ADVANCE RESERVATION AND SCHEDULING OF BULK FILE TRANSFERS IN E-SCIENCE

By

KANNAN RAJAH

A THESIS PRESENTED TO THE GRADUATE SCHOOL OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF MASTER OF SCIENCE

UNIVERSITY OF FLORIDA

2007
© 2007 Kannan Rajah
To my Mom and Dad
ACKNOWLEDGMENTS

I would like to express my sincere gratitude to my advisors Dr. Sanjay Ranka and Dr. Ye Xia for their continuous support and encouragement throughout my research work. I am thankful to Dr. Sartaj Sahni for being a vital member of my thesis committee and providing valuable comments on my thesis. I would also like to thank Dr. Rick Cavanaugh and Dr. Paul Avery from the Physics department for several discussions on the Ultralight project.
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>INTRODUCTION</td>
<td>12</td>
</tr>
<tr>
<td>1.1</td>
<td>Related Work</td>
<td>16</td>
</tr>
<tr>
<td>2</td>
<td>CONCURRENT FILE TRANSFER PROBLEM</td>
<td>19</td>
</tr>
<tr>
<td>2.1</td>
<td>Problem Definition</td>
<td>19</td>
</tr>
<tr>
<td>2.2</td>
<td>The Time Slice Structure</td>
<td>20</td>
</tr>
<tr>
<td>2.3</td>
<td>Node-Arc Form</td>
<td>22</td>
</tr>
<tr>
<td>2.4</td>
<td>Edge-Path Form</td>
<td>25</td>
</tr>
<tr>
<td>2.4.1</td>
<td>Shortest Paths</td>
<td>26</td>
</tr>
<tr>
<td>2.4.2</td>
<td>Shortest Disjoint Paths</td>
<td>29</td>
</tr>
<tr>
<td>2.5</td>
<td>Evaluation</td>
<td>30</td>
</tr>
<tr>
<td>2.5.1</td>
<td>Single Slice Scheduling (SSS)</td>
<td>31</td>
</tr>
<tr>
<td>2.5.1.1</td>
<td>Performance comparison of the formulations</td>
<td>32</td>
</tr>
<tr>
<td>2.5.1.2</td>
<td>Comparison of algorithm execution time</td>
<td>33</td>
</tr>
<tr>
<td>2.5.1.3</td>
<td>Algorithm scalability with network size</td>
<td>35</td>
</tr>
<tr>
<td>2.5.1.4</td>
<td>Average results over random network instances</td>
<td>36</td>
</tr>
<tr>
<td>2.5.2</td>
<td>Multiple Slice Scheduling (MSS)</td>
<td>38</td>
</tr>
<tr>
<td>2.5.2.1</td>
<td>Performance comparison of different formulations</td>
<td>38</td>
</tr>
<tr>
<td>2.5.2.2</td>
<td>Comparison of algorithm execution time</td>
<td>40</td>
</tr>
<tr>
<td>2.5.2.3</td>
<td>Optimal time slice</td>
<td>40</td>
</tr>
<tr>
<td>3</td>
<td>ADMISSION CONTROL AND SCHEDULING ALGORITHM</td>
<td>42</td>
</tr>
<tr>
<td>3.1</td>
<td>The Setup</td>
<td>42</td>
</tr>
<tr>
<td>3.2</td>
<td>The Time Slice Structure</td>
<td>43</td>
</tr>
<tr>
<td>3.3</td>
<td>Admission Control</td>
<td>47</td>
</tr>
<tr>
<td>3.4</td>
<td>Scheduling Algorithm</td>
<td>50</td>
</tr>
<tr>
<td>3.5</td>
<td>Putting It Together: The AC and Scheduling Algorithm</td>
<td>51</td>
</tr>
<tr>
<td>3.6</td>
<td>Non-uniform Slice Structure</td>
<td>52</td>
</tr>
<tr>
<td>3.6.1</td>
<td>Nested Slice Structure</td>
<td>53</td>
</tr>
<tr>
<td>3.6.2</td>
<td>Variant of Nested Slice Structure</td>
<td>56</td>
</tr>
<tr>
<td>3.7</td>
<td>Evaluation</td>
<td>57</td>
</tr>
<tr>
<td>3.7.1</td>
<td>Comparison of Algorithm Execution Time</td>
<td>60</td>
</tr>
<tr>
<td>Section</td>
<td>Page</td>
<td></td>
</tr>
<tr>
<td>----------------------------------------------</td>
<td>------</td>
<td></td>
</tr>
<tr>
<td>3.7.2 Performance Comparison of the Algorithms</td>
<td>62</td>
<td></td>
</tr>
<tr>
<td>3.7.3 Single vs Multi-path Scheme</td>
<td>64</td>
<td></td>
</tr>
<tr>
<td>3.7.4 Comparison with Typical AC/Scheduling Algorithm</td>
<td>66</td>
<td></td>
</tr>
<tr>
<td>3.7.5 Scalability of AC/Scheduling Algorithm</td>
<td>67</td>
<td></td>
</tr>
<tr>
<td>4 CONCLUSION</td>
<td>70</td>
<td></td>
</tr>
<tr>
<td>REFERENCES</td>
<td>73</td>
<td></td>
</tr>
<tr>
<td>BIOGRAPHICAL SKETCH</td>
<td>77</td>
<td></td>
</tr>
</tbody>
</table>
LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3-1</td>
<td>Frequently used notations and definitions</td>
</tr>
<tr>
<td>3-2</td>
<td>Average admission control/scheduling algorithm execution time (s)</td>
</tr>
<tr>
<td>3-3</td>
<td>Comparison of US and NS ($\tau = 5$ minutes)</td>
</tr>
<tr>
<td>3-4</td>
<td>Average number of slices of US and NS ($\tau = 5$ minutes)</td>
</tr>
<tr>
<td>3-5</td>
<td>Performance comparison of different algorithms</td>
</tr>
<tr>
<td>3-6</td>
<td>Rejection ratio of the simple scheme</td>
</tr>
</tbody>
</table>
## LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2-1</td>
<td>Examples of stringent rounding. The unshaded rectangles are time slices. The shaded rectangles represent jobs. The top ones show the requested start and end times. The bottom ones show rounded start and end times.</td>
<td>21</td>
</tr>
<tr>
<td>2-2</td>
<td>A network with 11 nodes and 13 bi-directional links, each of capacity 1GB shared in both directions.</td>
<td>24</td>
</tr>
<tr>
<td>2-3</td>
<td>The Abilene network with 11 backbone nodes. A and B are stub networks.</td>
<td>31</td>
</tr>
<tr>
<td>2-4</td>
<td>Z for different formulations on Abilene network using SSS. A) 121 jobs; B) 605 jobs; C) 1210 jobs; D) 6050 jobs.</td>
<td>33</td>
</tr>
<tr>
<td>2-5</td>
<td>Z for different formulations on a random network with 100 nodes using SSS. A) 100 jobs; B) 500 jobs; C) 1000 jobs; D) 5000 jobs.</td>
<td>34</td>
</tr>
<tr>
<td>2-6</td>
<td>Execution time for different formulations on the Abilene network using SSS. A) 121 jobs; B) 605 jobs; C) 1210 jobs; D) 6050 jobs.</td>
<td>35</td>
</tr>
<tr>
<td>2-7</td>
<td>Execution time for different formulations on a random network with 100 nodes using SSS. A) 100 jobs; B) 500 jobs; C) 1000 jobs; D) 5000 jobs.</td>
<td>35</td>
</tr>
<tr>
<td>2-8</td>
<td>Random network with $k = 8$. Execution time for different network sizes.</td>
<td>36</td>
</tr>
<tr>
<td>2-9</td>
<td>Average Z for different formulations on a random network with 100 nodes and 1000 jobs using SSS. The result is the average over 50 instances of the random network.</td>
<td>37</td>
</tr>
<tr>
<td>2-10</td>
<td>Average execution time for different formulations on a random network with 100 nodes and 1000 jobs using SSS. The result is the average over 50 instances of the random network.</td>
<td>37</td>
</tr>
<tr>
<td>2-11</td>
<td>Average throughput ratio for different formulations on a random network with 100 nodes and 1000 jobs using SSS. The result is the average over 50 instances of the random network.</td>
<td>37</td>
</tr>
<tr>
<td>2-12</td>
<td>Z for different formulations on the Abilene network with 121 jobs using MSS. A) Time slice = 60 min; B) Time slice = 30 min; C) Time slice = 15 min; D) Time slice = 10 min.</td>
<td>39</td>
</tr>
<tr>
<td>2-13</td>
<td>Z for different algorithms on a 100-node random network with 100 jobs using MSS. A) Time slice = 60 min; B) Time slice = 30 min; C) Time slice = 15 min; D) Time slice = 10 min.</td>
<td>39</td>
</tr>
<tr>
<td>2-14</td>
<td>Execution time for different formulations on the Abilene network with 121 jobs using MSS. A) Time slice = 60 min; B) Time slice = 30 min; C) Time slice = 15 min; D) Time slice = 10 min.</td>
<td>40</td>
</tr>
</tbody>
</table>
2-15 Execution time for different formulations on a 100-node random network with 100 jobs using MSS. A) Time slice = 60 min; B) Time slice = 30 min; C) Time slice = 15 min; D) Time slice = 10 min. ................................. 41

2-16 The Abilene network with 121 jobs and $k = 8$. A) Z for different time slices; B) Execution time for different time slice sizes. ................................. 41

3-1 Uniform time slice structure ................................. 44

3-2 Two rounding policies. The unshaded rectangles are time slices. The shaded rectangles represent jobs. The top ones show the requested start and end times. The bottom ones show the rounded start and end times. ................................. 46

3-3 Two-level nested time-slice structure. $\tau = 2, \Delta_1 = 4$ and $\Delta_2 = 1$. The anchored slice sets shown are for $t = \tau, 2\tau$ and $3\tau$, respectively. At-Most-$\sigma$ Design. $\sigma_2 = 8$. 56

3-4 Three-level nested time-slice structure. $\tau = 2, \Delta_1 = 16, \Delta_2 = 4$ and $\Delta_3 = 1$. The anchored slice sets shown are for $t = \tau, 2\tau$ and $8\tau$, respectively. At-Most-$\sigma$ Design. $\sigma_3 = 8, \sigma_2 = 2$. ................................. 57

3-5 Three-level nested slice structure Almost-$\sigma$ Variant. $\tau = 2, \Delta_1 = 16, \Delta_2 = 4$ and $\Delta_3 = 1$. The anchored slice sets shown are for $t = \tau, 2\tau$ and $3\tau$, respectively. $\sigma_3 = 8, \sigma_2 = 2$. The shaded areas are also slices, but are different in size from any level-$j$ slice, $j = 1, 2$ or 3. ................................. 58

3-6 Rejection ratio for different $\alpha$’s under SR. ................................. 64

3-7 Single vs. multiple paths under different traffic load. A) Response time; B) Rejection ratio. ................................. 65

3-8 Single vs. multiple paths under medium traffic load for different algorithms. A) Response time for QF; B) Response time for LB; C) Rejection ratio. ................................. 66

3-9 Scalability of the execution times with the number of jobs. ................................. 68

3-10 Scalability of the execution times with the number of time slices. ................................. 68

3-11 Scalability of the execution times with the network size. ................................. 69
ADVANCE RESERVATION AND SCHEDULING OF BULK FILE TRANSFERS IN E-SCIENCE

By

Kannan Rajah

August 2007

Chair: Dr. Sartaj Sahni
Major: Computer Engineering

The advancement of optical networking technologies has enabled e-science applications that often require transport of large volumes of scientific data. In support of such data-intensive applications, we develop and evaluate control plane algorithms for admission control and scheduling of bulk file transfers. Each file transfer request is made in advance to the central network controller by specifying a start time and an end time. If admitted, the network guarantees to begin the transfer after the start time and complete it before the end time. We formulate the scheduling problem as a special type of the multi-commodity flow problem. To cope with the start and end time constraints of the file-transfer jobs, we divide time into uniform time slices. Bandwidth is allocated to each job on every time slice and is allowed to vary from slice to slice. This enables periodical adjustment of the bandwidth assignment to the jobs so as to improve a chosen performance objective: throughput of the concurrent transfers. In this thesis, we study the effectiveness of using multiple time slices, the performance criterion being the trade-off between achievable throughput and the required computation time. Furthermore, we investigate using multiple paths for each file transfer to improve the throughput. We show that using a small number of paths per job is generally sufficient to achieve near optimal throughput with a practical execution time, and this is significantly higher than the throughput of a simple scheme that uses single shortest path for each job. The thesis combines the following novel elements into a cohesive framework of network resource
management: advance reservation, multi-path routing, rerouting and flow reassignment via periodic re-optimization. We evaluate our algorithm in terms of both network efficiency and the performance level of individual transfer. We also evaluate the feasibility of our scheme by studying the algorithm execution time.
CHAPTER 1
INTRODUCTION

The advancement of optical communication and networking technologies, together with the computing and storage technologies, is dramatically changing the ways how scientific research is conducted. A new term, *e-science*, has emerged to describe the “large-scale science carried out through distributed global collaborations enabled by networks, requiring access to very large scale data collections, computing resources, and high-performance visualization”. Well-quoted e-science (and the related grid computing [22]) examples include high-energy nuclear physics [10], radio astronomy, geoscience and climate studies.

The need for transporting large volume of data in e-science has been well-argued [1, 10, 33]. For instance, the HENP data is expected to grow from the current petabytes (PB) \(10^{15}\) to exabytes \(10^{18}\) by 2012 to 2015. Similarly, the Large Hadron Collider (LHC) facility at CERN is expected to generate petabytes of experimental data every year, for each experiment. In addition to the large volume, as noted in [17], “e-scientists routinely request schedulable high-bandwidth low-latency connectivity with known and knowable characteristics”. Instead of relying on the public Internet, national governments are sponsoring a new generation of optical networks to support e-science. Examples of such research and education networks include the Internet2 related National Lambda Rail and Abilene networks in the U.S., CA*net4 in Canada, and SURFnet in the Netherlands.

To meet the need of e-science, this thesis examines admission control and scheduling of high-bandwidth data transfers in the research networks. Admission control and network resource allocation are among the toughest classical problems for the Internet or any global-scale networks (See [16, 28] and their references.). There are three important aspects that motivate us to re-examine this issue, namely, specialized applications, fewer quality of service (QoS) classes and much smaller network size. Research networks are different from the public Internet as they typically have less than \(10^3\) core nodes in the
backbone. This makes it possible to have a centralized network controller for managing the network resources and for providing user service quality guarantee. With the central controller, there is more flexibility in designing sophisticated, efficient algorithms for scheduling user reservation requests, setting up network paths, and allocating bandwidth. Our work assumes that the optical network contains enough IP routers for traffic grooming, which is true for current research networks. Such a network allows fine-grained multiplexing of traffic for better network resource utilization.

The objective of this thesis is to develop and evaluate control plane algorithms for admission control (AC) and scheduling of large file transfers (also known as jobs) over optical networks. We assume that job requests are made in advance to a central network controller. Each request specifies a start time, an end time and the total file (demand) size. Such a request is satisfied as long as the network begins the transfer after the start time and completes it before the end time. There is, however, flexibility in how soon the transfer should be completed. It can be completed as soon as possible or, alternatively, be stretched until the requested end time. Our algorithms allow both possibilities and we will examine the consequences.

The network controller determines the admissibility of the new jobs by a process known as admission control (AC). Any admitted job will be guaranteed the performance level in accordance with its traffic class. The user of a rejected request may subsequently modify and re-submit the request. Once the jobs are admitted, the network controller has the flexibility in deciding the manner in which the files are transferred, i.e., how the bandwidth assignment to each job varies over time. This decision process is known as scheduling. Bulk transfer is not sensitive to the network delay but may be sensitive to the delivery time. It is useful for distributing high volumes of scientific data, which currently often relies on ground transportation of the storage media.

In Chapter 2, we focus on the scheduling problem at a single scheduling instance and compare different variations of the algorithm. Here, all file transfer requests are known in
advance; they can have different start and end times. We call this scheduling problem the *concurrent file transfer problem* (CFTP). There is no AC phase. We will formulate CFTP as a special type of the multi-commodity flow problem, known as the maximum concurrent flow (MCF) problem \[24, 36\]. While MCF is concerned with allocating bandwidth to persistent concurrent flows, CFTP has to cope with the start and end time constraints of the jobs. For this purpose, our formulations for CFTP involve dividing time into uniform time slices (Section 2.2) and allocating bandwidth to each job on every time slice. Such a setup allows an easy representation of the start and end time constraints, by setting the allocated bandwidth of a job to zero before the start time and after the end time. More importantly, in between the start and end times, the bandwidth allocated for each job is allowed to vary from time slice to time slice. This enables periodical adjustment of the bandwidth assignment to the jobs so as to improve some performance objective.

Motivated by the MCF problem, the chosen objective is the throughput of the concurrent transfers. For fixed traffic demand, it is well known that such an objective is equivalent to minimizing the worst-case link congestion, a form of network load balancing \[36\]. A balanced traffic load enables the network to accept more future job requests, and hence, achieve higher long-term resource utilization. In addition to the problem formulation, other contributions of this thesis are as follows. First, in scheduling file transfers over multiple time slices, we focus on the tradeoff between achievable throughput and the required computation time. Second, we investigate using multiple paths for each file transfer to improve the throughput. We will show that using a small number of paths per job is generally sufficient to achieve near optimal throughput, and this is shown to be significantly higher than the throughput of a simple scheme that uses single shortest path. In addition, the computation time for the formulation with a small number of paths is considerably shorter than that for the optimal scheme, which utilizes all possible paths for each job.
In Chapter 3, we describe a suite of algorithms for admission control and scheduling and compare their performance. Here, the file transfer requests arrive at different times; a decision needs to be taken at run time on which requests to be accepted and scheduled. Again, the key methodology is the discretization of time into a time slice structure so that the problems can be put into the linear programming framework. A highlight of our scheme is the introduction of non-uniform time slices, which can dramatically shorten the execution time of the AC and scheduling algorithms, making them practical (Section 3.6).

Our system handles two classes of jobs, bulk data transfer and those that require a minimum bandwidth guarantee (MBG). A request for the MBG class specifies a start time, an end time and the minimum bandwidth that the network should guarantee throughout the duration from the start to the end times. We assume that, once the bandwidth is granted, the optical network can be configured to achieve the desired low-latency for e-science. Such service is useful for realtime rendering or visualization of large volumes of data. In our framework, the algorithms for handling bulk transfer contain the main ingredients of the algorithms for handling the MBG class. For this reason, we will only give light treatment to the MBG class.

The e-science setting provides both new challenges and new possibilities for resource management that are not considered in the classical setting. The novel features of our work are as follows. First, bulk transfer is usually regarded as low-priority best-effort traffic, not subject to admission control in most QoS-provisioning frameworks such as InterServ [8], DiffServ [6], the ATM network [32], or MPLS [34]. The deadline-based AC and scheduling for the entire transfer (not each packet) has generally not been considered in traditional QoS frameworks. Second, our scheme allows each transfer session to take multiple paths rather than a single path. Third, the route and bandwidth assignment can be periodically re-evaluated and reassigned. This is in contrast to earlier schemes where such assignment remains fixed throughout the lifetime of the session.
To elaborate, we take the optimization approach for AC and scheduling on the ensemble of the jobs in the system. At each of the periodic AC and scheduling instances, AC is first administered. The admission of new jobs is formulated as a feasibility problem subject to the constraint that the existing jobs admitted earlier must retain their performance guarantee. However, to increase the admission rate, the routes and bandwidth of the existing jobs can be reassigned. In the second step, scheduling, the network controller assigns the actual routes and bandwidth to all jobs in the system so as to optimize a performance objective. Examples that we consider in this chapter are to minimize the worst case link utilization or to minimize an objective that encourages earlier completion of the jobs. The result of scheduling in turn affects the admission rate for future jobs. The classical AC schemes do not conduct periodic rerouting or bandwidth re-allocation of existing jobs. They only ask if the remaining network capacity is sufficient to handle new jobs. Furthermore, there is no additional scheduling step for performance optimization on all jobs in the system.

The rest of this thesis is organized as follows. The related work is shown in Section 1.1. There are two main technical contributions of this thesis: CFTP, described in Chapter 2 and Admission Control/Scheduling algorithms described in 3. In addition to the proposed formulations, we present a rigorous discussion on their experimental results in Section 2.5 and 3.7, respectively. Finally, the conclusions are drawn in Chapter 4.

1.1 Related Work

Our work is focused on building an efficient scheduling framework to perform advance reservation of bulk file transfer requests with admission control. The main technical contributions of this thesis are as follows: Path based scheduling is close to the optimal solution and also fast; Use of multiple paths and multiple time slices for scheduling; Non-uniform time slice structure to enable long coverage of reservation; Periodic re-optimization of flows to achieve better network utilization. Similar to our work, the authors of [5] also advocate periodic re-optimization to determine new routes
and bandwidth in optical networks. They also use a multi-commodity flow formulation. However, they do not assume users making advance reservations with requested start and end times. As a result, the scheduling problem is for a single time instance, rather than over multiple time slices. Furthermore, it does not consider the edge-path formulation with limited number of paths per job.

Several earlier studies \cite{9, 11, 13, 15, 35, 37, 38} consider advance bandwidth reservation with start and end times at an individual link for traffic that requires minimum bandwidth guarantee (MBG). The concern is typically about designing efficient data structures for keeping track of and querying bandwidth usage at the link on different time intervals. New jobs are admitted one at a time without changing the bandwidth assignment of the existing jobs in the system. The admission of a new job is based on the availability of the requested bandwidth between its start time and end time. \cite{11, 14, 19, 25, 37} and \cite{15} all go beyond single-link advance reservation and tackle the more general path-finding problem for the MBG traffic class, but typically only for the new requests, one at a time. The routes and bandwidth of the existing jobs are unchanged. \cite{12} discusses architectural and signaling-protocol issues about advance reservation of network resources. \cite{30} considers a network with known routing in which each admitted job derives a profit. It gives approximation algorithms for admitting a subset of the jobs so as to maximize the total profit.

\cite{14, 25} touch upon advance reservation for bulk transfer. \cite{14} proposes a malleable reservation scheme. The scheme checks every possible interval between the requested start time and end time for the job and tries to find a path that can accommodate the entire job on that interval. It favors intervals with earlier deadlines. \cite{25} studies the computation complexity of a related path-finding problem and suggests an approximation algorithm. \cite{31} starts with an advance reservation problem for bulk transfer. Then, the problem is converted into a bandwidth allocation problem at a single time instance to maximize the job acceptance rate. This is shown to be an NP-hard combinatorial problem. Heuristic
algorithms are then proposed. Many papers study advance reservation, re-routing, or re-optimization of lightpaths, at the granularity of a wavelength, in WDM optical networks [4, 7, 40]. They are complementary to our study.

In the control plane, [27] and [26] present architectures for advance reservation of intra and interdomain lightpaths. The DRAGON project [29] develops control plane protocols for multi-domain traffic engineering and resource allocation on GMPLS-capable [18] optical networks. GARA [23], the reservation and allocation architecture for the grid computing toolkit, Globus, supports advance reservation of network and computing resources. [20] adapts GARA to support advance reservation of lightpaths, MPLS paths and DiffServ paths.
CHAPTER 2
CONCURRENT FILE TRANSFER PROBLEM

2.1 Problem Definition

A network is represented as a directed graph $G = (V, E)$ where $V$ is the set of nodes and $E$ is the set of edges (or arcs). Each edge $e \in E$ represents a link whose capacity is denoted by $C_e$. A path $p$ is understood as a collection of links with no cycles. Job requests are submitted to the network using a 6-tuple representation $(A_i, s_i, d_i, D_i, S_i, E_i)$, where $A_i$ is the arrival time of the request, $s_i$ and $d_i$ are source and destination nodes, respectively, $D_i$ is the size of the file, $S_i$ and $E_i$ are requested start service time and end service time, where $A_i \leq S_i \leq E_i$. The meaning of the 6-tuple is, request $i$ is made at time $t = A_i$, asking the network to transfer a file of size $D_i$ from source node $s_i$ to destination node $d_i$ over the time interval $[S_i, E_i]$.

In our framework, the network resource is managed by a central network controller. File transfer requests arrive following a random process and are submitted to the network controller. The network controller verifies admissibility of the jobs through a process known as admission control (AC). Admitted jobs are thereafter scheduled with a guarantee of the start and end time constraints. Chapter 3 is devoted to a discussion on how the AC and scheduling algorithms work together. In this chapter, we focus on the scheduling problem at a single scheduling instance and compare different variations of the algorithm. There is no AC phase.

More specifically, we have the following scheduling problem. At a scheduling instance $t$, we have a network $G = (V, E)$ and the link capacity vector $C = (C_e)_{e \in E}$. The network may have some on-going file transfers; it may also have some jobs that were admitted earlier but yet to be started. The capacity $C$ is understood as the remaining capacity, obtained by removing the bandwidth committed to all unfinished jobs admitted prior to
The network controller has a collection of new job requests, denoted by \( J \). The task of the network controller is to schedule the transfer of the jobs in \( J \) so as to optimize a network efficiency measure. The chosen measure, which will be further explained later, is the value \( Z \) such that, if the demands are all scaled by \( Z \) (i.e., from \( D_i \) to \( ZD_i \) for every job \( i \)), they can be carried by the network without exceeding any link capacity. Such a \( Z \) is known as the throughput.

2.2 The Time Slice Structure

At any scheduling time \( t \), the timeline from \( t \) onward is divided into uniform time slices (intervals). The set of time slices starting from time \( t \) is denoted as \( \mathcal{G}_t \). The bandwidth assignment to each job is done on every time slice. In other words, the bandwidth reserved for a job remains constant throughout the time slice, but it can vary across time slices. At the scheduling time \( t \), let the time slices in \( \mathcal{G}_t \) be indexed as \( 1, 2, \ldots \) in increasing order of time. Let the start and end time of slice \( i \) be denoted by \( ST_t(i) \) and \( ET_t(i) \), respectively, and let its length be \( LEN_t(i) \). We say a time instance \( t' > t \) falls into slice \( i \) if \( ST_t(i) < t' \leq ET_t(i) \). The index of the slice that \( t' \) falls in is denoted by \( I_t(t') \).

The time slice structure is useful for bulk file transfers, wherein a request is satisfied as long as the network transfers the entire file between the start and end time. Such jobs offer a high degree of flexibility to the network in modulating the bandwidth assignment across time slices. This is in contrast to applications that require minimum bandwidth guarantee, for which the network must maintain the minimum required bandwidth from the start to the end time.

**Rounding of the start and end time.** While working with the time slice structure, the start and end time of the jobs should be adjusted to align on the slice

---

1 We no longer need to consider the request arrival times, \( A_i \), for \( i \in J \). We may take \( A_i = t \) for \( i \in J \).
boundaries. This is required because bandwidth assignment is done on a slice level. To illustrate, consider a file request \( (A_i, s_i, d_i, D_i, S_i, E_i) \). Let the rounded start and end time be denoted as \( \hat{S}_i \) and \( \hat{E}_i \), respectively. We round the requested start time \( S_i \) to be the maximum of the current time or the end time of the slice in which \( S_i \) falls, i.e.,

\[
\hat{S}_i = \max\{t, ET_t(I_t(S_i))\}. \tag{2.1}
\]

For rounding of the requested end time, we follow a stringent policy wherein the end time is rounded down, subject to the constraint that \( \hat{E}_i > \hat{S}_i \). That is, there has to be at least one-slice separation between the rounded start and end time. Otherwise, there is no way to schedule the job. More specifically,

\[
\hat{E}_i = \begin{cases} 
ET_t(I_t(\hat{S}_i) + 1) & \text{if } ST_t(I_t(E_i)) \leq \hat{S}_i \\
E_i & \text{else if } ET_t(I_t(E_i)) = E_i \\
ST_t(I_t(E_i)) & \text{otherwise.}
\end{cases} \tag{2.2}
\]

Fig. 2-1 shows several rounding examples. In practice, several variations of this strategy can be adopted. From the definition of uniform slices, the slice set anchored at \( t \), \( G_t \), contains infinitely many slices. In general, only a finite subset of \( G_t \) is useful to us. Let \( M_t \) be the index of last slice in which the rounded end time of some job falls. That is, \( M_t = I_t(\max_{i \in J} \hat{E}_i) \). Let \( L_t \subset G_t \) be the collection of time slices \( \{1, 2, ..., M_t\} \). It is sufficient to consider \( L_t \) for scheduling.

![Jobs and Jobs After Rounding](image)

Figure 2-1. Examples of stringent rounding. The unshaded rectangles are time slices. The shaded rectangles represent jobs. The top ones show the requested start and end times. The bottom ones show rounded start and end times.
The maximum concurrent file transfer problem is formulated as a special type of network linear programs (LP), known as the maximum concurrent flow problem (MCF) [24, 36]. We consider both the node-arc form and the edge-path form of the problem.

2.3 Node-Arc Form

Let \( f^i_{l,k}(j) \) be the total amount of data transfer on link \((l, k) \in E\) that is assigned to job \(i \in J\) on the time slice \(j \in \mathcal{L}_t\). We will loosely call it the flow for job \(i\) on arc \((l, k)\) on time slice \(j\).

\[
\text{Node-Arc}(t, J)
\]

\[
\begin{align*}
\text{max} & \quad Z \\
\text{subject to} & \quad \sum_{k: (l,k) \in E} f^i_{(l,k)}(j) - \sum_{k: (k,l) \in E} f^i_{(k,l)}(j) \\
& \quad = \begin{cases} y^i(j) & \text{if } l = s_i \\ -y^i(j) & \text{if } l = d_i \\ 0 & \text{otherwise} \end{cases} \\
& \quad \forall i \in J, \forall l \in V, \forall j \in \mathcal{L}_t
\end{align*}
\]

\[
\sum_{j=1}^{M_t} y^i(j) = Z \Delta_i \quad \forall i \in J \quad (2.4)
\]

\[
\sum_{i \in J} f^i_{(l,k)}(j) \leq C_{(l,k)}(j) LEN_t(j), \quad \forall (l, k) \in E, \forall j \in \mathcal{L}_t \quad (2.6)
\]

\[
f^i_{(l,k)}(j) = 0, \quad j \leq I_t(\hat{S}_i) \text{ or } j > I_t(\hat{E}_i), \quad \forall i \in J, \forall (l, k) \in E \quad (2.7)
\]

\[
f^i_{(l,k)}(j) \geq 0, \quad \forall i \in J, \forall j \in \mathcal{L}_t, \forall (l, k) \in E. \quad (2.8)
\]

Condition (2.4) is the flow conservation equation that is required to hold on every time slice \(j \in \mathcal{L}_t\). It says that, for each job \(i\), if node \(l\) is neither the source node for job \(i\) nor its destination, then the total flow of job \(i\) that enters node \(l\) must be equal to the
total flow of job $i$ that leaves node $l$. Moreover, on each time slice, the supply of job $i$ from its source must be equal to the demand at job $i$'s destination. This common quantity is denoted by $y^i(j)$ for job $i$ on time slice $j$. Condition (2.5) says that, for each job, the total supply (or, equivalently, total demand), when summed over all time slices, must be equal to $Z$ times the job size, where $Z$ is the variable to be maximized. Condition (2.6) says that the capacity constraints must be satisfied for all edges on every time slice. Note that the allocated rate on link $(l, k)$ for job $i$ on slice $j$ is $f^i_{(l,k)}(j)/LEN_t(j)$, where $LEN_t(j)$ is the length of slice $j$. The rate is assumed to be constant on the entire slice. Here, $C_{(l,k)}(j)$ is the capacity of link $(l, k)$ on slice $j$. In all the experiments in this paper, each link capacity is assumed to be a constant across the time slices, i.e., $C_{(l,k)}(j) = C_{(l,k)}$ for all $j$. But, the formulation allows the more general time-varying link capacity. (2.7) is the start and end time constraint for every job on every link. The flow must be zero before the rounded start time and after the rounded end time.

The linear program asks, what is the largest constant scaling factor $\hat{Z}$ such that, after every job size is scaled by $\hat{Z}$, the link capacity constraints, as well as the start and end time constraints, are still satisfied for all time slices? Let the optimal flow vector for the linear program be denoted by $\hat{f} = (\hat{f}^i_{(l,k)}(j))_{i,l,k,j}$. If $\hat{Z} \geq 1$, then the flow $\hat{Z}\hat{f}$ can still be handled by the network without the link capacity constraints being violated. If, in practice, the flow vector $\hat{Z}\hat{f}$ is used instead of $\hat{f}$, the file transfer can be completed faster. If $\hat{Z} < 1$, it is not possible to satisfy the deadline of all the jobs. However, if the file sizes are reduced by a common factor $\hat{Z}D_i$ for all $i$, then, the requests can all be satisfied.

There exists a different perspective to our optimization objective. Maximizing the throughput of the concurrent flow is equivalent to finding a concurrent flow that carries all the demands and also minimizes the worst-case link utilization, i.e., link congestion. To see this, we make the following substitution, $\tilde{f} = f/Z$. For our case, the largest link utilization over all links and across all time slices is minimized. The result is that the traffic load is balanced over the whole network and across all time slices. This feature
is desirable if the network also carries other types of traffic that is sensitive to network load bursts, such as real-time traffic or traffic requiring minimum bandwidth guarantee. In addition, by reserving only the minimum bandwidth in each time slice, more future requests can potentially be accommodated.

The problem formulated here is related to the MCF problem. The difference is that, in the MCF problem, the time dimension does not exist. Our problem becomes exactly the MCF problem if \( M_t = 1 \) (i.e., there is only one time slice) and if the constraints for the start and end times of the jobs, \((2.7)\), are removed. In the MCF problem, the variable \( Z \) is called the *throughput* of the concurrent flow. The MCF problem has been studied in a sequence of papers, e.g., \([2, 3, 21, 24, 36]\). Several approximation algorithms have been proposed, which run faster than the usual simplex or interior point methods. For our problem, we can replicate the graph \( G \) into a sequence of temporal graphs representing the network at different time slices and use virtual source and destination nodes to connect them. We then have an MCF problem on the new graph and we can apply the fast approximation algorithms to this MCF instance.

![Figure 2-2](image.png)

Figure 2-2. A network with 11 nodes and 13 bi-directional links, each of capacity 1GB shared in both directions.

**Example-1:** Consider the network shown in Fig. 2-2 with two file transfer requests, \( J_1 : (0, 1, 9, 8000, 0, 60) \) and \( J_2 : (0, 3, 6, 1000, 0, 60) \). Here, we have used our 6-tuple convention to represent the requests. Both jobs requests arrive at time 0. The start and end times are both at \( t = 0 \) and \( t = 60 \), respectively. The job size is measured in GB and the time
in minutes. When we schedule using a single slice of length 60 minutes, the node-arc formulation gives the following flow reservation for each job on edges $e_1$ through $e_{13}$.

$J_1 : \{3600, 0, 0, 0, 0, 0, 0, 0, 3600, 3600, 3600, 3600, 3600, 0, 3600\}$

$J_2 : \{0, 0, 900, 900, 900, 0, 0, 0, 0, 0, 0, 0, 0\}$

The throughput $Z$ is 0.9, which is optimal.

The number of variables required to solve the node-arc model is $\Theta(|E| \times |L_t| \times |J|)$, because, for every job, there is an arc flow variable associated with every link for every time slice. The resulting problem is computationally expensive even with the fast approximation algorithms. In Section 2.4, we will consider the edge-path form of the problem, where every job is associated with a set of path-flow variables corresponding to a small number of paths, for every time slice.

### 2.4 Edge-Path Form

The edge-path formulation uses a set of simple paths for each $i \in J$ and determines the flow on each of these paths on every time slice. The number of possible simple paths can actually be higher than the number of arcs and therefore the edge-path form has no computational advantage over the node-arc form. To avoid the computational complexity, we consider sub-optimal formulations where we allow only a small number of paths for each job. In such a setting, the edge-path form is an appropriate formulation.

Let $P_t(s_i, d_i)$ be the set of allowed paths for job $i$ (from the source node $s_i$ to the destination $d_i$). Let $f_p^i(j)$ be the total amount of data transfer on path $p \in P_t(s_i, d_i)$ that is assigned to job $i \in J$ on the time slice $j \in L_t$. We will loosely call it the flow for job $i$ on path $p$ on time slice $j$. 

25
Edge-Path\((t, J)\)

\[
\text{max} \quad Z \tag{2.9}
\]

subject to \(\sum_{j=1}^{M_t} \sum_{p \in P_t(s_i, d_i)} f^i_p(j) = ZD_i, \quad \forall i \in J\) \tag{2.10}

\[
\sum_{i \in J} \sum_{p \in P_t(s_i, d_i)} \sum_{e \in p} f^i_p(j) \leq C_e(j)\text{LEN}_t(j), \quad \forall e \in E, \forall j \in \mathcal{L}_t \tag{2.11}
\]

\[
f^i_p(j) = 0, \quad j \leq I_t(\hat{S}_i) \text{ or } j > I_t(\hat{E}_i), \quad \forall i \in J, \forall p \in P_t(s_i, d_i) \tag{2.12}
\]

\[
f^i_p(j) \geq 0, \quad \forall i \in J, \forall j \in \mathcal{L}_t, \forall p \in P_t(s_i, d_i). \tag{2.13}
\]

Condition (2.10) says that, for every job, the sum of all the flows assigned on all time slices for all allowed paths must be equal to \(Z\) times the job size, where \(Z\) is the variable to be maximized. (2.11) says that the capacity constraints must be satisfied for all edges on every time slice. Note that the allocated rate on path \(p\) for job \(i\) on slice \(j\) is \(f^i_p(j)/\text{LEN}_t(j)\), where \(\text{LEN}_t(j)\) is the length of slice \(j\). \(C_e(j)\) is the capacity of link \(e\) on slice \(j\). (2.13) is the start and end time constraint for every job on every allowed path. The flow must be zero before the rounded start time and after the rounded end time.

The edge-path formulation allows an explicitly defined collection of paths for each file-transfer job and flow reservations are done only on these paths. The number of variables required to solve the edge-path model is \(\Theta(k \times |\mathcal{L}_t| \times |J|)\), where \(k\) is the maximum number of paths allowed for each job. We will examine two possible collections of paths, \(k\)-shortest paths and \(k\)-shortest disjoint paths.

### 2.4.1 Shortest Paths

We use the algorithm in [39] to generate \(k\)-shortest paths. This algorithm is not the fastest one, but is easy to implement. Also, in Section 2.4.2, we will use it as a
building block in our algorithm for finding \( k \)-shortest disjoint paths. The key steps of the \( k \)-shortest-path algorithm are

1. Compute the shortest path using Dijkstra’s algorithm. This path is called the \( i^{th} \) shortest path for \( i = 1 \). Set \( B = \emptyset \).

2. Generate all possible deviations to the \( i^{th} \) shortest path and add them to \( B \). Pick the shortest path from \( B \) as the \( (i + 1)^{th} \) shortest path.

3. Repeat step 2) until \( k \) paths are generated or there are no more paths possible (i.e., \( B = \emptyset \)).

Given a sequence of paths \( p_1, p_2, ..., p_k \) from node \( s \) to \( d \), the deviation to \( p_k \) at its \( j^{th} \) node is defined as a new path, \( p \), which is the shortest path under the following constraint. First, \( p \) overlaps with \( p_k \) up to the \( j^{th} \) node, but the \( (j + 1)^{th} \) node of \( p \) cannot be the \( (j + 1)^{th} \) node of \( p_k \). In addition, if \( p \) also overlaps with \( p_l \) up to the \( j^{th} \) node, for any \( l = 1, 2, ..., k - 1 \), then the \( (j + 1)^{th} \) node of \( p \) cannot be the \( (j + 1)^{th} \) node of \( p_l \).

**Example-2:** Let us apply the edge-path formulation with \( k \)-shortest paths to the file transfer requests in Example-1 for the network shown in Fig. 2-2. The case of \( k = 1 \) corresponds to using the single shortest path for each job. Let \( p_j^i \) denote the \( j^{th} \) shortest path for job \( i \). The shortest paths are,

\[
p_1^1 : 1 - 11 - 10 - 9 \quad p_1^2 : 3 - 2 - 7 - 6
\]

Flow reservation for each job is given by

\[
f_{p_1^1}^1(1) = 3600 \quad f_{p_1^2}^2(1) = 450
\]

The throughput is 0.45, which is only half the optimal value obtained from the node-arc formulation.
For the case $k = 2$, i.e., with two shortest paths per job, we have,

- $p_1^1 : 1 - 11 - 10 - 9$
- $p_1^2 : 3 - 2 - 7 - 6$
- $p_2^1 : 1 - 2 - 10 - 9$
- $p_2^2 : 3 - 4 - 5 - 6$
- $f_{p_1^1}(1) = 3600$
- $f_{p_1^2}(1) = 450$
- $f_{p_2^1}(1) = 0$
- $f_{p_2^2}(1) = 0$

The total flow for $J_1$ is $f_{p_1^1}(1) + f_{p_1^2}(1) = 3600$. The total flow for $J_2$ is $f_{p_2^1}(1) + f_{p_2^2}(1) = 450$. The throughput is $0.45$.

From $k = 1$ to 2, we do not find any throughput improvement. This is because for $J_1$, the second path shares an edge with the first, and hence, the total flow reaching the destination node is limited to $3600$. By increasing the number of paths per job from 2 to 4, we get the following results.

- $p_1^1 : 1 - 11 - 10 - 9$
- $p_1^2 : 3 - 2 - 7 - 6$
- $p_2^1 : 1 - 2 - 10 - 9$
- $p_2^2 : 3 - 4 - 5 - 6$
- $p_3^1 : 1 - 2 - 7 - 8 - 9$
- $p_3^2 : 3 - 2 - 10 - 9 - 8 - 7 - 6$
- $p_4^1 : 1 - 11 - 10 - 2 - 7 - 8 - 9$
- $p_4^2 : 3 - 2 - 1 - 11 - 10 - 9 - 8 - 7 - 6$

- $f_{p_1^1}(1) = 3600$
- $f_{p_1^2}(1) = 0$
- $f_{p_2^1}(1) = 0$
- $f_{p_2^2}(1) = 900$
- $f_{p_3^1}(1) = 3600$
- $f_{p_3^2}(1) = 0$
- $f_{p_4^1}(1) = 0$
- $f_{p_4^2}(1) = 0$

The total flow for $J_1$ is $7200$; the total flow for $J_2$ is $900$. The throughput is $0.9$. This is equal to the optimal value achieved by the node-arc formulation.
2.4.2 Shortest Disjoint Paths

One interesting aspect that we noticed in Example-2 is that, while the $k$-shortest path algorithm minimizes the number of links used, the $k$-shortest paths for each job have a tendency to overlap on some links. As a result, addition of new paths do not necessarily improve the throughput. This motivates us to consider the $k$-shortest disjoint paths.

The algorithm for finding the $k$-shortest disjoint paths from node $s$ to $d$ is straightforward if such $k$ paths indeed exist. Given the directed graph $G$, in the first step of the algorithm, we find the shortest path from node $s$ to $d$, and then we remove all the edges on the path from the graph $G$. In the next step, we find the shortest path in the remaining graph, and then remove those edges on the selected path to create a new remaining graph. The algorithm continues until we find $k$ paths.

When the number of disjoint paths is less than $k$, we first find all the disjoint paths and then resort to the following heuristics to select additional paths so that the total number of selected paths is $k$. Let $S$ be the list of selected disjoint paths.

1. Set $S$ to be an empty list. Set $B = \emptyset$.
2. Find all the disjoint paths between the source $s$ and destination $d$ and append them to $S$ in the order they are found. Let $p$ be the first path in the list $S$.
3. Generate the deviations for $p$ and add them to $B$.
4. Select the path in $B$ that has the least number of overlapped edges with the paths in $S$, and append it to $S$.
5. Set $p$ to be the next path in the list $S$.
6. Repeat from step 3) until $S$ contains $k$ paths or there are no more paths possible (i.e., $B = \emptyset$).

In the above steps, the set $B$ contains short paths, generated from the deviations of some already selected disjoint paths. The newly selected path from $B$ has the least overlap with the already selected ones. It should be noted that while this approach reduces the overlap between the $k$ paths of each job, it does not guarantee the same for paths across jobs. This is because, the average path length of $k$-shortest disjoint paths tends to be
greater than that of the $k$-shortest paths, potentially causing the shortest disjoint paths of one job to heavily overlap with those of other jobs. This can have a negative effect on the overall throughput.

**Example-3:** Let us apply the $k$-shortest disjoint paths to Example-1. For $k = 2$, we have,

\[
p_1^1 : 1 - 11 - 10 - 9 \\
p_2^1 : 1 - 2 - 7 - 8 - 9 \\
f_{p_1^1}^1(1) = 3600 \\
f_{p_2^1}^1(1) = 3600
\]

\[
p_1^2 : 3 - 2 - 7 - 6 \\
p_2^2 : 3 - 4 - 5 - 6 \\
f_{p_1^2}^2(1) = 450 \\
f_{p_2^2}^2(1) = 450
\]

The total flow for $J_1$ is $f_{p_1^1}^1(1) + f_{p_2^1}^1(1) = 7200$. The total flow for $J_2$ is $f_{p_1^2}^2(1) + f_{p_2^2}^2(1) = 900$. The throughput is 0.9. Hence, the optimal throughput is achieved with $k = 2$.

### 2.5 Evaluation

This section shows the performance results of the edge-path formulation using the single and multi-path schemes. We compare its throughput with the optimal solution obtained from node-arc formulation. The scalability of the formulations are evaluated based on their required computation time.

The experiments were conducted on random networks and Abilene, an Internet2 high-performance backbone network (Fig. 2-3). The random networks have between 100 and 1000 nodes with a varying node degree of 5 to 10. Our instance of the Abilene network consists of a backbone with 11 nodes, in which each node is connected to a randomly generated stub network of average size 10. The backbone links are each 10GB. The entire network has 121 nodes and 490 links. We use the commercial CPLEX package for solving linear programs on Intel-based workstations\(^2\). In order to simulate the file

\(^2\) Since fast approximation algorithms are not the focus of this thesis, we use the standard LP solver for the evaluations.
size distribution of Internet traffic, we resort to the widely accepted heavy-tailed Pareto distribution, with the distribution function $F(x) = 1 - (x/b)^{-\alpha}$, where $x \geq b$ and $\alpha > 1$. As $\alpha$ value gets closer to 1, the distribution becomes more heavy-tailed and there is a higher probability of generating large file sizes. All the experiments described in this section were done using Pareto parameter $\alpha = 1.8$ and an average job size of 50GB. The plots use the following acronyms, S (Shortest path), SD (Shortest Disjoint path) and NA (Node-Arc).

While configuring the simulation environment, we can ignore the connection setup (path setup for the edge-path form) time for the following reasons. First, the small network size allows us to pre-compute the allowed paths for every possible request. Second, in the actual operation, the scheduling algorithm runs every few minutes or every tens of minutes. There is plenty of time to re-configure the control parameters for the paths in the small research network.

Figure 2-3. The Abilene network with 11 backbone nodes. A and B are stub networks.

2.5.1 Single Slice Scheduling (SSS)

When $|\mathcal{L}_r| = 1$ in the node-arc and edge-path formulations, we call the situation single slice scheduling (SSS). In this experiment, we keep the time-slice structure simple in order to examine how other factors affect the performance of different formulations. All jobs start at the 0th minute and end at 60th minute. Scheduling is done at time 0 with a (single) time slice size equal to 60 minutes.
2.5.1.1 Performance comparison of the formulations

Fig. 2-4 shows the throughput improvement on the Abilene network with increasing number of paths for the shortest (S) and shortest disjoint (SD) schemes, respectively. The optimal throughput obtained from the node-arc (NA) form is shown as a horizontal line. Similar plots are shown in Fig. 2-5 for a random network with 100 nodes\(^3\).

**Single v.s. Multiple paths.** Moving from a single path to multiple paths per job, we observe a drastic throughput increase. A small number of paths per job is sufficient to realize such throughput improvement. On the Abilene network, the throughput is increased by up to 10 times with 4 to 8 paths per job. Simply by switching from a single path to two paths per job, we observe 60% throughput gain. On the random network, the throughput is increased by 10 to 30 times with 4 or more paths. In most of our examples, the S and SD schemes reach the optimal throughput with \(k = 8\) or less.

In summary, the optimal throughput obtained from our multi-path scheme is significantly higher than that of a simple scheme, which uses single shortest path for every job. Throughput improvement by an order of magnitude can be expected with only a small number of paths. The performance gains saturate at around 8 paths in most of our simulation - the exact number in general depends on the topology and actual traffic.

**Shortest (S) v.s. Shortest Disjoint (SD) paths.** For random networks, SD tends to perform better than S. In most of our examples, the throughput of SD is several times higher than that of S for \(k = 2\) to 8. For the Abilene network, the opposite trend can often be observed. This behavior can be explained as follows. As we have mentioned in Section 2.4, the paths for different jobs have a higher chance to overlap in the SD case, potentially causing throughput degradation. In a well-connected random network, disjoint or nearly

---

\(^3\) The node-arc case is not shown in Fig. 2-5 (d) and in several subsequent figures because the problem size becomes too large to be solved on our workstations with 2 to 4 GB of memory, mainly due to the large memory requirement.
disjoint paths are more abundant and also tend to be short. The throughput benefit from the disjoint paths exceeds the throughput degradation from the longer average path length. On the other hand, in the Abilene network, the backbone network has few disjoint paths between each pair of nodes. Insisting on having (nearly) disjoint paths leads to longer average path length due to the lack of choices. Hence, the throughput penalty from longer path length is more pronounced in a small network such as Abilene. Therefore, it is often more beneficial to use the shortest paths instead.

In summary, we expect SD to be preferable in large, well-connected networks. In a small network with few disjoint paths, the performance of S and SD are generally comparable, with S sometimes being better. Finally, the difference between S and SD disappears quickly as the number of paths per job increase.

Figure 2-4. Z for different formulations on Abilene network using SSS. A) 121 jobs; B) 605 jobs; C) 1210 jobs; D) 6050 jobs.

2.5.1.2 Comparison of algorithm execution time

Recall that our motivation to move from the node-arc formulation to the edge-path formulation is that the latter allows us to restrict the number of permitted paths for each job, resulting in lower algorithm execution time. Fig. 2-6 and Fig. 2-7 show the execution
Figure 2-5. Z for different formulations on a random network with 100 nodes using SSS.
A) 100 jobs; B) 500 jobs; C) 1000 jobs; D) 5000 jobs.

We observe that the execution time for S or SD increases roughly linearly, when the number of permitted paths per job is small (up to 16 paths in the figures). With several hundred jobs or more, even the longest execution time (at 16 paths) is much shorter than that for the node-arc case, by an order of magnitude. We expect this difference in execution time to increase with more jobs and larger networks.

In Fig. 2-6 C and D, we see that the scheduling time for the node-arc formulation approaches or exceeds the actual 60-minute transfer time of the files. On the other hand, the edge-path formulation with a small number of allowed paths, is much more scalable with traffic intensity. Fast approximation algorithms in [2, 3, 21, 24, 36], if used, should

---

4 Unless mentioned otherwise, the execution time for the edge-path formulations does not include the path computation time for finding the shortest paths. This is because the shortest paths are computed only once, and the computation can be carried out off-line.
improve the execution time for all formulations. But, the significant difference between the node-arc case and the shortest or shortest disjoint cases should still remain.

Figure 2-6. Execution time for different formulations on the Abilene network using SSS. A) 121 jobs; B) 605 jobs; C) 1210 jobs; D) 6050 jobs.

Figure 2-7. Execution time for different formulations on a random network with 100 nodes using SSS. A) 100 jobs; B) 500 jobs; C) 1000 jobs; D) 5000 jobs.

2.5.1.3 Algorithm scalability with network size

Fig. 2-8 shows the variation of the algorithm execution time with network size. In our simulations, we schedule 100 jobs using SSS for a period of 60 minutes. The
edge-path algorithms (S and SD) with 8 paths have an execution time under 10 seconds for networks with less than 800 nodes. On the other hand, the execution time for the node-arc algorithm is nearly 15 minutes for a network size of 500 nodes. We conclude that the node-arc formulation is unsuitable for real-time scheduling of file transfers on networks of more than several hundred nodes.

Figure 2-8. Random network with \( k = 8 \). Execution time for different network sizes.

### 2.5.1.4 Average results over random network instances

When the experiments are conducted on random networks, unless mentioned otherwise, each plot typically presents the results obtained from a single network instance rather than an average result over many network instances. To demonstrate that the single-instance results are not anomalies but representative, we repeated the experiments in Section 2.5.1 for a 100-node random network and plotted the data points averaged over 50 network instances. Due to space limitation, we present only the results for 1000 jobs in Fig. 2-9. This should be compared with Fig. 2-5 C, which is for a single network instance. Besides the fact that the curves in Fig. 2-9 are smoother, the two figures show similar characteristics. All the observations that we have made about Fig. 2-5 C remain essentially true for Fig. 2-9. We should point out that, in order to run the experiment on many network instances in a reasonable amount of time, the networks for Fig. 2-9 were generated with fewer links than that for Fig. 2-5 C. This accounts for the difference in the throughput values between the two cases. Finally, the corresponding average execution time is shown in Fig. 2-10 on semilog scale.

We further confirmed the validity of our data and results by computing the confidence interval of the mean values plotted in Fig. 2-9. For instance, the mean and standard
Figure 2-9. Average Z for different formulations on a random network with 100 nodes and 1000 jobs using SSS. The result is the average over 50 instances of the random network.

Figure 2-10. Average execution time for different formulations on a random network with 100 nodes and 1000 jobs using SSS. The result is the average over 50 instances of the random network.

deviation of the throughput for node-arc formulation is 0.1489 and 0.0807, respectively. The 95% confidence interval for the mean is ±0.0188 around the mean. This is a good indicator of the accuracy of our results.

In addition, we also computed the average of the throughput ratio of S and SD schemes to the node-arc formulation. In Fig. 2-11, both S and SD schemes achieve nearly 80% of the optimal throughput by switching from single path to 2 paths. The throughput reaches 99% with 8 paths. For \( k \leq 4 \), SD performs better than S. The plot is consistent with our earlier results shown in Fig. 2-9.

Figure 2-11. Average throughput ratio for different formulations on a random network with 100 nodes and 1000 jobs using SSS. The result is the average over 50 instances of the random network.
2.5.2 Multiple Slice Scheduling (MSS)

When $|L_e| > 1$ in the node-arc and edge-path formulations, we call the situation multiple slice scheduling (MSS). In this experiment, 121 jobs are scheduled for a period of 1 day using multiple slices of identical size. The interval between the start times of the jobs are independently and identically distributed exponential random variables with a mean of 1 minute. We have tried four time-slice sizes, 60, 30, 15 and 10 minutes.

2.5.2.1 Performance comparison of different formulations

Fig. 2-12 shows the throughput improvement for the Abilene network with increasing number of paths for the S and SD schemes, respectively. The throughput of the node-arc formulation is shown as a flat line.

For each fixed slice size, the general behavior of the throughput follows the same pattern as the SSS case discussed in Section 2.5.1.1. In particular, the throughput improvement is significant as the number of paths per job decreases. In Fig. 2-12, we observe more than 50% throughput increase with 4 or fewer paths and nearly 30% to 50% increase with 8 or more paths. When comparing across different slice sizes, we see that smaller slice sizes have a throughput advantage, because they lead to more accurate quantization of time. Having more time slices in a fixed scheduling interval offers more opportunities to adjust the flow assignment to the jobs. In Fig. 2-12, the throughput values at 16 paths per job is 9 for 10-min slice size and 6 for 60-min slice size. This shows the benefit of having a fine-grained slice size, since in this experimental setup, 16 paths are sufficient for S and SD schemes to reach the optimal throughput. We observed more significant throughput improvement from using smaller time slices in other settings. For instance, with 603 jobs, the throughput obtained from 10-min slice size is nearly twice the throughput from 60-min slice size.

Fig. 2-13 shows similar results for a 100-node random network with 100 jobs. The maximum throughput at 16 paths is nearly the same for all cases. However, for situations with a small number of paths per job, smaller time slice sizes have a throughput
advantage. More throughput improvement has been observed under other experimental settings. For instance, with 500 jobs and 16 paths, a 24% improvement is observed when using 10-minute slices instead of 60-minute slices.

Figure 2-12. Z for different formulations on the Abilene network with 121 jobs using MSS. A) Time slice = 60 min; B) Time slice = 30 min; C) Time slice = 15 min; D) Time slice = 10 min.

Figure 2-13. Z for different algorithms on a 100-node random network with 100 jobs using MSS. A) Time slice = 60 min; B) Time slice = 30 min; C) Time slice = 15 min; D) Time slice = 10 min.
2.5.2.2 Comparison of algorithm execution time

Fig. 2-14 and Fig. 2-15 show the execution time for the Abilene network with 121 jobs and for a 100-node random network with 100 jobs, respectively. For each fixed time slice size, we continue to observe the linear or faster increase of the execution time as the number of paths increase in the S and SD schemes. Again, the execution time for the node-arc form is much greater than that for the S and SD cases; in most cases, too large to be observed from our experiments. Finally, the throughput advantage of using smaller slice sizes is achieved at the expense of significant longer execution time.

Figure 2-14. Execution time for different formulations on the Abilene network with 121 jobs using MSS. A) Time slice = 60 min; B) Time slice = 30 min; C) Time slice = 15 min; D) Time slice = 10 min.

2.5.2.3 Optimal time slice

The tradeoff of the three scheduling algorithms lies in two metrics, throughput and execution time. Fig. 2-16 helps to identify a suitable time slice size for which the throughput is high and the execution time is acceptable. We observe that the throughput begins to saturate when the time slice size is 15 minutes and the execution time is under half a minute. Note the sharp rise of the execution time as the slice size decreases. It is therefore essential to choose an appropriate slice size.
Figure 2-15. Execution time for different formulations on a 100-node random network with 100 jobs using MSS. A) Time slice = 60 min; B) Time slice = 30 min; C) Time slice = 15 min; D) Time slice = 10 min.

Figure 2-16. The Abilene network with 121 jobs and $k = 8$. A) $Z$ for different time slices; B) Execution time for different time slice sizes.
CHAPTER 3
ADMISSION CONTROL AND SCHEDULING ALGORITHM

3.1 The Setup

For easy reference, notations and definitions frequently used in this chapter are summarized in Table 3-1. The notations for network and job requests are same as discussed in Section 2.1. In addition, a request from the MBG class is a 6-tuple \((A_i, s_i, d_i, B_i, S_i, E_i)\), where \(B_i\) is the requested minimum bandwidth on the interval \([S_i, E_i]\). It may optionally specify a maximum bandwidth. But, we will ignore this option in the presentation.

The network controller performs admission control (AC) by evaluating the available network capacity to satisfy new job requests. It admits only those jobs whose required performance can be guaranteed by the network and rejects the rest. The network controller also performs file transfer scheduling for all admitted jobs, which determines how each job is transferred over time, i.e., how much bandwidth is allocated to each path of the job at every time instance.

In the basic scheme, AC and scheduling are done periodically after every \(\tau\) time units, where \(\tau\) is a positive number. More specifically, at time instances \(k\tau, k = 1, 2, ...,\) the controller collects all the new requests that arrived on the interval \([(k-1)\tau, k\tau]\), makes the admission control decision, and schedules the transfer of all admitted jobs. Both AC and scheduling must take into account the old jobs, i.e., those jobs that were admitted earlier but remain unfinished. The value of \(\tau\) should be small enough so that new job requests can be checked for admission and scheduled as early as possible. However, \(\tau\) should be more than the computation time required for AC and scheduling.

\[\text{1 In this scheme, a request generally needs to wait a duration no longer than } \tau \text{ for the admission decision. We will comment on how to conduct realtime admission control later.}\]
Table 3-1. Frequently used notations and definitions

<table>
<thead>
<tr>
<th>Notation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$C_e$</td>
<td>Capacity of link $e$</td>
</tr>
<tr>
<td>$D_i$</td>
<td>Demand size of job $i$</td>
</tr>
<tr>
<td>$S_i, \hat{S}_i$</td>
<td>Start time and rounded start time of job $i$</td>
</tr>
<tr>
<td>$E_i, \hat{E}_i$</td>
<td>End time and rounded end time of job $i$</td>
</tr>
<tr>
<td>$\tau$</td>
<td>Interval between consecutive AC/scheduling runs</td>
</tr>
</tbody>
</table>

In the following, assume $t = k\tau$.

- $G_k$: Slice set anchored at time $k\tau$
- $M_k$: Index of the last slice in which some rounded end time falls
- $L_k \subset G_k$: Finite slice set 1, ..., $M_k$
- $ST_k(i), ET_k(i)$: Start and end times of slice $i$
- $LEN_k(i)$: Length of slice $i$
- $I_k(t)$: Index of the slice that time $t$ falls in
- $J^o_k$: Set of the old jobs
- $J^n_k$: Set of the new jobs
- $P_k(s,d)$: Allowable paths from node $s$ to $d$
- $R_k(i)$: Remaining demand of job $i$
- $f_i(p,j)$: Total flow allocated to job $i$ on path $p$ on slice $j$
- $C_e(j)$: Remaining capacity of link $e$ on slice $j$

### 3.2 The Time Slice Structure

At each scheduling instance, $t = k\tau$, the timeline from $t$ onward is partitioned into time slices, i.e., closed intervals on the timeline, which are not necessarily uniform in size.

A set of time slices, $G_k$, is said to be anchored at $t = k\tau$ if all slices in $G_k$ are mutually disjoint and their union forms an interval $[t, t']$ for some $t'$. The set $\{G_k\}_{k=1}^\infty$ is called a slice structure if each $G_k$ is a set of slices anchored at $t = k\tau$, for $k = 1, ..., \infty$.

**Definition 1.** A slice structure $\{G_k\}_{k=1}^\infty$ is said to be congruent if the following property is satisfied for every pair of positive integers, $k$ and $k'$, where $k' > k \geq 1$. For any slice $s' \in G_{k'}$, if $s'$ overlaps in time with a slice $s$, $s \in G_k$, then $s' \subseteq s$.

In words, any slice in a later anchored slice collection must be completely contained in a slice of any earlier collection, if it overlaps in time with the earlier collection.

Alternatively speaking, if slice $s \in G_k$ overlaps in time with $G_{k'}$, then either $s \in G_{k'}$ or $s$ is partitioned into multiple slices all belonging to $G_{k'}$. 

43
The motivation for the definition of the congruent slice structure will become more clear later. In a nutshell, the AC and scheduling algorithm introduced in this thesis applies to any congruent slice structure; the congruent slice structure is the key construct that allows us to guarantee the performance of old jobs admitted previously while admitting new jobs, when a non-uniform slice structure is used. In this thesis, we focus on two simple congruent slice structures, the uniform slices (US) and the nested slices (NS), as shown in Fig. 3-1 and 3-3, respectively. For ease of presentation, we use the uniform slices as an example to explain the AC and scheduling algorithm. Discussion on the more sophisticated nested slices is deferred to Section 3.6.

In US, the timeline is divided into equal-sized time slices of duration \( \tau \) (coinciding with the AC/scheduling interval length). The set of slices anchored at any \( t = k \tau \) is all the slices after \( t \). Figure 3-1 shows the uniform slice structure at two time instances \( t = \tau \) and \( t = 2\tau \). In this example, \( \tau = 4 \) time units. The arrows point to the scheduling instances. The two collections of rectangles are the time slices anchored at \( t = \tau \) and \( t = 2\tau \), respectively. It is easy to check the congruent property of this slice structure.

![Uniform Slices](image)

**Figure 3-1. Uniform time slice structure**

At any AC/scheduling time \( t = k \tau \), let the time slices anchored at \( t \), i.e., those in \( G_k \), be indexed 1, 2, ... in increasing order of time. Let the start and end times of slice \( i \) be denoted by \( ST_k(i) \) and \( ET_k(i) \), respectively, and let its length be \( LEN_k(i) \). We say a time instance \( t' > t \) falls into slice \( i \) if \( ST_k(i) < t' \leq ET_k(i) \). The index of the slice that \( t' \) falls in is denoted by \( I_k(t') \).
At $t = k\tau$, let the set of jobs in the system yet to be completed be denoted by $J_k$. $J_k$ contains two types of jobs, those new requests (also known as new jobs) made on the interval $((k-1)\tau, k\tau]$, denoted by $J_k^n$, and those old jobs admitted at or before $(k-1)\tau$, denoted by $J_k^o$. The old jobs have already been admitted and should not be rejected by the admission control conducted at $t$. But some of the new requests may be rejected.

**Rounding of the start and end times.** With the time slice structure and the advancement of time, we adjust the start and end times of the requests. The main objective is to align the start and end times on the slice boundaries. After such rounding, the start and the end times will be denoted as $\hat{S}_i$ and $\hat{E}_i$, respectively. For a new request $i$, let the requested response time be $T_i = E_i - S_i$. We round the requested start time to be the maximum of the current time or the end time of the slice in which the requested start time $S_i$ falls, i.e.,

$$\hat{S}_i = \max\{t, ET_k(I_k(S_i))\}. \quad (3.1)$$

For rounding of the requested end time, we allow two policy choices, the **stringent policy** and the **relaxed policy**. In the stringent policy, if the requested end time does not coincide with a slice boundary, it is rounded down, subject to the constraint that $\hat{E}_i > \hat{S}_i$. This constraint ensures that there is at least one-slice separation between the rounded start time and the rounded end time. Otherwise, there is no way to schedule the job. In the relaxed policy, the end time is first shifted by $T_i$ with respect to the rounded start time, and then rounded up. More specifically,

\[\text{In the more sophisticated non-uniform slice structure introduced in Section 3.6, we allow the end time to be re-rounded at different scheduling instances. This way, the rounded end time can become closer to the requested end time, as the slice sizes become finer over time.}\]
stringent

\[ \hat{E}_i = \begin{cases} 
ET_k(I_k(\hat{S}_i) + 1) & \text{if } ST_k(I_k(E_i)) \leq \hat{S}_i \\
E_i & \text{else if } ET_k(I_k(E_i)) = E_i \\
ST_k(I_k(E_i)) & \text{otherwise.}
\end{cases} \]

\( (3.2) \)

relaxed

\[ \hat{E}_i = ET_k(I_k(\hat{S}_i + T_i)) \]

Figure 3-2 shows the effect of the two policies after three jobs are rounded.

![Figure 3-2](image)

Figure 3-2. Two rounding policies. The unshaded rectangles are time slices. The shaded rectangles represent jobs. The top ones show the requested start and end times. The bottom ones show the rounded start and end times.

If a job \( i \) is an old one, its rounded start time \( \hat{S}_i \) is replaced by the current time \( t \). The remaining demand is updated by subtracting from it the total amount of data transferred for job \( i \) on the previous interval, \( ((k - 1)\tau, k\tau] \).

From the definition of uniform slices, the slice set anchored at each \( t = k\tau, \mathcal{G}_k \), contains an infinite number of slices. In general, only a finite subset of \( \mathcal{G}_k \) is useful to us. Let \( M_k \) be the index of the last slice in which the rounded end time of some jobs falls. That is, \( M_k = I_k(\max_{i \in \mathcal{J}_k} \hat{E}_i) \). Let \( \mathcal{L}_k \subset \mathcal{G}_k \) be the collection of time slices 1, 2, ..., \( M_k \). We call the slices in \( \mathcal{L}_k \) as the active time slices. We will also think of \( \mathcal{L}_k \) as
an array of slices when there is no ambiguity. Clearly, the collection \( \{\mathcal{L}_k\}_{k=1}^{\infty} \) inherits the congruent property from \( \{\mathcal{G}_k\}_{k=1}^{\infty} \). Therefore, it is sufficient to consider \( \{\mathcal{L}_k\}_{k=1}^{\infty} \) for AC and scheduling.

### 3.3 Admission Control

For each pair of nodes \( s \) and \( d \), let the collection of allowable paths from \( s \) to \( d \) be denoted by \( P_k(s,d) \). In general, the set may vary with \( k \). For each job \( i \), let the remaining demand at time \( t = k\tau \) be denoted by \( R_k(i) \), which is equal to the total demand \( D_i \) minus the amount of data transferred till time \( t \).

At \( t = k\tau \), let \( J \subseteq J_k \) be a subset of the jobs in the systems. Let \( f_i(p,j) \) be the total flow (total data transfer) allocated to job \( i \) on path \( p \), where \( p \in P_k(s_i,d_i) \), on time slice \( j \), where \( j \in \mathcal{L}_k \). As part of the admission control algorithm, the solution to the following feasibility problem is used to determine whether the jobs in \( J \) can all be admitted.

\[
\text{AC}(k, J) \\
\sum_{j=1}^{M_k} \sum_{p \in P_k(s_i,d_i)} f_i(p,j) = R_k(i), \quad \forall i \in J \tag{3.3}
\]

\[
\sum_{i \in J} \sum_{p \in P_k(s_i,d_i)} f_i(p,j) \leq C_e(j)L\text{EN}_k(j), \quad \forall e \in E, \forall j \in \mathcal{L}_k \tag{3.4}
\]

\[
f_i(p,j) = 0, \quad j \leq I_k(\hat{S}_i) \text{ or } j > I_k(\hat{E}_i), \\
\forall i \in J, \forall p \in P_k(s_i,d_i) \tag{3.5}
\]

\[
f_i(p,j) \geq 0, \quad \forall i \in J, \forall j \in \mathcal{L}_k, \forall p \in P_k(s_i,d_i). \tag{3.6}
\]

(3.3) says that, for every job, the sum of all the flows assigned on all time slices for all paths must be equal to its remaining demand. (3.4) says that the capacity constraints must be satisfied for all edges on every time slice. Note that the allocated rate on path \( p \) for job \( i \) on slice \( j \) is \( f_i(p,j)/L\text{EN}_k(j) \), where \( L\text{EN}_k(j) \) is the length of slice \( j \). The rate is assumed to be constant on the entire slice. Here, \( C_e(j) \) is the remaining link capacity.
of link $e$ on slice $j$. (3.5) is the start and end time constraint for every job on every path. The flow must be zero before the rounded start time and after the rounded end time.

Recall that we are assuming every job to be a bulk transfer for simplicity. If job $i$ is of the MBG class, then the the remaining capacity constraint (3.3) will be replaced by a minimum bandwidth guarantee condition.

\[ \sum_{p \in P_{k}(s_i, d_i)} f_i(p, j) \geq B_i, \quad \forall j \in \mathcal{L}_k. \] (3.7)

The AC/scheduling algorithm is triggered every $\tau$ time units with the AC part before the scheduling part. AC examines the newly arrived jobs and determines their admissibility. In doing so, we need to ensure that the earlier commitments to the old jobs are not broken. This can be achieved by adopting one of the following AC procedures.

1. **Subtract-Resource (SR)**: An updated (remaining) network is obtained by subtracting the bandwidth assigned to old jobs on future time slices, from the link capacity. Then, we determine a subset of the new jobs that can be accommodated in this remaining network. This method is helpful to perform quick admission tests. However, it runs the risk of rejecting new jobs that can actually be accommodated by reassigning the flows to the old jobs on different paths and time slices.

2. **Reassign-Resource (RR)**: This method attempts to reassign flows to the old jobs. First, we cancel the existing flow assignment to the old jobs on the future time slices and restore the network to its original capacity. Then, we determine a subset of the new jobs that can be admitted along with all the old jobs under the original network capacity. This method is expected to have a better acceptance ratio than SR. However, it is computationally more expensive because the flow assignment is computed for all the jobs in the system, both the old and the new.

\[ ^3 \text{We can perform realtime admission with this method.} \]
The actual admission control is as follows. In the SR scheme, the remaining capacity
of link \( e \) on slice \( j \), \( C_e(j) \), is computed by subtracting from \( C_e \) (the original link capacity),
the total bandwidth allocated on slice \( j \) for all paths crossing \( e \), during the previous run of
the AC/scheduling algorithm (at \( t = (k - 1)\tau \)). In the RR scheme, simply let \( C_e(j) = C_e \),
for all \( e \) and \( j \).

In the SR scheme, we list the new jobs, \( J^u_k \), in a sequence, \( 1, 2, \ldots, m \). The particular
order of the sequence is flexible, possibly dependent on some customizable policy. For
instance, the order may be arbitrary, or based on the priority the jobs or based on
increasing order of the request times. We apply a binary search to the sequence to find
the last job \( j, 1 \leq j \leq m \), in the sequence such that all jobs before and including
it are admissible. That is, \( j \) is the largest index for which the subset of the new jobs
\( J = \{1, 2, \ldots, j\} \) is feasible for AC(\( k, J \)). All the jobs after \( j \) are rejected.

In the RR scheme, at time \( t = k\tau \), all the jobs are listed in a sequence where the
old jobs \( J^o_k \) are ahead of the new jobs \( J^u_k \) in the sequence. The order among the old jobs
is arbitrary. The order among the new jobs is again flexible. Denote this sequence as
\( 1, 2, \ldots, m \), in which jobs 1 through \( l \) are the old ones. We then apply a binary search to
the sequence of new jobs, \( l + 1, l + 2, \ldots, m \), to find the last job \( j, l < j \leq m \), such that
all jobs before and including it are admissible. That is, \( j \) is the largest index for which the
resulting subset of the jobs \( J = \{1, 2, \ldots, l, l + 1, \ldots, j\} \) is feasible for AC(\( k, J \)) under the
original network capacity.

**Discussion.** The binary search technique assumes a pre-defined list of jobs and
identifies the first \( j \) jobs that can be admitted into the system without violating the
deadline constraints. The presence of an exceptionally large job with unsatisfiable
demand will cause other jobs following it to be rejected, even though it may be possible
to accommodate them after removing the large job. The rejection ratio tends to be higher
when the large job lies closer to the head of the list. An interesting problem is how to
admit as many new jobs as possible, after all the old jobs are admitted. A solution to this
problem is orthogonal to the main issues addressed in this thesis, but can be incorporated into our general scheduling framework.

### 3.4 Scheduling Algorithm

Given the set of admitted jobs, $J^a_k$, which always includes the old jobs, the scheduling algorithm allocates flows to these jobs to optimize a certain objective. We consider two objectives, **Quick-Finish** (QF) and **Load-Balancing** (LB). Given a set of admissible jobs $J$, the problem associated with the former is

\[
\text{Quick-Finish}(k, J) \quad \text{min} \quad \sum_{j \in L_k} \gamma(j) \sum_{i \in J} \sum_{p \in P_k(s_i, d_i)} f_i(p, j) \\
\text{subject to (3.3) – (3.6)}
\]

In the above, $\gamma(j)$ is a weight function increasing in $j$, which is chosen to be $\gamma(j) = j + 1$ in our experiments. In this problem, the cost increases as time increases. The intention is to finish a job early rather than later, when it is possible. The solution tends to pack more flows in the earlier slices but leaves the load light in later slices. The problem associated with the LB objective is,

\[
\text{Load-Balancing}(k, J) \quad \text{max} \quad Z \\
\text{subject to} \quad \sum_{j=1}^{M_k} \sum_{p \in P_k(s_i, d_i)} f_i(p, j) = Z R_k(i), \quad \forall i \in J \\
(3.4) – (3.6)
\]

Let the optimal solution be $Z^*$ and $f_i^*(p, j)$ for all $i$, $j$, and $p$. The actual flows assigned are $f_i^*(p, j)/Z^*$. Note that (3.10) ensures that $f_i^*(p, j)/Z^*$’s satisfy (3.3). Also, $Z^* \geq 1$ must be true since $J$ is admissible. Hence, $f_i^*(p, j)/Z^*$’s are a feasible solution to the **AC**($k$, $J$) problem. The **Load-Balancing**($k$, $J$) problem above is written in the
maximizing concurrent throughput form. It reveals its load-balancing nature when written in the equivalent minimizing congestion form. For that, make a substitution of variables, \( f_i(p, j) \leftarrow f_i(p, j)/Z \), and let \( \mu = 1/Z \).

We have,

\[
\text{Load-Balancing-1}(k, J)
\]

\[
\min \mu 
\]

subject to \( \sum_{i \in J} \sum_{p \in P_k(s_i, d_i)} f_i(p, j) \leq \mu C_e(j) LEN_k(j), \)

\( \forall e \in E, \forall j \in L_k \)

(3.12)

(3.3), (3.5) and (3.6).

Hence, the solution minimizes the worst link congestion across all time slices in \( L_k \).

The scheduling algorithm is to apply \( J = J^a_k \) to \text{Quick-Finish}(k, J) or \text{Load-Balancing}(k, J). This determines an optimal flow assignment to all jobs on all allowed paths and on all time slices. Given the flow assignment \( f_i(p, j) \), the allocated rate on each time slice is denoted by \( x_i(p, j) = f_i(p, j)/LEN_k(j) \) for all \( j \in L_k \). The remaining capacity of each link on each time slice is given by,

\[
C_e(j) = \begin{cases} 
C_e - \sum_{i \in J_k} \sum_{p \in P_k(s_i, d_i)} x_i(p, j) & \text{if } \text{SR} \\
C_e & \text{if } \text{RR}.
\end{cases}
\]

(3.13)

### 3.5 Putting It Together: The AC and Scheduling Algorithm

In this section, we integrate various algorithmic components and present the complete AC and scheduling algorithm.

On the interval \(((k-1)\tau, k\tau] \), the system keeps track of the new requests arriving on that interval. It also keeps track of the status of the old jobs. If an old job is completed, it is removed from the system. If an old job is serviced on the interval, the amount of data
transferred for that job is recorded. At \( t = k\tau \), the steps described in Algorithm 1 are taken.

**Algorithm 1** Admission Control and Scheduling

1: Construct the anchored slice set at \( t = k\tau \), \( G_k \).
2: Construct the job sets \( J_k, J^o_k \) and \( J^a_k \), which are the collection of all jobs, the collection of old jobs, and the collection of new jobs in the system, respectively.
3: For each old job \( i \), update the remaining demand \( R_k(i) \) by subtracting from it the amount of data transferred for \( i \) on the interval \(( (k - 1)\tau, k\tau ]\). Round the start times as \( \hat{S}_i = t \).
4: For each new job \( l \), let \( R_k(l) = D_l \). Round the requested start and end time according to (3.1) and (3.2), depending on whether the stringent or relaxed rounding policy is used. This produces the rounded start and end times, \( \hat{S}_l \) and \( \hat{E}_l \).
5: Derive \( M_k = I_k(\max_{i \in J_k} \hat{E}_i) \). This determines the finite collection of slices \( L_k = \{1, 2, ..., M_k\} \), the first \( M_k \) slices of \( G_k \).
6: Perform admission control as in Algorithm 2. This produces the list of admitted jobs \( J^a_k \).
7: Schedule the admitted jobs as in Algorithm 3. This yields the flow amount \( f_i(p, j) \) for each admitted job \( i \in J^a_k \), over all paths for job \( i \), and all time slices \( j \in L_k \).
8: Compute the remaining network capacity by (3.13).

**Algorithm 2** AC - Step 6 of Algorithm 1

1: if Subtract-Resource is used then
2: Sequence the new jobs \( J^a_k \) in the system. Denote the sequence by \( (1, 2, ..., m) \).
3: Find the last job \( j \) in the sequence so that the set of jobs \( J = \{1, 2, ..., j\} \) is admissible by \( AC(k, J) \).
4: else if Reassign-Resource is used then
5: Sequence all the jobs \( J_k \) in the system, so that the old jobs \( J^o_k \) are ahead of the new jobs \( J^a_k \). Denote the sequence of jobs by \( (1, 2, ..., l, l + 1, ..., m) \), where the first \( l \) jobs are the old jobs, followed by the new jobs.
6: Apply binary search to the subsequence of new jobs \( (l + 1, l + 2, ..., m) \). Find the last job \( j \) in the subsequence so that the set of jobs \( J = \{1, 2, ..., j\} \) is admissible by \( AC(k, J) \).
7: end if
8: Return the admissible set, \( J^a_k = J \).

### 3.6 Non-uniform Slice Structure

The number of time slices directly affect the number of variables in our AC and scheduling linear programs, and in turn the execution speed of our algorithm. We face a problem of covering a large enough segment of the timeline for advance reservations with
Algorithm 3 Scheduling - Step 7 of Algorithm 1

1: if Quick-Finish is preferred then
2: Run Quick-Finish\( (k, J^a_k) \)
3: else
4: Run Load-Balancing\( (k, J^a_k) \)
5: end if

a small number of slices, say about 100. In order to cover a 30-day reservation period with 100 slices, the slice size in the US structure is 7.2 hours, too coarse for small to medium sized jobs. In this section, we will design a new slice structure with non-uniform slice sizes. They contain a geometrically increasing subsequence, and therefore, are able to cover a large timeline with a small number of slices. The challenge is that the slice structure must remain congruent.

Recall that the congruent property means that, if a slice in an earlier anchored slice set overlaps in time with a later anchored slice set, it either remains as a slice, or is partitioned into smaller slices in the later slice set. The definition is motivated by the need for maintaining consistency in bandwidth assignment across time. As an example, suppose at time \((k - 1)\tau\), a job is assigned a bandwidth \(x\) on a path on the slice \(j_{k-1}\). At the next scheduling instance \(t = k\tau\), suppose the slice \(j_{k-1}\) is partitioned into two slices. Then, we understand that a bandwidth \(x\) has been assigned on both slices. Without the congruent property, it is likely that a slice, say \(j_k\), in the slice set anchored at \(k\tau\) cuts across several slices in the slice set anchored at \((k - 1)\tau\). If the bandwidth assignments at \((k - 1)\tau\) are different for these latter slices, the bandwidth assignment for slice \(j_k\) is not well defined just before the AC/scheduling run at time \(k\tau\).

3.6.1 Nested Slice Structure

In the nested slice structure, there are \(l\) types of slices, known as level-\(i\) slices, \(i = 1, 2, ..., l\). Each level-\(i\) slice has a duration \(\Delta_i\), with the property that \(\Delta_i = \kappa_i \Delta_{i+1}\), where \(\kappa_i > 1\) is an integer, for \(i = 1, ..., l - 1\). Hence, the slice size increases at least geometrically as \(i\) decreases. For practical applications, a small number of levels suffices. We also require that, for \(i\) such that \(\Delta_{i+1} \leq \tau < \Delta_i\), \(\tau\) is an integer multiple of \(\Delta_{i+1}\).
\[ \Delta_i \] is an integer multiple of \( \tau \). This ensures that each scheduling interval contains an integral number of slices and that the sequence of scheduling instances does not skip any level-\( j \) slice boundaries, for \( 1 \leq j \leq i \).

The nested slice structure can be defined by construction. At \( t = 0 \), the timeline is partitioned into level-1 slices. The first \( j_1 \) level-1 slices, where \( j_1 \geq 1 \), are each partitioned into level-2 slices. This removes \( j_1 \) level-1 slices but adds \( j_1 \kappa_1 \) level-2 slices. Next, the first \( j_2 \) level-2 slices, where \( j_2 \leq j_1 \kappa_1 \), are each partitioned into level-3 slices. This removes \( j_2 \) level-2 slices but adds \( j_2 \kappa_2 \) level-3 slices. This process continues until, in the last step, the first \( j_{l-1} \) level-(\( l-1 \)) slices are partitioned into \( j_{l-1} \kappa_{l-1} \) level-\( l \) slices. Then, the first \( j_{l-1} \) level-(\( l-1 \)) slices are removed and \( j_{l-1} \kappa_{l-1} \) level-\( l \) slices are added at the beginning. In the end, the collection of slices at \( t = 0 \) contains \( \sigma_l \triangleq j_{l-1} \kappa_{l-1} \) level-\( l \) slices, \( \sigma_{l-1} \triangleq j_{l-2} \kappa_{l-2} - j_{l-1} \) level-(\( l-1 \)) slices, ..., \( \sigma_2 \triangleq j_1 \kappa_1 - j_2 \) level-2 slices, and followed by an infinite number of level-1 slices. The sequence of \( j_i \)'s must satisfy \( j_2 \leq j_1 \kappa_1, j_3 \leq j_2 \kappa_2, ..., j_{l-1} \leq j_{l-2} \kappa_{l-2} \). This collection of slices is denoted by \( G_0 \).

As an example, to cover a maximum of 30-day period, we can take \( \Delta_1 = 1 \) day, \( \Delta_2 = 1 \) hour, and \( \Delta_3 = 10 \) minutes. Hence, \( \kappa_1 = 24 \) and \( \kappa_2 = 6 \). The first two days are first divided into a total 48 one-hour slices, out of which the first 8 hours are further divided into 48 10-minute slices. The final slice structure has 48 level-3 (10-minute) slices, 40 level-2 (one-hour) slices, and as many level-1 (one-day) slices as needed, in this case, 28. The total number of slices is 116.

In designing the slice structure, sometimes one wishes to begin with specifying the set of \( \sigma_j \)'s. To have a nested slice structure, the \( \sigma_j \)'s should satisfy the following property. First, \( \lambda_l \triangleq \sigma_l \) is an integer multiple of \( \kappa_{l-1} \) and \( \lambda_{l-1} \triangleq \lambda_l / \kappa_{l-1} + \sigma_{l-1} \) is an integer multiple of \( \kappa_{l-2} \). In general, for \( i \) from \( l-1 \) down to 2, define \( \lambda_i \triangleq \lambda_{i+1} / \kappa_i + \sigma_i \) \(^4\). \( \lambda_i \) should be an

\(^4\) For each \( i, 2 \leq i \leq l \), \( \lambda_i \) has the meaning that the length of the portion of the timeline covered by level-\( j \) slices, for all \( i \leq j \leq l \), is equivalent to the length of \( \lambda_i \) level-\( i \) slices.
integer multiple of $\kappa_{i-1}$. The $\sigma_i$'s can be determined one by one in decreasing order of $i$. In the previous example, we can first choose $\sigma_3 = 48$ since it is a multiple of $\kappa_2 = 6$. This gives $\lambda_2 = 48/6 + \sigma_2$. If we choose $\sigma_2 = 40$, then $\lambda_2 = 48$ is divisible by $\kappa_1 = 24$.

For the subsequent scheduling instances, the objective is to maintain the same number of slices as $\mathcal{G}_0$ at different levels. But this cannot be done while satisfying the slice congruent property. Hence, we allow the number of slices at each level to deviate from $\sigma_j$, for $j = 2, ..., l$. This can be done in various ways. Let $z_j$ be the current number of level-$j$ slices at $t = k\tau$, for $j = 1, 2, ..., l$. Set $z_1 = \infty$.

1. **At-Least-$\sigma$**: For $j$ from $l$ down to 2, if the number of slices at level $j$, $z_j$, is less than $\sigma_j$, bring in (and remove) the next level-$(j-1)$ slice and partition it into $\kappa_{j-1}$ level-$j$ slices. This scheme maintains at least $\sigma_j$ and at most $\sigma_j + \kappa_{j-1} - 1$ level-$j$ slices for $j = 2, ..., l$.

2. **At-Most-$\sigma$**: In this scheme, we try to bring the current number of slices at level $j$, $z_j$, to $\sigma_j$, for $j = 2, ..., l$, subject to the constraint that new slices at level $j$ can only be created if $t$ is an integer multiple of $\Delta_{j-1}$.

More specifically, at $t = k\tau$, the following is repeated for $j$ from $l$ down to 2. If $t$ is not an integer multiple of $\Delta_{j-1}$, then nothing is done. Otherwise, if $z_j < \sigma_j$, we try to create level-$j$ slices out of a level-$(j-1)$ slice. In the creation process, if a level-$(j-1)$ slice exists, then bring in the first one and partition it. Otherwise, we try to create more level-$(j-1)$ slices, provided $t$ is an integer multiple of $\Delta_{j-2}$. Hence, a recursive slice-creation process may be involved. This procedure is made more concrete in Algorithm 4, which calls Algorithm 5, a recursive subroutine.

Fig. 3-3 and 3-4 show a two-level and three-level nested slice structure, respectively, under the At-Most-$\sigma$ design. In the special but typical case of $\sigma_j > \kappa_{j-1}$, for $j = 2, ..., l$, the At-Most-$\sigma$ algorithm can be simplified as follows. For $j$ from $l$ down to 2, if $z_j \leq \sigma_j - \kappa_{j-1}$, bring in (and remove) the next level-$(j-1)$ slice and partition it into $\kappa_{j-1}$ level-$j$ slices. This scheme maintains at least $\sigma_j - \kappa_{j-1}$ and at most $\sigma_j$ level-$j$ slices for $j = 2, ..., l$. 

55
Algorithm 4 At-Most-σ

1: for \( j = l \) down to 2 do
2:   if \( t \) is an integer multiple of \( \Delta_{j-1} \) then
3:     while \( z_j < \sigma_j \) do
4:       \( w_j \leftarrow z_j \) // remember \( z_j \)
5:       Create-Slices \((j)\)
6:     if \((z_j = w_j)\) then
7:       break // New slices cannot be created.
8:   end if
9: end while
10: end if
11: end for

Algorithm 5 Create-Slices \((j)\)

1: if \( z_{j-1} < 1 \) and \( j > 2 \) and \( t \) is an integer multiple of \( \Delta_{j-2} \) then
2:   Create-Slices \((j-1)\)
3: end if
4: if \( z_{j-1} \geq 1 \) then
5:   // The next level-(\(j-1\)) slice exists.
6:   Bring in the next level-(\(j-1\)) slice and partition it into \( \kappa_{j-1} \) level-\(j\) slices.
7:   \( z_j \leftarrow z_j + \kappa_{j-1} \)
8:   \( z_{j-1} \leftarrow z_{j-1} - 1 \)
9: end if

3.6.2 Variant of Nested Slice Structure

When some \( \kappa_j \) is large, it may be unappealing that the number of level-\(j\) slices varies by \( \kappa_{j-1} \) (sometimes more than \( \kappa_{j-1} \)). To solve this problem, we next introduce another

![Nested Slices Diagram](image)

Figure 3-3. Two-level nested time-slice structure. \( \tau = 2, \Delta_1 = 4 \) and \( \Delta_2 = 1 \). The anchored slice sets shown are for \( t = \tau, 2\tau \) and \( 3\tau \), respectively. At-Most-σ Design. \( \sigma_2 = 8 \).
Nested Slices

Figure 3-4. Three-level nested time-slice structure. \( \tau = 2, \Delta_1 = 16, \Delta_2 = 4 \) and \( \Delta_3 = 1 \).

The anchored slice sets shown are for \( t = \tau, 2\tau \) and \( 8\tau \), respectively. At-Most-\( \sigma \) Design. \( \sigma_3 = 8, \sigma_2 = 2 \).

The congruence slice structure related to the nested slice structure. We will call it the

**Almost-\( \sigma \) Variant** of the nested slice structure, because it maintains at least \( \sigma_j \) and at most \( \sigma_j + 1 \) level-\( j \) slices for \( j = 2, ..., l \).

The Almost-\( \sigma \) Variant starts the same way as the nested slice structure at \( t = 0 \). As time progresses from \( (k - 1)\tau \) to \( k\tau \), for \( k = 1, 2, ..., \), the collection of slices anchored at \( t = k\tau \), i.e., \( G_k \), is updated from \( G_{k-1} \) as in algorithm 6.

**Algorithm 6** Almost-\( \sigma \)-Variant

1: for \( j = l \) down to 2 do
2:  if \( z_j < \sigma_j \) then
3:      Bring in (and remove) the next available slice of a larger size and create additional \( \sigma_j - z_j \) level-\( j \) slices.
4:  \( z_j \leftarrow \sigma_j \).
5:  The remaining portion of the removed level-\( (j - 1) \) slice forms another slice.
6:  end if
7: end for

The price to pay is that the Almost-\( \sigma \) Variant introduces new slice types different from the pre-defined level-\( i \) slices, for \( i = 1, ..., l \). Fig. 3-5 shows a three-level Almost-\( \sigma \) Variant.

### 3.7 Evaluation

This section shows the performance results of different variations of our AC/scheduling algorithm. We also evaluate the required computation time to determine the scalability of our algorithms.
Figure 3-5. Three-level nested slice structure Almost-σ Variant. $\tau = 2$, $\Delta_1 = 16$, $\Delta_2 = 4$ and $\Delta_3 = 1$. The anchored slice sets shown are for $t = \tau$, $2\tau$ and $3\tau$, respectively. $\sigma_3 = 8$, $\sigma_2 = 2$. The shaded areas are also slices, but are different in size from any level-$j$ slice, $j = 1, 2$ or 3.

Most of the experiments are conducted on the Abilene network, which consists of 11 backbone nodes connected by 10 Gbps links. Each backbone node is connected to a randomly generated stub network. The link speed between each stub network and the backbone node is 1 Gbps. The entire network has 121 nodes and 490 links. For the scalability study of the algorithms, we use random networks with nodes ranging from 100 to 1000. We use the commercial CPLEX package for solving linear programs on Intel-based workstations.

Unless mentioned otherwise, we use the following experimental models and parameters. Job requests arrive following a Poisson process. In order to simulate the file size distribution of Internet traffic, we resort to the widely accepted heavy-tailed Pareto distribution, with the distribution function $F(x) = 1 - (x/b)^{-\alpha}$, where $x \geq b$ and $\alpha > 1$. The closer $\alpha$ is to 1, the more heavy-tailed is the distribution, and it is more likely to generate very large demand sizes. In most of our experiments, the average file size is 50 GB and $\alpha = 1.3$. By default, each job uses 8 shortest paths. We adopt this approach because our experiments on multi path scheduling revealed the following significant result; for a network of size several hundred nodes, 8 shortest paths are sufficient to achieve near
optimal solutions under practical execution time. We evaluate our algorithms under 3 traffic loads, namely, light, medium and heavy. By light, medium and heavy traffic loads, we mean that the average inter-arrival time between jobs is 5 minutes, 2 minutes and 30 seconds, respectively. In order to get stable results, we generated jobs under these different traffic loads for a period of 3 days. For example, under the heavy traffic load, roughly 10,000 file transfer requests were generated.

We will compare the uniform time slice (US) and the nested slice structure (NS) of the Almost-\(\sigma\) Variant type. For US, the time slice and AC/scheduling interval \((\tau)\) is 21.17 minutes. This corresponds to 68 slices in every 24-hour period. For NS, we use a two-level NS structure with 48 fine (level-2) slices and 20 coarse (level-1) slices. The fine slice size is \(\Delta_2 = 5\) minutes, and the coarse slice size is \(\Delta_1 = 60\) minutes. These parameters are chosen so that the first 24-hour period is divided into 68 fine and coarse slices, the same number as the US case. The AC/scheduling interval \(\tau\) is 5 minutes, which is finer than the US case.

The plots and tables use acronyms to denote the algorithms used in the experiments. Recall that SR stands for Subtract-Resource and RR stands for Reassign-Resource in admission control; LB stands for Load-Balancing as the scheduling objective and QF stands for Quick-Finish.

The performance measures are,

• Rejection ratio: This is the ratio between the number of jobs rejected and total number of job requests. From the system’s perspective, it is desirable to admit as many jobs as possible.

• Response time: This is the difference between the completion time of a job and the time when it is first being transmitted. From an individual job’s perspective, it is desirable to have shorter response time.

While configuring the simulation environment, we can ignore the connection setup (path setup) time because the small network size allows us to pre-compute the allowed paths for every possible request.
3.7.1 Comparison of Algorithm Execution Time

Before comparing the performance of the algorithms, we first compare their execution time. Short execution time is important for the practicality of our centralized network control strategy. The results on execution time put the performance comparison (Section 3.7.2) in perspective: better performance often comes with longer execution time. Table 3-2 shows the execution time of different schemes under two representative traffic conditions.

Table 3-2. Average admission control/scheduling algorithm execution time (s)

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Heavy Load</th>
<th>Light Load</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>AC</td>
<td>Scheduling</td>
</tr>
<tr>
<td>US+SR+LB</td>
<td>13.13</td>
<td>5.70</td>
</tr>
<tr>
<td>US+SR+QF</td>
<td>12.03</td>
<td>1.86</td>
</tr>
<tr>
<td>US+RR+LB</td>
<td>80.89</td>
<td>5.89</td>
</tr>
<tr>
<td>US+RR+QF</td>
<td>34.36</td>
<td>4.74</td>
</tr>
<tr>
<td>NS+SR+LB</td>
<td>1.54</td>
<td>4.50</td>
</tr>
<tr>
<td>NS+SR+QF</td>
<td>1.57</td>
<td>1.60</td>
</tr>
<tr>
<td>NS+RR+LB</td>
<td>25.16</td>
<td>4.30</td>
</tr>
<tr>
<td>NS+RR+QF</td>
<td>17.43</td>
<td>3.54</td>
</tr>
</tbody>
</table>

SR vs. RR and LB vs. QF. The results show that for admission control, SR can have much smaller average execution time than RR. This is because, in SR, AC works only on the new jobs, whereas in RR, AC works on all the jobs currently in the system. Hence, for SR, the AC(k, J) feasibility problem has much fewer variables.

When the AC algorithm is fixed, the choice of the scheduling algorithm, LB or QF, also affects the execution time for AC. For instance, the RR+LB combination has much longer execution time for AC than the RR+QF combination. This is because, in LB, the flow for each job tends to be stretched over time in an effort to reduce the network load on each time slice. This results in more jobs and more active slices (slices in \( L_k \)) in the system at any moment, which mean more variables for the linear program.
For scheduling, since LB and QF are very different linear programs, it is difficult to explain their execution times. But, we do observe that LB has longer execution time, again, possibly due to more variables for the reason stated in the previous paragraph.

**US vs. NS.** Depending on the number of levels of the nested slice structure, the number of slices at each level and the slice sizes, the NS can be configured to achieve different objectives; improving the algorithm performance, reducing the execution time, or doing both simultaneously. Our experimental results in Table 3-2 correspond to the third case. Since the two-level NS structure has $\Delta_1 = 60$ minutes and the US has the uniform slice size $\Delta = 21.17$ minutes, the NS typically has fewer slices than the US. For instance, under heavy load, US+RR+QF uses 150.5 active slices on an average for AC, while NS+RR+QF uses 129.6 active slices on an average. The number of variables, which directly affect the computation time of the linear programs, is generally proportional to the number of slices.

Part of the performance advantage of NS (this is shown in Section 3.7.2 later.) is attributed to the smaller scheduling interval $\tau$. To reduce the scheduling interval for US, we must reduce the slice size $\Delta$, since $\Delta = \tau$ in US. In the next experiment, we set the US slice size to be 5 minutes, which is equal to the size of the finer slice in the NS. Table 3-3 shows the performance and execution time comparison between US and NS. Here, we use RR for admission control and QF for Scheduling. The US and NS have nearly identical performance in terms of the response time and job rejection ratio. But, NS is far superior in execution times for both AC and scheduling. Upon closer inspection (Table 3-4), the NS requires far fewer active time slices than the US on an average.

In summary,

- SR is much faster than RR for admission control.
- LB tends to be slower than QF for both AC and scheduling.
- NS requires much smaller execution time than US, or achieves better performance, or has both properties.
Table 3-3. Comparison of US and NS ($\tau = 5$ minutes)

<table>
<thead>
<tr>
<th></th>
<th>Response Time (min)</th>
<th>Rejection Ratio</th>
<th>Execution Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>AC Scheduling</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LIGHT LOAD</td>
<td>US</td>
<td>6.064</td>
<td>0.000</td>
</tr>
<tr>
<td></td>
<td>NS</td>
<td>5.821</td>
<td>0.000</td>
</tr>
<tr>
<td>MEDIUM LOAD</td>
<td>US</td>
<td>9.767</td>
<td>0.006</td>
</tr>
<tr>
<td></td>
<td>NS</td>
<td>9.354</td>
<td>0.006</td>
</tr>
<tr>
<td>HEAVY LOAD</td>
<td>US</td>
<td>16.486</td>
<td>0.183</td>
</tr>
<tr>
<td></td>
<td>NS</td>
<td>17.107</td>
<td>0.173</td>
</tr>
</tbody>
</table>

Table 3-4. Average number of slices of US and NS ($\tau = 5$ minutes)

<table>
<thead>
<tr>
<th></th>
<th>Average Number of Slices</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>AC</td>
</tr>
<tr>
<td>Light Load</td>
<td>US</td>
</tr>
<tr>
<td></td>
<td>NS</td>
</tr>
<tr>
<td>Medium Load</td>
<td>US</td>
</tr>
<tr>
<td></td>
<td>NS</td>
</tr>
<tr>
<td>Heavy Load</td>
<td>US</td>
</tr>
<tr>
<td></td>
<td>NS</td>
</tr>
</tbody>
</table>

The advantage of NS can be furthered by increasing the number of slice levels. In practice, it is likely that US is too time consuming and NS is a must.

### 3.7.2 Performance Comparison of the Algorithms

In this subsection, the experimental parameters are as stated in the introduction for Section 3.7. In particular, we fix the number of paths per job ($K$) to be 8. Table 3-5 shows the response time and rejection ratio of different algorithms.

**US vs. NS.** In Table 3-5, the algorithms with NS have a comparable to much better performance than those with US. Furthermore, it has already been established in Section 3.7.1 that NS has much smaller algorithm execution times.

**Best performance.** The best performance in terms of both response time and the rejection ratio is achieved by the RR+QF combination.

Suppose we fix the slice structure and the scheduling algorithm. Then, SR has worse rejection ratio than RR because SR does not consider flow reassignment for the old jobs.
Table 3-5. Performance comparison of different algorithms

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Light Load</th>
<th>Medium Load</th>
<th>Heavy Load</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Response</td>
<td>Rejection</td>
<td>Response</td>
</tr>
<tr>
<td></td>
<td>Time (s)</td>
<td>Ratio</td>
<td>Time (s)</td>
</tr>
<tr>
<td>US+SR+LB</td>
<td>46.55</td>
<td>0.000</td>
<td>42.35</td>
</tr>
<tr>
<td>US+SR+QF</td>
<td>21.51</td>
<td>0.014</td>
<td>22.21</td>
</tr>
<tr>
<td>US+RR+LB</td>
<td>46.55</td>
<td>0.000</td>
<td>40.73</td>
</tr>
<tr>
<td>US+RR+QF</td>
<td>21.55</td>
<td>0.000</td>
<td>23.36</td>
</tr>
<tr>
<td>NS+SR+LB</td>
<td>49.60</td>
<td>0.000</td>
<td>43.83</td>
</tr>
<tr>
<td>NS+SR+QF</td>
<td>5.73</td>
<td>0.006</td>
<td>7.56</td>
</tr>
<tr>
<td>NS+RR+LB</td>
<td>49.60</td>
<td>0.000</td>
<td>43.88</td>
</tr>
<tr>
<td>NS+RR+QF</td>
<td>5.82</td>
<td>0.000</td>
<td>9.35</td>
</tr>
</tbody>
</table>

during admission control. Since response time increases with the admitted traffic load, an algorithm that leads to lower rejection ratio can have higher response time. This explains why RR often has higher response time than the corresponding SR algorithm. Note that a lower rejection ratio does not always lead to higher traffic load since some algorithms, such as RR, use the network capacity more efficiently.

Suppose we fix the slice structure and the AC algorithm. Then, LB does much worse than QF in terms of response time, because LB tends to stretch the job until its requested end time while QF tries to complete a job early. If RR is used for admission control, then under high load, the different scheduling algorithms have a similar effect on the rejection ratio of the next admission control operation. However, for medium load we notice that the work conserving nature of QF contributes to a low rejection ratio as compared to LB that tends to waste some bandwidth.

**Merit of SR and LB.** Given the above discussion, one may tend to quickly dismiss SR and LB. But as we have noticed in Section 3.7.1, SR can be considerably faster than RR in execution speed. Furthermore, it is a candidate for conducting real time admission control at the instant a request is made, which is not possible with RR.

If SR is used, then LB often has smaller rejection ratio than QF. The reason is that QF tends to highly utilize the network on earlier time slices, making it more likely to reject small jobs requested for the near future. This is a legitimate concern because, in
practice, it is more likely that small jobs are requested to be completed in the near future rather than the more distant future.

There is indication that, the more heavy-tailed is the file size distribution, the larger is the difference in rejection ratio between LB and QF. Evidence is shown in Fig. 3-6 for the light traffic load. As the Pareto parameter \( \alpha \) approaches 1 while the average job size is held constant, the chances of having a very large file increases. Even if it is transmitted at full network capacity, as in QF, such a large file can still congest the network for a long time, causing more future jobs to be rejected. The correct thing to do, if SR is used, is to spread out the transmission of a large file over its requested time interval.

![Figure 3-6. Rejection ratio for different \( \alpha \)'s under SR.](image)

To summarize the key points,

- between the admission control methods, RR is much more efficient in utilizing the network capacity, which leads to fewer jobs being rejected, while SR is suitable for fast or realtime admission control;

- if SR is used for admission control, then the scheduling method LB is superior to QF in terms of the rejection ratio.

### 3.7.3 Single vs Multi-path Scheme

The effect of using multiple paths is shown in Fig. 3-7 for the light, medium and heavy traffic loads. Here, NS is used along with the admission control scheme RR, and scheduling objective QF. For every source-destination node pair, the \( K \) shortest paths between them are selected and used by any job between the node pair. We vary \( K \) from 1 to 10, and find that multi-path often produces better response time and always produces
a lower rejection ratio. The amount of improvement depends on many factors such as the traffic load, the version of the algorithm, and the network parameters. For light load, no job is rejected. As the number of paths per job increases from 1 to 8, we get 35% reduction in response time. No further improvement is gained with more than 8 paths. For medium load, the response time is almost halved from 1 path to 10 paths. The improvement in the rejection ratio is even more impressive, from 13.3% down to 0.3%. For heavy load, there is no improvement in response time due to the significant reduction in the rejection ratio; with multiple paths, many more jobs are admitted, resulting in a large increase of the actual network load.

Figure 3-7. Single vs. multiple paths under different traffic load. A) Response time; B) Rejection ratio.

Fig. 3-8 shows the response time (A and B) and the rejection ratio (C) under medium traffic load for all algorithms. It is observed that the rejection ratio decreases significantly for all algorithms, as $K$ increases. All algorithms that use LB for scheduling, experience an increase in response time due to the reduction in the rejection ratio. But, this is not a disappointing result because it is not the goal of LB to reduce response time. All the algorithms using QF for scheduling experience a decrease in response time. Inspite of the increased load, QF is able to pack more number of jobs in earlier slices by utilizing the additional paths.
Figure 3-8. Single vs. multiple paths under medium traffic load for different algorithms. A) Response time for QF; B) Response time for LB; C) Rejection ratio.

3.7.4 Comparison with Typical AC/Scheduling Algorithm

The next experiment compares our AC/scheduling algorithm with typical, incremental AC algorithm proposed in most QoS architectures, which will be called the simple scheme. The simple scheme decouples AC from routing, and assumes a single default path given by the routing protocol. AC is conducted in real time upon the arrival of a request. The requested resource is compared with the remaining resource in the network on the default path. If the latter is sufficient, then the job is admitted. The remaining resource is updated by subtracting from it what is allocated to the new request.

Compared to our AC/scheduling algorithm, the simple scheme resembles our SR admission control algorithm but operates only on one path. For bulk transfer with start and end time constraints, the simple scheme still requires a scheduling stage, because bandwidth needs to be allocated to the newly admitted job over the time slices on its default path. Hence, we can apply the time slice structure and the scheduling objective
of LB or QF to the newly admitted job. However, unlike our scheduling algorithm, the
scheduling of the simple scheme does not reschedule the old jobs; that is, it does not
involve multi-path traffic re-assignment for the old jobs. Table 3-6 shows the rejection
ratio of the simple scheme with different slice structures and scheduling algorithms for
different traffic loads. This should be compared with Table 3-5. The simple scheme leads
to considerably higher rejection ratio than all of our schemes involving SR, which in turn
have higher rejection ratio than the corresponding schemes involving RR.

Table 3-6. Rejection ratio of the simple scheme

<table>
<thead>
<tr>
<th></th>
<th>Light Load</th>
<th>Medium Load</th>
<th>Heavy Load</th>
</tr>
</thead>
<tbody>
<tr>
<td>US+SR+LB</td>
<td>0.010</td>
<td>0.345</td>
<td>0.781</td>
</tr>
<tr>
<td>US+SR+QF</td>
<td>0.031</td>
<td>0.308</td>
<td>0.792</td>
</tr>
<tr>
<td>NS+SR+LB</td>
<td>0.000</td>
<td>0.225</td>
<td>0.596</td>
</tr>
<tr>
<td>NS+SR+QF</td>
<td>0.026</td>
<td>0.249</td>
<td>0.642</td>
</tr>
</tbody>
</table>

3.7.5 Scalability of AC/Scheduling Algorithm

For this experiment, all the jobs request to start and end at the same time, and
the AC/scheduling algorithm runs only once. The objective is to determine how the
execution time of the algorithm scales with the number of simultaneous jobs in the system,
or the number of time slices used, or the network size. In this case, RR and SR are
indistinguishable. In the following results, we use the US+SR+QF scheme.

Fig. 3-9 shows the execution time of AC and scheduling as a function of the number
of jobs. The interval between the start and end times is partitioned into 24 uniform time
slices. It is observed that the increase in execution time is linear or slightly faster than
linear. Scaling up to thousands of simultaneous jobs appears to be possible.

Fig. 3-10 shows the execution time against the number of time slices for 100 requests.
The increase is linear. With respect to the execution time, the practical limit is several
hundred slices. This is sufficient if NS is used. But with US, the slice size may be too
coarse for practical use if one wishes to cover several months of advance reservation.
Fig. 3-11 shows the scalability of the algorithm against the network size. For this, we generate random networks with 100 to 1000 nodes in 100-node increments. The average node degree is 5, 5, 7, 9, 9, 10, 10, 11, 11, and 11 respectively, so that the number of edges also increases. The network link capacity ranges from 0.1 Gbps to 10 Gbps. There are 100 jobs to be admitted and scheduled. It is observed that the execution times increase slightly faster than linear, indicating acceptable scaling behavior.

Figure 3-9. Scalability of the execution times with the number of jobs.

Figure 3-10. Scalability of the execution times with the number of time slices.
Figure 3-11. Scalability of the execution times with the network size.
CHAPTER 4
CONCLUSION

This study aims at contributing to the management and resource allocation of research networks for data-intensive e-science collaborations. The need for large file transfers is among the main challenges posed by such applications. The opportunities lie in the fact that research networks are generally much smaller in size than the public Internet, and hence, can afford a centralized resource management platform.

In Chapter 2, we formulate two linear programs, the node-arc form and edge-path form, for scheduling bulk file transfers with start and end time constraints. Our objective is to maximize the throughput, subject to the link capacity constraints. The throughput is a common scaling factor for all demand (file) sizes. This performance objective is equivalent to finding a transfer schedule that carries all the demands and also minimizes the worst-case link congestion across all links and time. It has the effect of balancing the traffic load over the whole network and across time. This feature enables the network to accept more future file transfer requests and in turn achieve higher long-term resource utilization.

An important contribution of this thesis is towards the application of the edge-path formulation to obtaining close to optimal throughput with a reasonable time complexity. We have shown that the node-arc formulation, while giving the optimal throughput, is computationally very expensive. The edge-path formulation can lead to drastic reduction of the computation time by using a small number of pre-defined paths for each file-transfer job. We discussed two path selection schemes, the shortest paths (S) and the shortest disjoint paths (SD). Both schemes are capable of achieving near optimal throughput with a small number of paths, e.g. 8 or less, for each file-transfer request. Both S and SD perform well in a small network with few disjoint paths, e.g. the Abilene backbone, while SD performs better than S in larger, well connected networks. In the evaluation process, we also showed that having multiple paths per job yields much higher throughput
than having one shortest path per job. To handle the start and end time requirement of advance reservation, we divide time into uniform time slices in our formulations. The thesis showed that using finer slices leads to significant throughput increase at the expense of longer execution time. It is therefore important to choose the right slice size that best balances such a tradeoff.

In Chapter 3, we developed a cohesive framework of admission control and flow scheduling algorithms with the following novel elements: advance reservation for bulk transfer and minimum bandwidth guaranteed traffic, multi-path routing, and rerouting and flow reassignment via periodic re-optimization.

In order to handle the advancement of time, we identify a suitable family of discrete time-slice structures, namely, the congruent slice structures. With such a structure, we avoid the combinatorial nature of the problem and are able to formulate several linear programs as the core of our AC and scheduling algorithm. Our main algorithms apply to all congruent slice structures, which are fairly rich. In particular, we describe the design of the nested slice structure and its variants. They allow the coverage of a long segment of time for advance reservation with a small number of slices without compromising performance. They lead to reduced execution time of the AC/scheduling algorithm, thereby making it practical. The following inferences were drawn from our experiments.

- The algorithm can handle up to several hundred time slices within the time limit imposed by practicality concern. If NS is used, this number can cover months, even years, of advance reservation with sufficient time slice resolution. If US is used, either the duration of coverage must be significantly shortened or the time slice be kept very coarse. Either approach tends to degrade the algorithm’s utility or performance.

- We have argued that between the admission control methods, RR is much more efficient than SR in utilizing the network capacity, thereby leading to fewer jobs being rejected. On the other hand, SR is suitable for fast or real time admission control. If SR is used for admission control, then the scheduling method LB is superior to QF in terms of rejection ratio. We also observed that multi-path improves the network utilization dramatically.
• The execution time of our AC/scheduling algorithms exhibit acceptable scaling behavior, i.e., linear or slight faster than linear scaling, with respect to the network size, the number of simultaneous jobs, and the number of slices. We have high confidence that they can be practical. The execution time can be further shortened by using fast approximation algorithms, more powerful computers, and better decomposition of the algorithms for parallel implementation.

Even in the limited application context of e-science, admission control and scheduling is a large and complex problem. In this thesis, we have limited our attention to a set of issues that we think are unique and important. This work can be extended in many directions. To name just a few, one can develop and evaluate faster approximation algorithms as in [3, 21, 24, 36]; address additional policy constraints for the network usage; incorporate the discrete lightpath scheduling problem; develop a price-based bidding system for making admission request; or address more carefully the needs of the MBG traffic class, such as minimizing the end-to-end delay.
REFERENCES


BIOGRAPHICAL SKETCH

Kannan Rajah received his Master of Science in computer engineering from University of Florida in 2007. He pursued research in scheduling and optimization algorithms for bulk file transfers under advisors Dr. Sanjay Ranka and Dr. Ye Xia. He has published a paper titled Scheduling Bulk File Transfers with Start and End Times in the IEEE Network Computing and Applications (NCA) 2007 proceedings. Kannan received his Bachelor of Engineering (Hons.) in computer science and Master of Science (Hons.) in chemistry from Birla Institute of Technology and Science (BITS)-Pilani, India in 2000.