td>
</tr>
<tr>
<td>mode value ($\omega$)</td>
<td>2</td>
</tr>
<tr>
<td>preemption probabilities (p.p.)</td>
<td>0.9</td>
</tr>
</tbody>
</table>

All the simulation results were averaged over sufficient number of test cases to obtain a 95% confidence interval such that the results would lie within a ± 5% (or smaller) region of the average. The various results obtained from these experiments are examined in detail in the next chapter.
8. RESULTS OF THE SIMULATION EXPERIMENTS

8.1. Overview

The simulation experiments conducted to examine the performance of the heuristics have been described in Section 7. The results of these experiments and the various trends exhibited are analyzed in this chapter.

As explained in Section 7.6, three versions of the heuristic have been simulated for eight different scenarios obtained by varying the loading factor, the mode value, and the preemption probabilities. For each scenario 20 experiments were conducted. For each of these experiments, a different set of approximately 2000 requests is generated (as described in Section 7.2). The three versions of the heuristic are simulated for this set of requests. The performance of the complete sharing policy and the three upper bounds are averaged over the 20 experiments for each scenario; thus providing seven distinct sets of values for the sum of weighted priorities of satisfied requests in each class for each of the eight simulation scenarios. Of these seven values, the six values corresponding to the complete sharing performance, the two tight upper bounds and the three heuristics have been plotted in Figures 5 to 12 presented in Section 8.2 in this chapter. (The loose upper bound is a very optimistic estimate of the performance and including it in the same figures as the other six results causes the scales involved to be large, thus masking the differences in performance of the other six values to some extent; hence, rather than plotting it in the Figures 5 to 12, its value has been stated in the corresponding text.)

To determine the suitable number of experiments for each scenario required to get a reasonable estimate of the sum of weighted priorities with a confidence interval of 95%, the simulation experiment corresponding to the parameter values of \( \omega = 10 \), preemption probability = 0.9, and loading factor = 1.2 were performed 20, 40, and 60 times. These experiments were conducted only for the full heuristic. The sum of
weighted priorities of satisfied class 1 requests were averaged over the 20, 40, and 60 experiments and their corresponding 95% confidence intervals have been plotted in Figure 4. This scenario is expected to provide the "worst case" variance in the results and any value of repetitions determined for this scenario should be more than sufficient for the other seven scenarios.

A confidence interval of 95% implies that given the calculated sample mean over 20, 40, and 60 cases, the probability that the true mean lies somewhere within the range bars (or error bars) is 0.95. (As is standard practice in communications research, a Student's t distribution is assumed for the calculations of confidence intervals; for more details see [CaS99]).

![Figure 4: The confidence intervals for loading factor = 1.2, \( \omega = 10 \), and preemption probability = 0.9 with the full heuristic (sorting + preemption + reposition).](image)

The confidence interval ranges (at 95% confidence value) are ± 3.47% for 20 experiments, ± 2.05% for 40 experiments, and ± 1.85% for 60 experiments. Note that
even in the worst case, i.e., with 20 runs of the simulation, the 95% confidence interval range is less than $\pm 3.5\%$ of the sample mean. Hence, all further simulation experiments for each of the eight scenarios have been carried out as an average over 20 runs of the simulation with the 95% confidence interval plotted for the class 1 values in each graph.

8.2. Simulation Results for the Eight Scenarios

As indicated in Figures 5 to 12, several trends are apparent in the observed results. The complete sharing performance for class 1 is much below the performance of the sorting, sorting + preemption, and the full heuristic in all eight simulation scenarios, indicating that it is always better to have some sort of an admission control policy for the AICE-like system simulated here, rather than none at all. Also, from comparing Figures 7 and 11 or 8 and 12, it is clear that when the system is more heavily loaded there is more advantage obtained from an intelligent allocation of the bandwidth as compared to a simplistic one. Thus, it may be argued that while a system that is lightly loaded may still function reasonably well with a simple scheduling heuristic such as first-come-first-served mechanism, for systems where overload periods are a common occurrence, the presence of intelligent schedulers is critical. As expected, the complete sharing policy allocates the capacity in the network such that each class contributes to roughly one third of the total sum of weighted priorities achieved by the complete sharing performance. Consequently, it results in poor relative performance in class 1 as compared to the three heuristic variations but higher values for the class 2 and class 3 satisfied worth.

The sorting part of the heuristic orders the requests by class and then by their worth per bit and does better at allocating more class 1 requests and hence obtaining more class 1 worth from the system as compared to the lower bound. For AICE-like systems where preemption is not a practical alternative, the sorting routines may serve the purpose of a simple bandwidth allocation policy.
Figure 5: The relative performance for loading factor = 0.7, $\omega = 2$, and preemption probability = 1.0 for complete sharing policy, upper bounds, and the three variations of the heuristic averaged over 20 experiments. Note: C.S. = complete sharing performance, ingress = ingress upper bound, egress = egress upper bound and C1 = class 1, C2 = class 2, C3 = class 3. The loose upper bound for the simulation experiment averaged over 20 experiments is 7633.
Figure 6: The relative performance for loading factor = 0.7, $\omega = 2$, and preemption probability = 0.9 for complete sharing policy, upper bounds, and the three variations of the heuristic averaged over 20 experiments. Note: c.s. = complete sharing performance, ingress = ingress upper bound, egress = egress upper bound and $C_1$ = class 1, $C_2$ = class 2, $C_3$ = class 3. The loose upper bound for the simulation experiment averaged over 20 experiments is 7603.
Figure 7: The relative performance for loading factor = 0.7, \( \omega = 10 \), and preemption probability = 1.0 for complete sharing policy, upper bounds, and the three variations of the heuristic averaged over 20 experiments. Note: C.S. = complete sharing performance, ingress = ingress upper bound, egress = egress upper bound and C1 = class 1, C2 = class 2, C3 = class 3. The loose upper bound for the simulation experiment averaged over 20 experiments is 583277.
Figure 8: The relative performance for loading factor $= 0.7$, $\omega = 10$, and preemption probability $= 0.9$ for complete sharing policy, upper bounds, and the three variations of the heuristic averaged over 20 experiments. Note: c.s. = complete sharing performance, ingress = ingress upper bound, egress = egress upper bound and C1 = class 1, C2 = class 2, C3 = class 3. The loose upper bound averaged over 20 experiments is 576222.
Figure 9: The relative performance for loading factor $= 1.2$, $\omega = 2$, and preemption probability $= 1.0$ for complete sharing policy, upper bounds, and the three variations of the heuristic averaged over 20 experiments. Note: C.S. = complete sharing performance, ingress = ingress upper bound, egress = egress upper bound and C1 = class 1, C2 = class 2, C3 = class 3. The loose upper bound averaged over 20 experiments is 7610.
Figure 10: The relative performance for loading factor = 1.2, $\omega = 2$, and preemption probability = 0.9 for complete sharing policy, upper bounds, and the three variations of the heuristic averaged over 20 experiments. Note: C.S. = complete sharing performance, ingress = ingress upper bound, egress = egress upper bound and C1 = class 1, C2 = class 2, C3 = class 3. The loose upper bound averaged over 20 experiments is 7589.
Figure 11: The relative performance for loading factor $= 1.2$, $\omega = 10$, and preemption probability $= 1.0$ for complete sharing policy, upper bounds, and the three variations of the heuristic averaged over 20 experiments. Note: c.s. = complete sharing performance, ingress = ingress upper bound, egress = egress upper bound and C1 = class 1, C2 = class 2, C3 = class 3. The loose upper bound averaged over 20 experiments is 569298.
Figure 12: The relative performance for loading factor = 1.2, \( \omega = 10 \), and preemption probability = 0.9 for complete sharing policy, upper bounds, and the three variations of the heuristic averaged over 20 experiments. Note: c.s. = complete sharing performance, ingress = ingress upper bound, egress = egress upper bound and C1 = class 1, C2 = class 2, C3 = class 3. The loose upper bound averaged over 20 experiments was 578567.
The sorting + preemption variation of the heuristic demonstrates the uses of preemption as a tool to improve on-line performance. The chief difference between the sorting and the sorting + preemption heuristics is the ability of the latter to schedule more valuable requests that arrive later in place of less valuable ones that got scheduled earlier. With the use of preemption, the Figures 5 to 12 indicate that, especially at light loads, it is possible to come very close to the ingress and egress upper bounds; in terms of class 1 performance.

The performance of the full heuristic does not vary significantly for class 1 from the performance of the sorting + preemption version, as was expected. The data reposition routines in the full version of the heuristic are called in aid of classes 2 and 3 only and hence do not affect the performance of class 1 significantly (they do cause more preemption in class 1, as is observed in Table 3 given in Section 8.3). The real difference is in the slight performance improvement for class 2 and class 3 values of the sum of the weighted priorities. Hence, if the extra overhead imposed by the data repositioning routines is acceptable then the full heuristic may be able to get good performance for class 1 and improve the class 2 and 3 performance as well.

As indicated by a comparison of the Figures 5 to 12, the overall performance of all the three versions of the heuristic remains relatively the same for the two mode values simulated of \( \omega = 2 \) and \( \omega = 10 \). More about the differences between those two mode values is given in Section 8.3.

A comparison of Figures 7 and 8 etc, where only the preemption probability is varied, indicates that contrary to expectations, the presence of the preemption probability factor does not cause a significant improvement in performance and in fact may cause a slight reduction in performance in certain cases. (Note that 0.9 was the value for the preemption probability that performed the best amongst the values of 0.75, 0.8, 0.9, and 0.95 that were tried out in some sample experiments conducted for one of the eight simulation scenarios). This result may have one of several causes. It may be possible that an incorrect value of the factor was chosen and hence the reduction in performance. In future work, where this factor is expected to be linked to several other conditions that exist at the time of preemption (such as the relative worths of the preempted and
preempting requests, the relative probabilities of preemption of the requests), this factor might very well yield the improvement in performance expected.

In the next subsection, the results regarding the actual number of class 1 requests allocated, listed by the four priority levels, and the number of preemptions and data repositions for the various cases are presented.

8.3. Priority Level and Number of Preemption Related Results

As indicated from Figures 13 and 14, for \( \omega = 2 \) the number of priority 1 requests accepted is slightly less than the number of priority 1 requests accepted in the \( \omega = 10 \) case for the heavily loaded cases. This is to be expected, as a priority 1 request in the \( \omega = 10 \) case corresponds to a worth of 1000 while a similar request in the \( \omega = 2 \) case corresponds to only a worth of 8. Consequently, more number of priority 4 requests get accepted in the \( \omega = 2 \) case than in the \( \omega = 10 \) case. When the loading is not heavy, as in Figures 15 and 16, note that, almost all the class 1 requests of all priorities tend to be satisfied and hence these trends are not visible in these figures. For class 3 requests, where the competition for bandwidth is much higher (because of bandwidth used by class 1 and class 2 requests), from sample data it can be observed that mostly priority 1 and 2 requests get selected for execution in the \( \omega = 10 \) case whereas for \( \omega = 2 \) requests in all four priority levels tend to get selected. (For the heavy loading case with preemption probability = 1.0 it was observed from sample data that for class 3, \( \omega = 10 \) accepted nearly 54% more priority 1 requests as compared to \( \omega = 2 \).)

Table 3 indicates the average number of requests preempted for the sorting + preemption and the full version of the heuristic and the number of data type requests repositioned for the full heuristic. Recall that the reposition routines can reposition only requests of either class 1 or class 2, and hence only these two classes are indicated. Also, the preemptions are counted such that any time a request gets preempted, the count is incremented by one.
Figure 13: The number of class 1 requests of each priority level (1, 2, 3, and 4). This is for the scenario with loading factor $= 1.2$, and preemption probability $= 0.9$ and for $\omega = 10$ and $\omega = 2$. Each number is averaged over 20 simulation experiments.
Figure 14: The number of class 1 requests of each priority level (1, 2, 3, and 4). This is for the scenario with loading factor = 1.2, and preemption probability = 1.0 and for $\omega = 10$ and $\omega = 2$. Each number is averaged over 20 simulation experiments.
Figure 15: The number of class 1 requests of each priority level (1, 2, 3, and 4). This is for the scenario with loading factor $= 0.7$, and preemption probability $= 0.9$ and for $\omega = 10$ and $\omega = 2$. Each number is averaged over 20 simulation experiments.
Figure 16: The number of class 1 requests of each priority level (1, 2, 3, and 4). This is for the scenario with loading factor $= 0.7$, and preemption probability $= 1.0$ and for $\omega = 10$ and $\omega = 2$. Each number is averaged over 20 simulation experiments.
Table 3: Average number of preemptions by class for the sort + preempt version and the full versions of the heuristic for different values of the loading factor (l.f.), $\omega$, and preemption probability (p.p.), each averaged over 20 simulation runs.

<table>
<thead>
<tr>
<th>l.f. = 1.2</th>
<th>sort + preempt</th>
<th>full</th>
<th>data repositions</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\omega = 10$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>p.p. = 0.9</td>
<td>67</td>
<td>98</td>
<td>62</td>
</tr>
<tr>
<td></td>
<td>226</td>
<td>246</td>
<td>56</td>
</tr>
<tr>
<td></td>
<td>187</td>
<td>232</td>
<td></td>
</tr>
<tr>
<td>l.f. = 1.2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\omega = 10$</td>
<td>81</td>
<td>109</td>
<td>58</td>
</tr>
<tr>
<td>p.p. = 1.0</td>
<td>255</td>
<td>290</td>
<td>63</td>
</tr>
<tr>
<td></td>
<td>231</td>
<td>253</td>
<td></td>
</tr>
<tr>
<td>l.f. = 1.2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\omega = 2$</td>
<td>55</td>
<td>89</td>
<td>69</td>
</tr>
<tr>
<td>p.p. = 0.9</td>
<td>197</td>
<td>200</td>
<td>47</td>
</tr>
<tr>
<td></td>
<td>187</td>
<td>221</td>
<td></td>
</tr>
<tr>
<td>l.f. = 1.2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\omega = 2$</td>
<td>57</td>
<td>88</td>
<td>59</td>
</tr>
<tr>
<td>p.p. = 1.0</td>
<td>216</td>
<td>225</td>
<td>42</td>
</tr>
<tr>
<td></td>
<td>173</td>
<td>198</td>
<td></td>
</tr>
<tr>
<td>l.f. = 0.7</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\omega = 10$</td>
<td>6</td>
<td>14</td>
<td>49</td>
</tr>
<tr>
<td>p.p. = 0.9</td>
<td>208</td>
<td>225</td>
<td>37</td>
</tr>
<tr>
<td></td>
<td>154</td>
<td>146</td>
<td></td>
</tr>
<tr>
<td>l.f. = 0.7</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\omega = 10$</td>
<td>9</td>
<td>15</td>
<td>43</td>
</tr>
<tr>
<td>p.p. = 1.0</td>
<td>252</td>
<td>271</td>
<td>39</td>
</tr>
<tr>
<td></td>
<td>198</td>
<td>207</td>
<td></td>
</tr>
<tr>
<td>l.f. = 0.7</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\omega = 2$</td>
<td>3</td>
<td>6</td>
<td>42</td>
</tr>
<tr>
<td>p.p. = 0.9</td>
<td>189</td>
<td>201</td>
<td>32</td>
</tr>
<tr>
<td></td>
<td>179</td>
<td>209</td>
<td></td>
</tr>
<tr>
<td>l.f. = 0.7</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\omega = 2$</td>
<td>4</td>
<td>9</td>
<td>37</td>
</tr>
<tr>
<td>p.p. = 1.0</td>
<td>252</td>
<td>264</td>
<td>22</td>
</tr>
<tr>
<td></td>
<td>197</td>
<td>203</td>
<td></td>
</tr>
</tbody>
</table>
This may imply that the same request may be scheduled and then preempted multiple number times thus contriiting to the average number of preemptions multiple times. A similar argument holds for the Figures indicated in the data reposition columns also.

The table indicates that, as expected, the number of preemptions increase from the preemption probability = 0.9 to preemption probability = 1.0 case. One result that may appear surprising is the fact that class 2 requests show, on average, more preemption than class 3 requests, while in fact the expected result may be that the number of preemptions should increase monotonically from class 1 to class 3.

One possible reason that the class 2 requests show more preemptions is that, as indicated from the performance values in Figures 5 to 12, not many class 3 requests may get scheduled in the first place while many more class 2 requests may be scheduled. Hence, while picking candidates for preemption, more class 2 requests may get selected, even though class 3 requests are the first to be picked, simply because more class 2 requests are scheduled in the first place. While satisfying requests, an exact opposite policy is followed and class 3 requests get picked last for scheduling. This might explain the relatively larger class 2 preemption values. (A sample examination for one of the loading factor = 1.2, $\omega = 10$ experiments, indicates that many more class 3 requests simply get dropped without being scheduled because of conflicts as compared to class 2 requests, which get scheduled and then preempted.)

A final result in Table 3 is the increased number of preemptions in class 1 when going from the sorting + preemption variation to the full heuristic. One possible explanation for this increase in the class 1 preemptions, which is provided by an examination of the sample experiments, is that when the class 1 data type requests get repositioned by the heuristic, the probability that they will be picked for preemption increases substantially. The heuristic, while making preemption decisions amongst requests of the same class and the same priority, tends to always pick requests that occupy more bandwidth. Also, the repositioning tends to allocate large bandwidths to the requests to compensate for the small time duration available in which to transfer the data. This implies that for scheduling a session type request of class 1, this repositioned request might have to be preempted as it is blocking a large portion of link bandwidth.
(for a small amount of time). (A sample experiment indicates that for the loading level of 1.2, preemption probability of 1.0, and $\omega = 10$, about 27% of the repositioned class 1 requests subsequently got preempted.)

In spite of the fact that the full variation of the heuristic causes more preemption in class 1, an observation of Figures 5 to 12 indicates that the class 1 performance of the full version is the same as that of the sort + preempt version of the heuristic. The reason for this is that more preemption does not necessarily imply greater loss of worth. Recall from Section 5, that a request that gets preempted might still get scheduled later, provided its start time (or deadline) has not expired. So, even though more class 1 requests show preemption in the full variation, most of these requests do get scheduled later by preempting requests of classes 2 and/or 3. This indicates that though the performance of the sort + preempt and full variations is the same for class 1, the two variations achieve this similar performance by the use of different scheduling mechanisms.

8.4. Execution Times

Some sample execution times are presented in this subsection to provide an approximate idea of the relative differences in execution time between the three heuristic variations. For the loading factor = 1.2, preemption probability = 1.0, and $\omega = 10$ scenario, sample execution times with the current implementation were: sorting = 11 seconds per batch, sorting + preempt = 29 seconds per batch, and full heuristic = 47 seconds per batch. The computing environment was an Intel Pentium II workstation running Windows – NT 4.0. (The simulation code was not optimized to provide fast execution and a better implementation and/or a faster computer environment might provide substantially different results.)

As stated in Section 3.2, the maximum time between two scheduling events is 60 seconds and the minimum lead time (time between arrival and start (or available)) for any request is 120 seconds. With these timing parameters, it is unlikely that requests will
be missed because the heuristic taking too long to execute causes its starting time to expire before it is processed.

8.5. Summary

The results presented in this chapter indicate that the proposed heuristics outperform the complete sharing policy in all simulation scenarios described. The results also indicate that while significant gain in performance over the complete sharing policy is possible in the 0.7 load case, the real performance difference is apparent at the higher loading level where the performance gain is in the range of a factor of three over the complete sharing policy. This serves to illustrate the fact that an intelligent bandwidth allocation mechanism can greatly improve the performance of an overloaded system. The sorting + preemption version of the heuristic provides a good class 1 performance while the full heuristic, including the data item repositioning routines, helps to improve the performance of class 2 and class 3 requests in addition to maintaining comparable class 1 performance. However, the full heuristic takes approximately 50% more time to execute with the current implementation.
The problem of bandwidth allocation in a preemptive communication network has been examined in detail in this work. The intelligent allocation of bandwidth is an important tool in being able to support today's multimedia driven networking requirements. With increasing demand for providing QoS to customers, there has been considerable effort devoted to mechanisms that are capable of differentiated handling of requests according to differing needs.

An innovative grouping mechanism was utilized in this work to introduce the concepts of both differentiated handling (class) and prioritized worths (priority level within a class) in a unified manner. A heuristic was proposed to perform scheduling of both session and data type requests in hierarchical preemptive networks. Simulation experiments were conducted to quantify the performance of the heuristic under various scenarios. Three different parameters, the loading factor, the mode value ($\omega$), and the probability of preemption, were each varied between two values for a total of eight different simulation scenarios. Three different versions of the heuristic, from a simple sorting based version to one that used sorting, preemption, and data repositioning were each tested for eight different scenarios and their performance compared with the upper bounds and complete sharing policy. The performance measure used was the sum of weighted priorities of satisfied requests on a class by class basis.

The results indicate that the proposed heuristics outperform the complete sharing policy and, in most cases, approach close to the upper bounds in term of class 1 performance. The full heuristic shows the best performance for both class 1 and the lower classes while the version not implementing data repositioning shows good class 1 performance but relatively lower class 2 and class 3 performance. Hence, depending on
the processing overhead tolerable in the system, either version could be used to maximize class 1 performance.

Future work in this area could include several possibilities. There may be several different ways to schedule data type requests instead of scheduling them with the minimum possible bandwidth as is done here. Similarly, there may be other interesting trade-offs between scheduling of session and data type requests that may be used to improve the performance of the scheduling heuristics as a whole. The preemption probability factor has been somewhat arbitrarily chosen in this work to be a constant of 0.9. This factor actually depends on a lot of parameters associated with the preemting request and the preempted request(s) and could in fact vary with each preemption. Hence, using a preemption probability derived in this manner (instead of using a constant factor) could result in a further improvement in the performance of the heuristic presented here. Similarly, there may be other approaches to make the heuristics less greedy and more "global" in nature. In this work, the requests were uniformly distributed among three classes and four priorities. It might be interesting to examine the performance of the heuristic for scenarios in which the request distribution was different in each of the three classes or four priorities (as in [ThB00]) or when there were more than three classes.

Modifications to the heuristic, such as introduction of a "threshold" on the utilization of the link by class 1 requests, may improve the performance of class 2 and class 3 while not significantly degrading the performance of class 1 requests. Other methods of deriving the priority weightings from the priority levels may be considered in addition to the one proposed in this work. It may also be possible to consider a maximization of performance when the performance measure includes several different factors in addition to the worth of the requests.

There may also be future directions for this work that involve a broadening of the basic assumptions. The system model may allow requests to be degraded before they are allocated, i.e., there may be a system that allows fractional gain in worth for providing only a portion of the requested resource. For such a system, the work presented here has to be extended/modified to incorporate this degradation. Another possible direction is
the consideration of an additional QoS parameter, such as loss rate or latency or jitter in addition to bandwidth.
LIST OF REFERENCES


APPENDIX 1: PSEUDO CODE FOR FULL HEURISTIC

function main()
1. assume class \( i = \{1,2,3\} \), priority \( j \) within a class \( = \{1,2,3,4\} \)
2. for each batch during simulation period
3. group requests in batch by class
4. for \( (i = 1 \) to \( i = 3\) ), group all requests in class \( j \) by type (session or data)
5. for \( (i = 1 \) to \( i = 3\) ), order all session type requests in class \( i \) by worth per bit
6. for \( (i = 1 \) to \( i = 3\) ), order all data type requests in class \( i \) by worth per bit

/* main scheduler */
7. consider each request \( R \) in current batch in the sorted order
8. if (type—of—\( R \) = session)
9. if \( \{(\text{check} \_\text{used} \_\text{bw}(R\text{gress}, R\text{start} \_\text{time}, R\text{finish} \_\text{time}) < \text{link} \_\text{bw} - R\text{bw}) \)
and \( (\text{check} \_\text{used} \_\text{bw}(R\text{gress}, R\text{start} \_\text{time}, R\text{finish} \_\text{time}) < \text{link} \_\text{bw} - R\text{bw})\} \)
10. change \_link \_status(\( R \)) /* schedule request \( R \) */
11. else
12. call main_session_scheduler(\( R \), flag = true)
13. if (type—of—\( R \) = data)
14. if \( ((\text{check} \text{g} \_\text{os} \_\text{s} \_\text{i} \_\text{ble} \_\text{data} \_\text{schedule}(R, R\text{available} \_\text{time}, \text{headline})) = \text{false}) \)
15. call main_data_scheduler(\( R \), \( R\text{available} \_\text{time} \), headline, \( R\text{class} \))
16. end
17. end
function check_used_bw(link L, time Rstart_time, time Rfinish_time)
1. for all scheduled requests T at L
2. if (Tstart_time ≤ Rfinish_time and Tfinish_time ≥ Rstart_time)
3. add T to active_requests_list
4. if (size of active_requests_list > 0)
5. for all requests in active_requests_list
6. find the total bandwidths used by the requests at each point in time
    between Rstart_time and Rfinish_time
7. max_used_bw =
    max(total_used_bandwidths between Rstart_time and Rfinish_time)
8. return max_used_bw
9. else
10. return 0
11. end

function change_link_status(request R)
1. add Rbw for Rstart_time to Rfinish_time into allocated_request_list for link R ingress
2. add Rbw for Rstart_time to Rfinish_time into allocated_request_list for link R egress

function main_data_scheduler
(request R, time Rbegin_time, time Rend_time, int Rclass_value)
1. set Rstart_time = Rbegin_time
2. set Rfinish_time = Rend_time
3. set Rclass = Rclass_value
4.. set Rbw = Rdata_size / (Rend_time - Rbegin_time)
   /* Request R is scheduled as a session type request */
5. call main_session_scheduler(R, flag = false)
6.. end
function check_possible_data_schedule
(request R, time R_begin_time, R_end_time)

1. find the total occupied bandwidth for each point in time between $R_{\text{begin\_time}}$ and $R_{\text{end\_time}}$ for link $R_{\text{ingress}}$
2. find the total occupied bandwidth for each point in time between $R_{\text{begin\_time}}$ and $R_{\text{end\_time}}$ for link $R_{\text{egress}}$
3. take the maximum of the bandwidths at each point in time, found in step 1 and 2 to create unified resource picture for R
4. build table resource\_status\_unified for request R, for time period between $R_{\text{begin\_time}}$ and $R_{\text{end\_time}}$ by calculating the unused bandwidth and the period for which it is available from the unified resource picture obtained in step 3 (see 'Table 4')
5. sort table resource\_status\_unified in descending order by duration
6. for (i = 1 to i = number of rows in table), consider each row in sorted order
7. if ($(\text{duration}(i) * \text{bw}(i)) \leq R_{\text{data\_size}}$)
8. $R_{\text{start\_time}}$ = starting time in time instants column of row i
9. $R_{\text{finish\_time}}$ = ending time in time instants column of row i
10. $R_{\text{bw}} = R_{\text{data\_size}} / (R_{\text{finish\_time}} - R_{\text{start\_time}})$
11. change_link_status(R) /* schedule R */
12. return true
13. else
14. consider next row in table in sorted order
15. end i loop
16. no suitable unused capacity can be allocated to data request
17. return false
<table>
<thead>
<tr>
<th>time Instants</th>
<th>duration</th>
<th>bandwidth</th>
<th>order</th>
</tr>
</thead>
<tbody>
<tr>
<td>3 - 15</td>
<td>12</td>
<td>45</td>
<td>5</td>
</tr>
<tr>
<td>5 - 20</td>
<td>15</td>
<td>20</td>
<td>2</td>
</tr>
<tr>
<td>6 - 19</td>
<td>13</td>
<td>40</td>
<td>4</td>
</tr>
<tr>
<td>3 - 19</td>
<td>16</td>
<td>20</td>
<td>1</td>
</tr>
<tr>
<td>6 - 20</td>
<td>14</td>
<td>30</td>
<td>3</td>
</tr>
<tr>
<td>7 - 19</td>
<td>12</td>
<td>50</td>
<td>5</td>
</tr>
</tbody>
</table>

**Table 4:** Example of a resource_status_unified table depicting the minimum of the unused bandwidths available at each point in time at the ingress and egress links of request R. The order values are based on duration values.
function main_session_scheduler
(request R, flag data—reposition—allowed)
1. for all scheduled requests T at R_{\text{ingress}}
2. if (T_{\text{start time}} \leq R_{\text{finish time}} and T_{\text{finish time}} \geq R_{\text{start time}})
3. add T to active_requests_list_{\text{ingress}}
4. max_used_bw_{\text{higher session}} = \max \text{ of used bandwidth for all requests T in active_requests_list_{\text{ingress}} where class_of_request(T) < R_{\text{class}} and type_of_request(T) = session}
5. if (max_used_bw_{\text{higher session}} + R_{bw} > \text{link–bw})
6. add request R to unscheduled_requests_list
7. return /* request R conflicts with session requests of higher class at ingress link */
8. else
9. build active_requests_list_{\text{egress}} for all scheduled requests Q at R_{\text{egress}};
repeat steps 4–7 on this list;
/* R does not conflict with session requests < R_{\text{class}} at ingress and egress; does R conflict with data requests < R_{\text{class}} */
10. max_used_bandwidth_{\text{ingress}} = \max \text{ of bandwidths used by requests T in active–request–list–ingress where class_of_request(T) < R_{\text{class}}}
11. max_used_bandwidth_{\text{egress}} = \max \text{ of bandwidths used by requests Q in active–request–list–egress where class_of_request(Q) < R_{\text{class}}}
12. if ((max_used_bandwidth_{\text{ingress}} + R_{bw} \leq \text{link–bw}) and
(max_used_bandwidth_{\text{egress}} + R_{bw} \leq \text{link–bw}))
/* no data repositioning required; request R may be blocked by requests of either same class or a less important class, so may need preemption for scheduling; handled by preempt routine */
13. call preemptRoutine(R)
14. else if (data–reposition–allowed = false) /* R is originally of data type */
15. add request R to unscheduled_requests_list
16. else
/* attempt two ingress, egress or ingress + egress data repositioning */
17. if \( (\text{max\_used\_bw\_ingress} + \text{R}_\text{bw} > \text{link\_bw} ) \) and
   \( (\text{max\_used\_bandwidth\_egress} + \text{R}_\text{bw} \leq \text{link\_bw}) \)
   /* data reposition required at ingress only */
18. call data-reposition (R, active-request-list-ingress);
19. if repositioning successful, recalculate \( \text{max\_used\_bw\_ingress} \);
   if \( (\text{max\_used\_bw\_ingress} + \text{R}_\text{bw} > \text{link\_bw} ) \), call data-reposition (R,
   active-request-list-ingress) again
20. if even one reposition in 18 or 19 fails /* reposition failed at ingress */
21. add request R to unscheduled-requests-list
22. else
23. call preempt_routine(R)
24. else if \( (\text{max\_used\_bw\_egress} + \text{R}_\text{bw} > \text{link\_bw} ) \) and
   \( (\text{max\_used\_bandwidth\_ingress} + \text{R}_\text{bw} \leq \text{link\_bw}) \)
   /* data reposition required at egress only */
25. repeat steps 18 to 23 for active-request-list-egress
26. else
   /* reposition may be required at both ingress and egress */
27. call data-reposition (R, active-request-list-ingress)
28. call data-reposition (R, active-request-list-egress)
29. if even one repositioning in 27 and 28 fails
30. add request R to unscheduled-requests-list
31. else
32. call preempt_routine(R)
   /* request R may still be blocked with requests of either same
   class or a less important class, so may need preemption for
   scheduling; handled by preempt routine */
function preempt_routine(request R)
1. get active-request-list for requests at link = R ingress
2. order active-request-list by descending values of class
3. within each class, sort the requests by descending values of priority
4. if multiple requests of same priority within a class, then sort by descending values of request bandwidth.
5. while (bw-preempted < R_bw - unused bandwidth on R ingress)
6. for (class = 3 to class = R_class)
7. insert into preempt-candidate-list request T where
   class_of_request(T) > R_class, or if class_of_request(T) = R_class then
   priority_of_request(T) > R priority
8. for each such request added, bw_preempted = bw_preempted + T_bw
9. end for loop
10. end while loop
11. repeat steps 1-10 for active-request-list at link = egress, consider the
    bandwidth of any request, already in preempt-candidate-list that shares
    R_egress as its egress link, as preempted bandwidth
12. worth_preempted = sum of weighted priorities of requests in
    preempt-candidate-list where class-of-request = R_class
13. if (worth_preempted < R_weighted_priority)
    /* steps 12-13 not executed if preemption probability = 1.0 */
14. generate random number rand where(0 < rand < 1)
15. if (rand > 0.10) /*same class preemption probability = 0.90 */
16. call preempt_request(R, preempt-candidate-list) /* schedule R */
17. else
18. add request R to unscheduled_requests_list
19. else P current request weighted priority is less than requests preempted*/
20. add request R to unscheduled-requests-list
21. end
function preempt_request(request R, list preempt–candidates–list)
1. for all requests in preempt–candidates–list
2. delete request allocations from ingress and egress links
3. if (type–of–request = session)
4. if (start time of request < current time)
5. drop request completely from system
6. else
7. add request to preempted–requests–list
8. else /* request is of data type */
9. if {(latest time of request < current time) or
   (size of data item requested ≥ (155Mbps * (deadline – current time)))}
10. drop request completely from system
11. else
12. add request to preempted–requests–list
13. change_link_status(R) /* schedule R */
   /* allocate the left over capacity after preemption */
14. select requests from unscheduled_requests_list where either ingress and/or
   egress links matches any of the ingress and/or egress links in the
   preempt–candidates–list and call main scheduler for each of these requests
15. merge preempted–requests–list into unscheduled_requests_list
16. end
function data_reposition(request R, list active-request-list-L)
1. select request T from active-request-list-L where (type-of-T = data) and
   bandwidth_of_T = max(bws of all data type requests of class < R_class)
   available-time-of-T > current-time such that
   bandwidth occupied by T + unused bandwidth ≥ R_bw
2. if {{T_available_time ≥ R_start_time and T_deadline ≤ R_finish_time) or (no such request)}
   /* data request duration ≥ session request duration or no repositionable data
   type request of correct size in active-request-list-L */
3. data_reposition = false
4. else if (T_available_time ≥ R_start_time and T_deadline > R_finish_time)
5.   T_new_available_time = R_finish_time
6.   if (check_possible_data_schedule(T, T_new_available_time, T_deadline) = false)
7.     T_new_class = R_class + 1
8.   call main_data_scheduler(T, T_new_available_time, T_deadline, T_new_class)
9. else if (T_available_time < R_start_time and T_deadline ≤ R_finish_time)
10.  T_new_deadline = R_start_time
11.  if (check_possible_data_schedule(T, T_available_time, T_new_deadline) = false)
12.    T_new_class = R_class + 1
13.   call main_data_scheduler(T, T_available_time, T_new_deadline, T_new_class)
14. else if (T_available_time < R_start_time and T_deadline > R_finish_time)
15.  if ((R_start_time - T_available_time) > (T_deadline - R_finish_time))
16.    T_new_deadline = R_start_time
17.    if (check_possible_data_schedule(T, T_available_time, T_new_deadline) = false)
18.      T_new_class = R_class + 1
19.    call main_data_scheduler(T, T_available_time, T_new_deadline, T_new_class)
20. else
21.    T_new_available_time = R_finish_time
22.    if (check_possible_data_schedule(T, T_new_available_time, T_deadline) = false)
23.      T_new_class = R_class + 1
24.    call main_data_scheduler(T, T_new_available_time, T_deadline, T_new_class)
APPENDIX 2: PSEUDO CODE FOR SORT + PREEMPT

Same as Appendix 1 except for line 12 in function main()
change line 12 from:
12. call main_session_scheduler(R, flag = true)

to

12. call main_session_scheduler(R, flag = false)

/* changing the flag status to false causes the data repositioning routines in
the heuristics to be suppressed, i.e., not called, hence, the heuristic operates
only with sorting and preemption */
APPENDIX 3: PSEUDO CODE FOR SORT

Same as Appendix 1 except for function main(). Change function main() to one below.

function main()
1. assume class $i = \{1,2,3\}$, priority $j$ within a class $= \{1,2,3,4\}$
2. for each batch during simulation period
3. group requests in batch by class
4. for (i = 1 to i = 3), group all requests in class $i$ by type (session or data)
5. for (i = 1 to i = 3), order all session type requests in class $i$ by worth per bit
6. for (i = 1 to i = 3), order all data type requests in class $i$ by worth per bit
7. consider each request $R$ in current batch in the sorted order
8. if (type $R =$ session)
9. if $\{$(check_used_bw($R_{\text{ingress}}, R_{\text{start\_time}}, R_{\text{finish\_time}}) < \text{link\_bw} - R_{\text{bw}})$
        \text{ and } $(\text{check\_used\_bw}(R_{\text{egress}}, R_{\text{start\_time}}, R_{\text{finish\_time}}) < \text{link\_bw} - R_{\text{bw}}))}$
10. change_link_status($R$) /* schedule request $R$ */
11. else /* $R$ is of type data*/
12. check_possible_data_schedule($R, R_{\text{available\_time}}, \text{headline}$)
13. end
14. end