IMPROVING NETWORK PERFORMANCE
FOR APPLICATIONS RUNNING OVER WIDE
AREA NETWORKS

BY

YINAN LIU

A THESIS SUBMITTED IN CONFORMITY WITH THE REQUIREMENTS
FOR THE DEGREE OF MASTER OF APPLIED SCIENCE,
DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING,
at the UNIVERSITY OF TORONTO.

COPYRIGHT © 2016 BY YINAN LIU.
ALL RIGHTS RESERVED.
Abstract

In recent years, research on wide area networks has gradually spread to more and more applications. However, due to bandwidth limitation of wide area networks, it is necessary to tackle with to provide better services for users, while in traditional applications operating within a single region, the link bandwidth is no concern. In this thesis, we propose and implement different routing algorithms for two types of applications, delay-sensitive applications and streaming analytics applications, respecting their own demands and working conditions. For applications with stringent requirements on latency, we implement our delay-optimized routing mechanisms to minimize packet delay explicitly for applications with high priorities. We then concentrate on streaming analytics application, which is another popular topic recently. On the basis of existing batch-based stream processing system, Spark Streaming, we propose an algorithm jointly considering query routing and batch sizing to exploit detoured high-bandwidth links.
TO MY PARENTS
Acknowledgments

First and foremost, I would like to express my sincere gratitude to my thesis supervisor, Professor Baochun Li, who has inspired my mind and guided me throughout my master studies at University of Toronto. I have tremendously benefited from his sharp visions, scientific insights and practical sensibility. I deeply appreciate his technical support and precious advice in all aspects of my academic development.

I also would like to thank Professor Di Niu for his close collaboration and helpful suggestions throughout the work. I am grateful for my examination committee, including Professor Ben Liang, Professor Alberto Leon-Garcia and Professor Cristiana Amza. Their valuable comments and advice help greatly on the improvement of thesis quality.

I feel very fortunate to work with talented members in iQua Research Group. The list is in the order of seniority: Dr. Wei Wang, Jun Li, Li Chen, Liyao Xiang, Shuhao Liu, Hao Wang, Wanyu Lin, Siqi Ji. I enjoy every moment with them, in the BA4176 or out of campus. I wish them all the best in pursuing their dreams in the future.

Last but not least, my deepest gratitude goes to my parents — my father Jianhu Liu and my mother Yafang Wang — for their unconditional love and constant support. It’s their encourage to sustain me through trying times. The spirit of this thesis is dedicated to them.
# Contents

Abstract \hspace{2cm} ii

Acknowledgments \hspace{2cm} iii

List of Tables \hspace{2cm} vii

List of Figures \hspace{2cm} ix

1 Introduction \hspace{2cm} 1
   1.1 Delay-Sensitive Applications \hspace{2cm} 2
   1.2 Streaming Analytics Applications \hspace{2cm} 6
   1.3 Thesis Organization \hspace{2cm} 9

2 Related Work \hspace{2cm} 10
   2.1 Delay-Sensitive Applications \hspace{2cm} 10
   2.2 Streaming Analytics Applications \hspace{2cm} 12

3 Delay-Optimized Traffic Routing \hspace{2cm} 15
   in Software-Defined Wide Area Networks \hspace{2cm} 16
   3.1 Motivation Example \hspace{2cm} 16
3.2 System Model and Problem Formulation ........................................... 17
  3.2.1 Latency Minimization in Over-Provisioned Networks .................. 18
  3.2.2 Utility Maximization under Insufficient Bandwidth ..................... 21
  3.2.3 Finding Available Paths .......................................................... 23
3.3 A Sparse Solution ............................................................................. 24
  3.3.1 Convergence Analysis ................................................................. 26
3.4 Implementation ................................................................................ 28
3.5 Performance Evaluation ................................................................. 31
  3.5.1 Bandwidth and Latency Measurements ........................................ 33
  3.5.2 Schemes for Comparison ............................................................. 34
  3.5.3 Sparse Traffic Routing vs. Shortest Path Routing ....................... 36
  3.5.4 Latency-Based vs. Hop-Based Optimizations ............................... 41
  3.5.5 Flow Competition under Insufficient Bandwidth ......................... 43
3.6 Summary ......................................................................................... 45

4 Joint Routing and Batch Sizing

in Wide Area Stream Processing Systems ............................................. 47
4.1 Motivation Example ......................................................................... 48
4.2 System Model and Problem Formulation ......................................... 49
  4.2.1 System Model ............................................................................ 50
  4.2.2 Maximizing the Query Goodput ................................................ 51
  4.2.3 Learning the Basis Functions ...................................................... 53
4.3 An ADMM Algorithm for Joint Tree Selection and Batch Sizing ........ 55
  4.3.1 An ADMM Algorithm to Decouple Non-Convex Constraints ....... 57
  4.3.2 Generating Sparse Tree Selection .............................................. 59
4.4 Implementation .......................................................... 60
4.5 Performance Evaluation ............................................... 63
  4.5.1 Experimental Setup ............................................... 64
  4.5.2 Batch Sizing and Evaluation Methodology ..................... 65
  4.5.3 Experimental Results ............................................. 67
4.6 Summary ................................................................. 70

5 Conclusion ................................................................. 71
  5.1 Concluding Remarks .................................................. 71
  5.2 List of Publications ................................................... 73

Bibliography ............................................................... 74
List of Tables

3.1 10th percentile of link capacities measured in the Amazon EC2 Inter-datacenter Network. ........................................... 36

3.2 Number of Trees/Paths Selected in Experiment 1 with Sparse Traffic Routing (1). ........................................... 41

3.3 Number of Trees/Paths Selected in Experiment 2 with Sparse Traffic Routing (1), (6) and (7). ........................................... 43

3.4 Number of Trees/Paths Selected in Experiment 3 with Sparse Traffic Routing (8) as compared to Multi-Commodity Flow (MCF). ........................................... 44

4.1 Input data generation rate used in our experiment. ................. 66

4.2 Batch size generated by BW-Aware Routing and Star Topology scheme. 67
List of Figures

3.1 A poor routing decision may lead to unoptimized packet delays even in an over-provisioned network. Each link has a bandwidth capacity of 2 units. 16
3.2 Examples of software-defined networks at the network vs. application layer with the same underlying network topology. 28
3.3 The 6 Amazon EC2 datacenters used in our deployment and experiments. 32
3.4 Our experiment to verify the independence of link latency measurements on the throughput. 32
3.5 The relationship between average RTT and throughput on each link in the experiment in Fig. 3.4. 32
3.6 A dynamic comparison of Shortest Path Routing and Sparse Traffic Routing (1), as 40 sessions join progressively over time. 38
3.7 End-to-end packet delays for Sparse Traffic Routing (1) and Shortest Path Routing. 40
3.8 Comparison among different Sparse Traffic Routing schemes (1), (6) and (7). 42
3.9 Packet delays of different Sparse Traffic Routing schemes (1), (6) and (7). 42
3.10 Performance comparison among Shortest Path Routing, MCF and Sparse Traffic Routing (8) ......................................................... 44

4.1 The topology of motivation example. The red line represents the bottleneck link. ................................................................. 48

4.2 The input-output relationship of WordCount base on 12 GB of Wikipedia data. Linear regression reveals the fit of the model for the output size $U(I) = 0.40I + 0.43$ as a function of the input size $I$. ......................... 53

4.3 The wide area network emulation testbed launched for streaming experiments. ................................................................. 65

4.4 The average computation time, transfer time and processing time of two sets of experiments. Fig. 4.4(a), Fig. 4.4(c) and Fig. 4.4(e) show the results of the first set of experiments, Fig. 4.4(b), Fig. 4.4(d) and Fig. 4.4(f) represent the results of the second set of experiments. ......................... 68

4.5 The scheduling delay of query 2 under two sets of experiments. ......... 69
Chapter 1

Introduction

Many types of applications are being generated, transferred and computed over the wide area. For example, social live video streaming applications, such as Periscope [1], Meerkat [2], and FaceTime [3], may allow users to broadcast their videos live to other users; online service providers, such as Google and Microsoft, may require to copy all of the users’ data from one region to another for long-term storage; social networks, such as Facebook and Twitter, may need to detect popular keywords in minutes. In these examples, data are generated from all over the world, first collected at local nodes or points of presence (PoP), before being transmitted to the end users or the destination sites.

This recent trend of operating applications over the wide area, as opposed to traditional types of applications running locally, introduce a new set of research issues at the intersection of applications and networks. Bandwidth limit is one of the challenges should be definitely taken into consideration, when designing algorithms for wide area applications.

However, each kind of applications has its own requirements. For example, video streaming applications care about the latency most, background flows want much more
bandwidth, streaming analytics applications hope to minimize job completion time, etc. It is not easy to accomplish all these goals in one algorithm. Therefore, in this thesis, we target at designing algorithms to optimize network performance for different types of applications explicitly, in terms of their specific demands.

In the following sections, for both two types of applications, we give a brief overview of existing problems and corresponding limitations of current solutions, show the major contributions we make. Finally, we present how this thesis is organized.

1.1 Delay-Sensitive Applications

Modern datacenters are deployed around the world, in a geographically distributed way, such as Amazon Web Service (AWS), Google Cloud and Microsoft Azure. To reduce management cost and improve performance, many delay-sensitive applications (e.g., video streaming applications) rely on these public cloud providers. For example, Netflix has migrated its entire video streaming service to AWS datacenters [4]. All these applications with high sensitivity to latency can greatly benefit from a well provisioned inter-datacenter wide-area network (inter-DC WAN), provided by major cloud service providers. With inter-datacenter capacities approaching hundreds of Mbps or higher [5], content stored at one datacenter can be swiftly transferred to another that is closer to the requesting user. In terms of social live streaming, the content generated at one user’s mobile device can be relayed through the inter-datacenter network to other users in remote regions at a much higher throughput than that of direct point-to-point links.

For the same reason, other types of applications are also relying on the high-speed inter-datacenter network of cloud providers, typically for service replication, bulk transfers, data backup, and database maintenance, especially in cloud storage (e.g., Dropbox,
1.1. DELAY-SENSITIVE APPLICATIONS

Google Drive and Microsoft OneDrive) and email services. Therefore, flows with different service priorities share the same links in inter-datacenter networks. Under this scenario, the delay-sensitive flows, such as video traffic, which has a much more stringent requirement on latency, has to compete for the shortest paths with all kinds of traffic, including bandwidth-hungry flows, which are large in size yet less sensitive to latency.

Current solutions for distributed inter-datacenter network resource allocation are not particularly designed to accommodate applications with high delay-sensitivities. As an example, Multiprotocol Label Switching Traffic Engineering (MPLS TE) is commonly applied in most inter-datacenter wide area networks today [6, 7]. Without using any global coordination, MPLS TE may greedily assign flows to the shortest paths with available capacity as they arrive [6], which often leads to suboptimal routing decisions. For example, delay-sensitive traffic may be forced onto detoured paths with longer latencies, when earlier background flows have occupied direct links between datacenter pairs. Moreover, with MPLS TE, it is difficult to prioritize routing decisions and path selections according to service urgency levels.

In response, it has recently become an important research topic to engineer inter-datacenter traffic from a global point of view. Google’s software-defined inter-datacenter WAN, B4 [8], adopts an approximate fairness criterion to greedily maximize flow rate allocations according to certain bandwidth functions that indicate flow priorities. Microsoft has presented SWAN [5], a software-driven inter-datacenter WAN, that classifies flows into three classes — interactive, elastic and background flows — according to their delay-sensitivities, and solves a multi-commodity flow problem in each flow class to maximize the throughput. With centralized traffic engineering and control over Openflow-enabled switches in a software-defined network, these solutions have improved network
1.1. DELAY-SENSITIVE APPLICATIONS

throughput and addressed the flow priority issues to a certain degree.

Unfortunately, the routing algorithms in B4 and SWAN are not explicitly designed to minimize packet delays for delay-sensitive flows, which are transmitted among other diverse types of inter-datacenter traffic. Moreover, a common limitation faced by both B4 and SWAN is that traffic engineering usually leads to fractional rate allocations on multiple paths, incurring a large number of rules to be installed on switches, disregarding rule count limits on hardware switches. Additionally, splitting traffic over a large number of parallel paths may incur packet resequencing overhead at the destination, which may also increase latencies. In fact, the problem of finding a fixed number of paths that maximize throughput is NP-complete [9].

Therefore, we have designed and implemented a new software-defined sparse traffic routing solution to optimize packet delays, specifically tailored to optimize packet delays for delay-sensitive traffic, when they are transmitted among other diverse types of flows over the inter-datacenter wide area network. We make the following major contributions:

First, unlike prior work on inter-datacenter traffic engineering that mainly aims to maximize throughput or link utilization, we explicitly optimize flow latencies according to delay-sensitivities, which is more important for traffic with high delay-sensitivities. Prior work [5] implicitly optimizes latencies by allocating rates to three flow classes (interactive, elastic, background) one after another following the order of urgency. In contrast, we do not divide delay-sensitivities into a small and fixed number of classes. Instead, we respect the diverse delay-sensitivities of all flows and incorporate a fine-grained priority measure in our optimization. Note that unlike throughput which is a linear function of rate allocation, the latency of a flow split on multiple paths (i.e., the maximum latency of all chosen paths with non-zero rates) is a non-convex function of the allocated rates,
which makes the problem a non-convex integer program. We adapt the Log-det heuristic [10], which was originally introduced to solve matrix rank minimization [11], to tackle our delay minimization problem through a short series of linear programs (LPs). We mathematically prove the convergence of the proposed algorithm under a range of delay measures and demonstrate its fast convergence and scalability in general scenarios.

Second, while leveraging multipath diversity, our solution yields sparse path selection such that each flow is assigned to only one path in most cases (two paths in some rare cases), leading to a small number of switching rules, and addressing the issue of fine-grained flow splitting. Moreover, sparse path selection also helps to reduce packet reordering overhead, and thus further reducing packet delays. We have shown that our proposed Log-det algorithm converges under sparsity regularization and effectively limits the path selection for most sessions to a single path.

Third, we have implemented delay-optimized sparse traffic routing in a software-defined networking (SDN) framework at the application layer instead of the network layer, requiring no change on the underlying infrastructure. Since each datacenter usually has only a few WAN switches [5] and the number of flow entries a hardware switch can support is limited, these hardware switches will not be able to store a large number of computed rules after all, as the number of flows scales up. Further supported by the fact that not all switches in datacenters support OpenFlow [12] and SDN, it may be costly, if feasible at all, to implement complicated and dynamic inter-datacenter routing rules at large scales in hardware switches.
1.2 Streaming Analytics Applications

Since more and more large scale organizations provide real-time services globally, streaming analytics which analyze continuously created data across wide area network is an important workload. Examples of such analyses include querying word count to detect popular keywords in minutes; obtaining the number of clicks on websites to make recommendation every few seconds; and monitoring system logs to detect failures in seconds. In each of the examples above, log-like analytics is generated at edge locations that have adequate computation and storage capacity, but there is limited or unpredictable bandwidth to allow data aggregated to a central site repeatedly answer a standing query at a certain frequency.

Wide area stream processing also applies to the cases beyond log-like data. For example, video clips from networks of cameras used in urban surveillance and traffic monitoring are collected in a distributed fashion and sent to a central site for storage and analysis. User-posted photos on a social networking service are first uploaded to local servers, which may need to synchronize them to backup servers in another region or country. In these cases, a streaming query is simply a repeated collection process.

Spark Streaming [13] is a general-purpose framework to simplify large-scale stream processing deployment, based on an extension of the Apache Spark framework [14]. Spark is built on the concept of Resilient Distributed Datasets (RDDs) [15], where an RDD is a batch of input data. Similarly, Spark Streaming relies on the concept of discretized streams (DStreams) for data abstraction. A DStream is a continuous sequence of RDDs arriving at different time steps, where each RDD contains one time slice of the data in the stream. The length of the time slice is referred to as the batch size. Spark Streaming performs a transformation on a DStream by applying the same transformation on each
1.2. STREAMING ANALYTICS APPLICATIONS

RDD in the DStream. For example, an operator may place the text data received every second into an interval, and run a MapReduce operation on each interval to compute a word count per second. Therefore, Spark Streaming is based on a “micro-batch” architecture, where the streaming computation is carried out as a continuous series of batch computations.

However, the current Spark Streaming framework mainly focuses on fast recovery from faults and stragglers (slow nodes) [13] in a single datacenter, with machines interconnected by a high-bandwidth network. Although streaming analytics find a wide variety of applications in the wide area, Spark Streaming is not able to allocate the network resource appropriately from a global view. It is not specifically designed to take into account the significant bandwidth variation on wide area network links. Directly transferring all the collected data from a source to its central collecting site may not always be the best choice, if the direct link between them has limited bandwidth availability. In fact, since Spark Streaming processes all the micro-batches from different sources generated at the same timestamp together. It means that even a bottleneck link at a single source can significantly slow down the overall query response rate, which implies that the operator may lose the chance to make a key decision based on the query.

Since simple data locality mechanisms, such as performing reduction or local aggregation at data sources, are insufficient as a solution in wide area networks, we extend state-of-the-art batched stream processing systems, represented by the popular Spark Streaming framework, to incorporate bandwidth-aware batch sizing and routing of data streams. To be specific, the system should be able to (1) select a fast path to route the DStream from each source to its collecting site and avoid bottleneck links, where the fastest path is not necessarily the direct route but might be a detoured path with higher
bandwidth; (2) decide the minimum batch size for each query which may rely on data generated from multiple input sources, when multiple queries coexist; and (3) place reducer tasks at proper intermediate nodes to perform data reduction and support flexible routing decisions. The complexity of the problem comes from their coupled nature: both the achievable batch sizes and reducer placement depend on which paths are selected, while path selection in turn depends on the required batch sizes. To tackle the problem, we make the following original contributions:

First, we have solved the seemingly complex problem above in a neat and efficient optimization framework called ADMM [16], an algorithmic framework that has recently gained widespread popularity in statistical machine learning. Through an innovative problem reformulation, we have proposed an ADMM algorithm to jointly learn the efficient data paths for each query together with its optimal query response rate, based on easily monitored traces including link bandwidth and the relationship between input and output sizes in the application of interest. We have also introduced additional sparsity enforcement mechanisms in ADMM to yield sparse route selection for each query, and to avoid traffic splitting.

Second, we have implemented a routing functionality for Spark Streaming to support flexible routing and task scheduling decisions, including detoured transfers, since the most efficient path for each data source in a query to its collecting site is not always the direct link. In our implementation, we adopted a non-intrusive approach by adding a new transformation of DStreams to Spark Streaming. This approach allows explicit DStream migration between any two nodes in a geo-distributed cluster. The new transformation realizes any computed routing and task scheduling decisions through straightforward modifications to the application flow only, while leaving all load balancing and data
1.3. THESIS ORGANIZATION

locality mechanisms in the original spark task scheduler intact.

1.3 Thesis Organization

The remainder of this thesis is organized as follows. We present related work regarding these two types of applications separately in Chapter 2. Focusing on minimizing packet delay when multiple flows with different priorities competing the bandwidth over the wide area networks, in Chapter 3 we propose a delay-optimized traffic routing scheme to explicitly differentiate path selection for different applications according to their delay sensitivities. In Chapter 4, we extend batch-based stream processing frameworks, and present an efficient sparsity-penalized ADMM algorithm, so that detoured high-bandwidth links in wide area can be fully exploited and meet the batch deadlines in streaming. Finally, we summarize and conclude our work in Chapter 5.
Chapter 2

Related Work

There have been many studies focusing on improving network performance over the wide area networks. In this chapter, we first present some reading closely related to delay-sensitive applications running on the inter-datacenter wide area networks, and then discuss our work about streaming analytics applications processed over wide area in the context of related work.

2.1 Delay-Sensitive Applications

Inter-datacenter wide area network traffic engineering. It has been an important research topic recently, with increasingly high volumes of traffic [17]. Multiprotocol Label Switching Traffic Engineering (MPLS TE) [18,19] were commonly used in many production inter-datacenter networks. In MPLS TE, equal cost multipath routing (ECMP) was used to first split the traffic at ingress router, and then a constrained shortest path first (CSPF) algorithm was adopted to find the best paths for each flow running in inter-datacenter networks. There are two mechanisms applied in MPLS TE to make sure that
2.1. DELAY-SENSITIVE APPLICATIONS

different services can enjoy the corresponding forwarding treatment based on their service priorities. First, paths with lower latencies and higher bandwidth are assigned to high-priority services. Second, there are multiple priority queues (4-8) on each switch, different types of services are queued at different priority queues. However, MPLS TE is likely to select the locally optimal routes instead of the globally optimal ones, due to distributed knowledge of the entire network state [6]. Our Sparse Traffic Routing algorithm running in an SDN-based system has the global view of the entire network, and can coordinate all running sessions on demand to achieve their desired resource allocation objectives.

**Software-defined networking.** There is a trend to adopt switch-level software-defined networking architecture to provide central control from a global point of view [20–26]. These existing literature focused on the case of intra-datacenter networks rather than inter-datacenter networks where latency is a concern. SWAN [5] and B4 [8] focused instead on inter-datacenter traffic engineering, and are closely related to our work. Both SWAN and B4 were implemented based on the SDN paradigm to manage traffic forwarding across multiple datacenters. SWAN [5] used linear programs (LPs), which solved a multi-commodity flow (MCF) problem, to allocate rates for sessions to maximize the throughput. It performed rate allocation first for interactive flows, then for elastic flows and finally for background flows. B4 [8] adopted approximate fairness to greedily maximize flow rate allocations according to certain bandwidth functions that indicated flow priorities. Both B4 and SWAN used heuristics to select a subset of routes from a large number of paths generated by optimization, and quantized the traffic splitting granularity down to the level supported by the switching hardware.

Our work distinguishes itself from SWAN and B4 in four aspects. *First*, instead of
maximizing throughput, we aim at reducing delays for delay-sensitive sessions. Different from Internet Service Provider (ISP) WANs that reach end users, to achieve reliability, cloud providers typically over-provision their inter-datacenter link capacity by 2-3× on a dedicated backbone [8], with an average utilization of 30-60% even on busy links [5]. Such a high bandwidth capacity implies that most inter-datacenter flows can always be accommodated at their target rates, except for a few bandwidth-hungry yet delay-tolerant background flows, which can easily be set aside during busy periods. In this case, it is less relevant to increase flow rates and more important to optimize flow latencies according to their diverse needs. Second, we do not divide flows into three fixed classes. Instead, we allow more fine-grained session priority measures indicated by a session weight, where a high weight implies high delay-sensitivity. Third, we propose a new algorithm that is able to generate sparse path selection by solving the sparsity-regularized optimization via LPs, which greatly and effectively reduces traffic splitting overhead at the source and packet resequencing overhead at the destination. Finally, our system is implemented as an application-layer SDN instead of at the network layer, and can thus support a large number of forwarding rules, with no concern of the flow table size limit. Therefore, it can potentially scale up to a much larger number of sessions.

2.2 Streaming Analytics Applications

Batch processing in wide area. Geode [27, 28], Pixida [29] are proposed to reduce cross-datacenter traffic. Flutter [30] and Iridium [31] aimed to short the whole job completion time. However, our work is designed for streaming analytics application, which has different interesting research areas compared with big data processing.

Stream processing in wide area. A number of new streaming computing engines
for large-scale stream processing are presented in recent literature [13, 32–37]. Some of them [13, 32, 35] mainly focused on providing fault-tolerant streaming services, which are orthogonal to our concerns of bandwidth variations on wide area network links with scarce bandwidth, and can be used in conjunction. The most related work is JetStream [34]. We both care the bandwidth limits, since the gap between the data generation rate and wide area network link bandwidth is constantly increasing. The strategy of aggregation and degradation leveraged in JetStream trades accuracy for reducing data size. In contrast, our work preserves the data fidelity. We relieve the bandwidth pressure by determining proper batch sizing and selecting detour routes with sufficient bandwidth.

**Batch sizing in stream processing system.** From the aspect of operating method, the stream processing system can be roughly divided into two categories. Storm [33], TimeStream [38], TeleGraphCQ [39] process stream based on *continuous operator model*. The long-lived operators exchange messages with each other in the predefined order. The streaming data can be computed immediately as it arrives. Spark Streaming [13] and Comet [37] can be classified as *micro-batch* computing engines. The streaming data arrives within a batch interval will be collected together as a micro-batch, and then the micro-batch will be computed just like the traditional batch.

Our proposed algorithm is primarily focused on the second kind of stream processing system. The problem of how to well decide periodicity of the batch is left. Das et al. [40] discussed how the batch size effects on the performance of streaming workloads. They have designed a control algorithm to automatically adapt batch size to make sure that the batched data can be processed as fast as they arrived. It is implemented within a datacenter which available bandwidth is consistently high. However, our work faces to wide area network links. We target at forwarding intermediate results on a fast path to
reduce the transfer time. As the result, the processing time can be minimized as well, and thus a small batch sizing can be achieved.

**Task scheduling.** There is a recent trend to enhance the performance of distributed jobs by carefully scheduling tasks [41–44]. These work intended to accomplish different goals. For example, Quincy [41] took data locality into consideration to reduce job completion time; Delay Scheduling [42] balanced the locality and fairness. However, our work builds upon the default Spark scheduler [42]. By adding a routing functionality, we are able to support flexible routing and task placement optimized by our algorithm.
Chapter 3

Delay-Optimized Traffic Routing in Software-Defined Wide Area Networks

Many delay-sensitive applications, such as video streaming applications, operate their geo-distributed services in the cloud, taking advantage of superior connectivities between datacenters to push content closer to users. In the meantime, inter-datacenter wide area networks carry high volumes of other types of traffic as well, including service replication and data backups, e.g., for storage and email services. It is an important research topic to optimally engineer and schedule inter-datacenter wide area traffic, taking into account the stringent latency requirements of flows with high delay-sensitivities when transmitted along links shared with other types of traffic. In this chapter, we study the problem of path selections among applications with diverse priorities to prioritize and improve the delay performance for delay-sensitive flows at a low cost.

The remainder of this chapter is organized as follows. We show a simple example
in Sec. 3.1 to motivate our work. With a sharp focus on minimizing packet latency, we formulate optimization problems under various situations in Sec. 3.2, which explicitly improve delay performance for flows with high delay-sensitivities in inter-datacenter wide area networks. Further, in Sec. 3.3, we propose an algorithm to translate a non-convex problem to convex one iteratively, and show the algorithm converges for a class of objective functions. In Sec. 3.4, we present our real-world implementation of proposed sparse traffic routing mechanism in details, based on software-defined networking framework implemented at the application layer. Our performance evaluation in Sec. 3.5 has demonstrated that our sparse traffic routing solution efficiently reduces packet delays for time-sensitive flows when they are transmitted among a diverse range of other types of flows. Sec. 3.6 summarizes the chapter.

3.1 Motivation Example

Figure 3.1: A poor routing decision may lead to unoptimized packet delays even in an over-provisioned network. Each link has a bandwidth capacity of 2 units.

To motivate our work, we begin with a simple example. Consider the network shown in Fig. 3.1, where each link has a bandwidth capacity of 2 units and a link latency of 1 ms. There are 5 sessions S1, ..., S5 all sending packets from the same source, node A, to the same destination, node C. Suppose that sessions S1 and S2 are background sessions,
and each consumes 1 unit of bandwidth with lower delay-sensitivity, while sessions S3, S4, and S5 are video sessions, and each consumes 0.5 unit of bandwidth yet with higher delay-sensitivity. The total bandwidth capacity from node A to C is 4 units and is able to accommodate all 5 sessions which have a total bandwidth demand of 3.5 units.

Now we describe a typical scenario in which the routing decision is not respecting delay-sensitivities of different sessions, and degrades the performance of video sessions. Suppose S1 and S2 are background flows. They typically arrive first and occupy the best direct route $A \rightarrow C$. The video sessions S3, S4 and S5 join afterwards, and will have to take the detoured path $A \rightarrow B \rightarrow C$ if constrained shortest path first (CSPF) routing is used, since S1 and S2 have already used up the bandwidth on the direct route $A \rightarrow C$, as shown in Fig. 3.1(a). Yet, in this case, video sessions S3, S4, and S5 will suffer from a longer latency of 2 ms, whereas the direct route $A \rightarrow C$ has been occupied by background sessions who do not care about latency at all.

A better routing decision is to adjust the routing decision after sessions S3, S4 and S5 have joined, such that video sessions will take the direct route $A \rightarrow C$, while background flows should only be allocated with the remaining bandwidth on $A \rightarrow C$, and even rerouted to detoured routes if the direct route does not have enough bandwidth, as shown in Fig. 3.1(b).

### 3.2 System Model and Problem Formulation

To fully utilize the inter-datacenter wide area network link bandwidth, we solve a traffic routing problem to determine the sending rate of each session on each available path (or tree in the case of multicast), given a collection of coexisting unicast or multicast sessions of different delay-sensitivities. Our objective is to maximize a certain aggregate
network utility that translates to delay or throughput objectives, subject to bandwidth
capacity constraints on inter-datacenter links. Moreover, for each session, the sending
rates should be *non-zero only on a couple of paths or trees* to yield sparse path selection.

We model an inter-datacenter wide area network of geo-distributed datacenters as
a complete and directed graph $G = (V, E)$, with $N = |V|$ representing the number
of datacenters. For each edge $e \in E$, we use $C(e)$ to denote its availability bandwidth
capacity, and $L(e)$ to denote the latency on link $e$, measured by the *one-way* delay, taking
into account both propagation delay and queuing delay. Suppose there are $S$ unicast
and/or multicast sessions in $G$, each with a required target rate $R_i$, $i = 1, 2, \ldots, S$ and
a *priority parameter* $w_i > 0$, where a larger $w_i$ indicates a higher priority and a greater
sensitivity to latency. Typically, interactive sessions have a higher priority value, whereas
elastic and background flows have lower priority values. Since each unicast session is a
special case of a multicast session, we will only consider multicast sessions henceforth.

For each session $i$ originated at a certain datacenter, it is not difficult to find out all
the feasible trees (or paths in the case of unicast) to reach all the destinations by a depth-
first search algorithm to be elaborated in Sec. 3.2.3. For each session $i$, we denote the
feasible multicast trees as $T_{i1}, \ldots, T_{ik_i}$, and represent the packet latency on each multicast
tree $T_{ij}$ by $L_{ij}$, for $j = 1, \ldots, k_i$. Furthermore, let $r_{ij}$ denote the rate allocated to the
tree $T_{ij}$. Then, our objective is to determine a sparse rate allocation $r_{ij}$ such that most
$r_{ij}$ are zeros while achieving different network utility objectives under different cases.

### 3.2.1 Latency Minimization in Over-Provisioned Networks

Most inter-datacenter networks are over-provisioned, as has been reported by Microsoft
[5] and Google [8]. Although the inter-datacenter network capacity is usually sufficient to
accommodate all the session demands, there could still be inefficient routing decisions that do not optimize session packet delays based on their priorities, as shown in Fig. 3.1(a).

We run a data-driven optimization framework to update traffic routing strategies periodically, respecting delay-sensitivity levels of different sessions, regardless of the order they join the network. In over-provisioned networks, all rate demands can be accommodated. Therefore, maximizing the network utility can be translated to minimizing an aggregate delay objective as follows:

\[
\min_{\{r_{ij}\}} \sum_{i=1}^{S} w_i \max_{j \in \{1, \ldots, k_i\}} \{L_{ij}\phi(r_{ij})\} + \gamma \sum_{(i,j)} \phi(r_{ij}) \tag{3.1}
\]

subject to

\[
\sum_{j=1}^{k_i} r_{ij} = R_i, \quad \forall \ i \in \{1, \ldots, S\}, \tag{3.2}
\]

\[
\sum_{(i,j): e \in T_{ij}} r_{ij} \leq C(e), \quad \forall \ e \in E, \tag{3.3}
\]

\[
r_{ij} \geq 0, \quad \forall \ (i, j), \tag{3.4}
\]

where \(\phi(\cdot)\) is an identity function defined as

\[
\phi(x) = \begin{cases} 
1, & \text{if } x > 0, \\
0, & \text{if } x = 0. 
\end{cases} \tag{3.5}
\]

Now we explain the rationale behind the formulation above. The objective of problem (3.1) is to minimize the weighted sum of session delays by controlling \(r_{ij}\), i.e., the flow rate assigned to the tree \(j\) within each session \(i\). The term \(\max_{j \in \{1, \ldots, k_i\}} \{L_{ij}\phi(r_{ij})\}\) represents the worst-case packet latency of session \(i\), which is the longest latency among all the
3.2. SYSTEM MODEL AND PROBLEM FORMULATION

chosen trees that have non-zero $r_{ij}$s, since the tree $T_{ij}$ is adopted with $\phi(r_{ij}) = 1$ only if $r_{ij} \neq 0$. The weight $w_i > 0$ is used to prioritize different sessions according to their delay-sensitivities. The weight of background flow with the minimum requirement on latency should have a $w_i$ close to 0, and the weight of an interactive delay-sensitive session will have a larger $w_i$. The regularizing term $\gamma \sum_{(i,j)} \phi(r_{ij})$ is a penalty function to yield sparse tree selection by preferring solutions with fewer non-zero $r_{ij}$s. Constraint (3.2) guarantees that flow rates distributed to different trees must sum up to the session target rate in over-provisioned networks. Constraint (3.3) ensures that none of the link capacities is violated. Thus, congestion will never occur on each link and in this case, $L_{ij}$ will be independent of rate allocations during optimization.

A variation to problem (3.1) is

$$\min_{\{r_{ij}\}} \sum_{i=1}^{S} w_i \sum_{j=1}^{k_i} L_{ij} \phi(r_{ij}) + \gamma \sum_{(i,j)} \phi(r_{ij})$$

subject to the same constraints (3.2)−(3.4). In (3.6), the worst-case latency of each session $i$ is replaced by the sum of latencies of all the chosen trees $\sum_{j=1}^{k_i} L_{ij} \phi(r_{ij})$. Note that $\sum_{j=1}^{k_i} L_{ij} \phi(r_{ij})$ can also be replaced by $\sum_{j=1}^{k_i} L_{ij} r_{ij}$, which indicates the average packet latency. However, problem (3.6) is preferred in order to yield both sparsity and low latencies, since minimizing $\sum_{j=1}^{k_i} L_{ij} \phi(r_{ij})$ also penalizes the number of trees with non-zero rates while favouring low-latency trees among the sparse selections.

Another variation is to minimize an aggregate objective on tree depths, without measuring latencies $L_{ij}$, i.e.,

$$\min_{\{r_{ij}\}} \sum_{i=1}^{S} w_i \sum_{d=1}^{D} \max_{d: \text{depth}(T_{ij})=d} \phi(r_{ij}) + \gamma \sum_{(i,j)} \phi(r_{ij}),$$

(3.7)
where for each session $i$, $\max_{j:\text{depth}(T_{ij})=d} \phi(r_{ij})$ is 0 if and only if no tree $T_{ij}$ of depth $d$ is selected. Thus, unlike problem (3.6) which minimizes the sum of latencies on all the chosen trees, problem (3.7) minimizes for each session the sum of all distinct tree depth values that are adopted. For example, if session $i$ only uses one-hop trees, $\sum_{d=1}^{D} \max_{j:\text{depth}(T_{ij})=d} \phi(r_{ij}) = 1$; if it uses both one-hop and two-hop trees, $\sum_{d=1}^{D} \max_{j:\text{depth}(T_{ij})=d} \phi(r_{ij}) = 1 + 2 = 3$. Therefore, problem (3.7) penalizes the number of distinct tree depths adopted, while preferring trees with low depths. And the advantage of problem (3.7) is that it does not even require latency measurements between datacenters.

In all of these problems (3.1), (3.6) and (3.7), delay-sensitive sessions (e.g., video streaming) with larger $w_i$s will be routed to low-latency (low-depth) paths, while delay-tolerant sessions like background flows with low $w_i$s may be placed on detour. However, as soon as delay-sensitive video sessions conclude, background flows will be routed back onto the shortest paths due to positive though small $w_i$s. Apparently, problems (3.1), (3.6) and (3.7) are all non-convex optimization problems, because of the identity function $\phi(r_{ij})$.

### 3.2.2 Utility Maximization under Insufficient Bandwidth

In some rare cases, in spite of over-provisioned inter-datacenter networks, some bandwidth-hungry background flows (with $w_i$ close to 0) may be too large to be accommodated by the available network capacity. Even in this case, the aggregate traffic of delay-sensitive sessions is most likely still small enough to be accommodated by the network. Therefore, we can perform a two-stage process to handle bandwidth-hungry flows. In the first stage, we solve problem (3.1), (3.6) or (3.7) for all delay-sensitive sessions. Then, in the second stage, we subtract the rates allocated to delay-sensitive sessions from the
bandwidth capacity to obtain the remaining link capacity. And we allocate the rates for background flows in the remaining network to maximize their throughput, e.g., by solving a multi-commodity flow (MCF) problem [5], which is an LP, or a min-cost MCF to penalize long paths. Alternatively, in the second stage, we can perform max-min fair rate allocation for all the background flows onto the remaining network [9], although in most cases, background flows are indifferent to latency performance and fairness issues.

In case bandwidth is not sufficient to accommodate even delay-sensitive sessions, which is an extremely rare scenario in today’s heavily over-provisioned inter-datacenter networks, we propose to solve a sparsity-penalized network utility maximization (NUM) problem to judiciously decide the allowable sending rate $\sum_{j=1}^{k_i} r_{ij}$ for each session $i$:

$$\text{maximize} \quad \sum_{i=1}^{S} w_i U_i(r_{i1}, \ldots, r_{ik_i}) - \gamma \sum_{(i,j)} \phi(r_{ij}) \quad (3.8)$$

subject to

$$R^L_i \leq \sum_{j=1}^{k_i} r_{ij} \leq R_i, \quad \forall \ i \in \{1, \ldots, S\}, \quad (3.9)$$

$$\sum_{(i,j): e \in T_{ij}} r_{ij} \leq C(e), \quad \forall \ e \in E, \quad (3.10)$$

$$r_{ij} \geq 0, \quad \forall \ (i, j), \quad (3.11)$$

where $U_i(r_{i1}, \ldots, r_{ik_i})$ is a utility function for each session $i$ depending on the achieved rate and latency, e.g.,

$$U_i(\{r_{ij}\}) = \begin{cases} 
\sum_{j=1}^{k_i} r_{ij} - \beta_i \max_{j \in \{1, \ldots, k_i\}} \{L_{ij} \phi(r_{ij})\}, \\
\sum_{j=1}^{k_i} r_{ij} - \beta_i \sum_{j=1}^{k_i} L_{ij} \phi(r_{ij}), \\
\sum_{j=1}^{k_i} r_{ij} - \beta_i \sum_{d=1}^{D} d \max_{j: \text{depth}(T_{ij})=d} \phi(r_{ij}), \\
\end{cases}$$
3.2. SYSTEM MODEL AND PROBLEM FORMULATION

where each $U_i$ is a function of the total rate allocated for session $i$ minus a certain delay measure. As NUM is a well-known formulation for optimal resource allocation and for handling network congestion, our sparsity-penalized NUM (3.8) further drives the rate allocations to zero on most trees. Furthermore, for a delay-sensitive session with a larger $w_i$, problem (3.8) will prefer a solution that allocates more rate $\sum_{j=1}^{k_i} r_{ij}$ to session $i$ until its target rate $R_i$ is achieved in constraint (3.9). On the other hand, for a background flow with a small $w_i$, problem (3.8) will allocate a rate to it with a lower priority, if there is still leftover bandwidth after all delay-sensitive sessions have reached their target rates.

3.2.3 Finding Available Paths

Although the number of datacenters, $N$, is small, the number of possible multicast trees for each multicast session may be large. Taking the worst case of unicast sessions as an example, on a complete graph, there exist $\sum_{i=0}^{N-2} A_{N-2}^i$ possible paths between every pair of datacenters. For an Amazon EC2 cloud with 7 datacenters distributed globally, the maximum number of paths between each pair of datacenters is 326. It is not hard to imagine that the number of available multicast trees for each multicast session will be even larger.

To reduce complexity, when generating the set of feasible multicast trees for each session $i$, we restrict tree depth to be no more than $D$ and only select $k$ trees with the least latencies $L_{i1}, \ldots, L_{ik}$. The rationale is that our goal is to reduce an aggregate delay objective; longer paths imply longer latencies and are unlikely to be chosen by our optimization outcome. Thus, we should exclude these longer paths from consideration upfront.

We use a simple variant of the depth-first search (DFS) algorithm to enumerate the
3.3. A SPARSE SOLUTION

$k$ shortest multicast trees mentioned above for each session $i$. In particular, starting from the source node, we continue exploring as far as possible along each branch in $G$ before reaching all the destination nodes or exceeding a hop count restriction. After all possible multicast trees $\{T_{ij}\}$ are found, we can easily calculate the latency $L_{ij}$ based on measured link latencies, i.e., $L_{ij}$ equals to the latency of the longest path from the source to any receiver in $\{T_{ij}\}$. Then the $k$ shortest multicast trees can be picked. Note that in reality, choosing only 1 or 2-hop trees would be sufficient for delay-optimized routing. Furthermore, the small number of datacenters of each cloud provider makes it very fast to generate available trees using the scheme mentioned above.

3.3 A Sparse Solution

Apparently, problems (3.1), (3.6), (3.7) and (3.8) are all non-convex optimization problems, since $\phi(r_{ij})$ is a $0 - 1$ valued integer to indicate whether tree $T_{ij}$ is chosen or not. In this section, we propose to solve these problems using a Log-det heuristic, which was first used to solve matrix rank minimization in statistical sparse recovery. As a special case, the method has been applied to cardinality minimization [10], i.e., finding a vector $x = (x_1, \ldots, x_n)$ with the minimum cardinality in a convex set $\mathcal{C}$, or equivalently, minimizing $\sum_i \phi(x_i)$ subject to $x \in \mathcal{C}$. The basic idea is to replace the $0 - 1$ valued function $\phi(x_i)$ by a smooth log function $\log(|x_i| + \delta)$ and to minimize a linearization of $\log(|x_i| + \delta)$ iteratively. In the following, we extend this technique to our particular sparse routing problems, whose objective functions are clearly far more complicated than cardinality, and provide a convergence proof for certain cases.

We present Algorithm 3.1, which replaces the integer value $\phi(\cdot)$ by a series of carefully re-weighted linear functions that are updated in each iteration.
Algorithm 3.1 Iterative Linear Relaxation for Problems (3.1), (3.6), (3.7), or (3.8)

1. \( t := 0 \). \( r_{ij}^0 := 1 - \delta \).
2. \( t := t + 1 \).
3. Given the solution \( \{r_{ij}^{t-1}\} \) in the previous iteration, define
   \[
   \hat{\phi}_ij^t(rij) := \frac{r_{ij}^{t}}{r_{ij}^{t-1} + \delta}, \quad \forall \ j \in \{1, \ldots, k_i\}, \ \forall \ i \in \{1, \ldots, S\},
   \]
   where \( \delta > 0 \) is a small positive constant.
4. Replace \( \phi(rij) \) in problem (3.1), (3.6), (3.7), or (3.8) by \( \hat{\phi}_ij^t(rij) \), and solve the modified problem to obtain \( \{r_{ij}^t\} \).
5. If \( \{r_{ij}^t\} \) approximately equals \( \{r_{ij}^{t-1}\} \), return \( r_{ij}^* = r_{ij}^t \) for all possible \( (i, j) \); else, go to Step 2.

Clearly, by replacing \( \phi(rij) \) by \( \hat{\phi}_ij^t(rij) \) in problem (3.1), (3.6), (3.7), or (3.8), the corresponding modified problem solved by Algorithm 3.1 in each iteration is a convex problem, which can also be converted to an LP in the case of problem (3.1), (3.6), or (3.7). After a number of iterations, Algorithm 3.1 will eventually yield a sparse solution of \( \{r_{ij}^*\} \) with most \( r_{ij}^* \) being zero. Recall that if \( r_{ij}^* = 0 \), the multicast tree \( T_{ij} \) is not adopted by session \( i \).

To see why the modified problem eventually approximates the corresponding original problem, note that for a sufficiently small \( \delta > 0 \), upon convergence, i.e., when \( r_{ij}^{t-1} \approx r_{ij}^t = r_{ij}^* \), we have
   \[
   \hat{\phi}_ij^t(r_{ij}^*) = \frac{r_{ij}^*}{r_{ij}^{t-1} + \delta} \approx \begin{cases} 0, \ & \text{if } r_{ij}^* = 0, \\ 1, \ & \text{if } r_{ij}^* > 0, \end{cases}
   \]
   which approximately equals \( \phi(rij^*) \). Therefore, the objective function involving \( \hat{\phi}_ij^t(rij^*) \) eventually approaches that of the corresponding original problem.
3.3. A SPARSE SOLUTION

3.3.1 Convergence Analysis

In the following, we provide a proof for the convergence of Algorithm 3.1 for a class of objective functions, while prior literature can only show convergence when the objective function is the cardinality Card(x) = \sum_i \phi(x_i) \[10\]. We point out how this result applies to various formulations proposed in the Sec. 3.2.

**Proposition 3.1.** Consider a non-convex optimization problem:

\[
\begin{align*}
\text{minimize} \quad & \sum_{i=1}^{n} w_i \phi(x_i) + \sum_{j=1}^{J} l_j \max_{i \in I_j} \phi(x_i) \\
\text{subject to} \quad & x = (x_1, \ldots, x_n) \in \mathcal{C},
\end{align*}
\]

with \( w_i \geq 0 \) and \( l_i \geq 0 \), where \( \mathcal{C} \subseteq \mathbb{R}^n \) is a convex set, and \( I_j \subseteq \{1, \ldots, n\} \) for \( j = 1, \ldots, J \). If Algorithm 3.1 is applied to problem (3.12) with \( \phi(x_i) \) replaced by \( \hat{\phi}_i^t(x_i) = x_i/(x_i^{t-1} + \delta) \) with \( \delta > 0 \) in each iteration \( t \), then we have \( x_i^t - x_i^{t-1} \to 0 \), for all \( i \).

**Proof.** Since \( \phi(x_i) = \max_{i \in I_j} \phi(x_i) \), the first term in (3.12) is a special case of the second term. Thus, it suffices to prove the case when \( w_i = 0 \) for all \( i \). Since \( \{x_i^t\} \) solves problem (3.12) with \( \phi(x_i) \) replaced by \( \hat{\phi}_i^t(x_i) = x_i/(x_i^{t-1} + \delta) \), and \( \{x_i^{t-1}\} \in \mathcal{C} \), we have

\[
\sum_{j=1}^{J} l_j \max_{i \in I_j} \frac{x_i^{t} + \delta}{x_i^{t-1} + \delta} \leq \sum_{j=1}^{J} l_j \max_{i \in I_j} \frac{x_i^{t-1} + \delta}{x_i^{t-1} + \delta} = \sum_{j=1}^{J} l_j.
\]

On the other hand, define \( y_i(t) := \frac{x_i^{t+\delta}}{x_i^{t} + \delta} \). By the inequalities between geometric and
arithmetic means, we have

\[ \sum_{j=1}^{J} l_j \max_{i \in I_j} y_i(t) \geq \sum_{j=1}^{J} l_j \prod_{i \in I_j} y_i(t)^{1/|I_j|} \]

\[ \geq \left( \sum_{j=1}^{J} l_j \right) \cdot \left( \prod_{j=1}^{J} \prod_{i \in I_j} y_i(t)^{1/|I_j|} \right)^{\frac{1}{\sum_{j=1}^{J} l_j}}. \tag{3.14} \]

Combining with inequality (3.13), we obtain

\[ \prod_{j=1}^{J} \prod_{i \in I_j} y_i(t)^{1/|I_j|} = \prod_{i=1}^{n} y_i(t)^{\sum_{j:i \in I_j} l_j/|I_j|} \leq 1, \]

or equivalently,

\[ \prod_{i=1}^{n} (x_i^t + \delta)^{\sum_{j:i \in I_j} l_j/|I_j|} \leq \prod_{i=1}^{n} (x_i^{t-1} + \delta)^{\sum_{j:i \in I_j} l_j/|I_j|}. \]

Since \( l_i \geq 0 \), the left-hand side of the above is bounded below by \( \delta^{\sum_{i \in I_j} l_j/|I_j|} \). Therefore, the non-increasing sequence \( \{\prod_{i=1}^{n} (x_i^t + \delta)^{\sum_{j:i \in I_j} l_j/|I_j|}\} \) is converging over \( t \) to a nonzero limit, which implies that

\[ \lim_{t \to \infty} \prod_{i=1}^{n} y_i(t)^{\sum_{j:i \in I_j} l_j/|I_j|} = 1. \]

Now using the inequality (3.14), we obtain

\[ \sum_{j=1}^{J} l_j \max_{i \in I_j} \left( \lim_{t \to \infty} y_i(t) \right) \]

\[ \geq \left( \sum_{j=1}^{J} l_j \right) \cdot \left( \lim_{t \to \infty} \prod_{i=1}^{n} y_i(t)^{\sum_{j:i \in I_j} l_j/|I_j|} \right)^{\frac{1}{\sum_{j=1}^{J} l_j}} = \sum_{j=1}^{J} l_j, \]
3.4. IMPLEMENTATION

We have completed a real-world implementation of proposed sparse traffic inter-datacenter wide area network routing mechanism. Our implementation is based on SDN implemented at the application layer, which provides us with the capability to optimize packet forwarding in a globally optimal manner. Despite the merits of existing SDN solutions with equality achieved only if $\lim_{t \to \infty} y_i(t) = 1$ for all $i$. Combining with inequality (3.13), we obtain $\sum_{j=1}^{J} l_j \max_{i \in I_j} \lim_{t \to \infty} y_i(t) = \sum_{j=1}^{J} l_j$. Therefore, we must have $\lim_{t \to \infty} y_i(t) = 1$ for all $i$, which means $x_i^t - x_i^{t-1} \to 0$ for all $i$.

It is not hard to check that problem (3.6) is a special case of the general form (3.12), when there is no second term (i.e., $l_j = 0$ for all $j$), and (3.7) is also a special case of (3.12) when each index group $I_j$ corresponds to all the trees of a certain depth $d$ in a certain session $i$. However, the convergence in the case of problem (3.1) is much harder to analyze, which involves a different $L_{ij}$ inside each max function.

3.4 Implementation

We have completed a real-world implementation of proposed sparse traffic inter-datacenter wide area network routing mechanism. Our implementation is based on SDN implemented at the application layer, which provides us with the capability to optimize packet forwarding in a globally optimal manner. Despite the merits of existing SDN solutions with equality achieved only if $\lim_{t \to \infty} y_i(t) = 1$ for all $i$. Combining with inequality (3.13), we obtain $\sum_{j=1}^{J} l_j \max_{i \in I_j} \lim_{t \to \infty} y_i(t) = \sum_{j=1}^{J} l_j$. Therefore, we must have $\lim_{t \to \infty} y_i(t) = 1$ for all $i$, which means $x_i^t - x_i^{t-1} \to 0$ for all $i$.

It is not hard to check that problem (3.6) is a special case of the general form (3.12), when there is no second term (i.e., $l_j = 0$ for all $j$), and (3.7) is also a special case of (3.12) when each index group $I_j$ corresponds to all the trees of a certain depth $d$ in a certain session $i$. However, the convergence in the case of problem (3.1) is much harder to analyze, which involves a different $L_{ij}$ inside each max function.

3.4 Implementation

We have completed a real-world implementation of proposed sparse traffic inter-datacenter wide area network routing mechanism. Our implementation is based on SDN implemented at the application layer, which provides us with the capability to optimize packet forwarding in a globally optimal manner. Despite the merits of existing SDN solutions with equality achieved only if $\lim_{t \to \infty} y_i(t) = 1$ for all $i$. Combining with inequality (3.13), we obtain $\sum_{j=1}^{J} l_j \max_{i \in I_j} \lim_{t \to \infty} y_i(t) = \sum_{j=1}^{J} l_j$. Therefore, we must have $\lim_{t \to \infty} y_i(t) = 1$ for all $i$, which means $x_i^t - x_i^{t-1} \to 0$ for all $i$.

It is not hard to check that problem (3.6) is a special case of the general form (3.12), when there is no second term (i.e., $l_j = 0$ for all $j$), and (3.7) is also a special case of (3.12) when each index group $I_j$ corresponds to all the trees of a certain depth $d$ in a certain session $i$. However, the convergence in the case of problem (3.1) is much harder to analyze, which involves a different $L_{ij}$ inside each max function.

3.4 Implementation

We have completed a real-world implementation of proposed sparse traffic inter-datacenter wide area network routing mechanism. Our implementation is based on SDN implemented at the application layer, which provides us with the capability to optimize packet forwarding in a globally optimal manner. Despite the merits of existing SDN solutions with equality achieved only if $\lim_{t \to \infty} y_i(t) = 1$ for all $i$. Combining with inequality (3.13), we obtain $\sum_{j=1}^{J} l_j \max_{i \in I_j} \lim_{t \to \infty} y_i(t) = \sum_{j=1}^{J} l_j$. Therefore, we must have $\lim_{t \to \infty} y_i(t) = 1$ for all $i$, which means $x_i^t - x_i^{t-1} \to 0$ for all $i$.

It is not hard to check that problem (3.6) is a special case of the general form (3.12), when there is no second term (i.e., $l_j = 0$ for all $j$), and (3.7) is also a special case of (3.12) when each index group $I_j$ corresponds to all the trees of a certain depth $d$ in a certain session $i$. However, the convergence in the case of problem (3.1) is much harder to analyze, which involves a different $L_{ij}$ inside each max function.

3.4 Implementation

We have completed a real-world implementation of proposed sparse traffic inter-datacenter wide area network routing mechanism. Our implementation is based on SDN implemented at the application layer, which provides us with the capability to optimize packet forwarding in a globally optimal manner. Despite the merits of existing SDN solutions with equality achieved only if $\lim_{t \to \infty} y_i(t) = 1$ for all $i$. Combining with inequality (3.13), we obtain $\sum_{j=1}^{J} l_j \max_{i \in I_j} \lim_{t \to \infty} y_i(t) = \sum_{j=1}^{J} l_j$. Therefore, we must have $\lim_{t \to \infty} y_i(t) = 1$ for all $i$, which means $x_i^t - x_i^{t-1} \to 0$ for all $i$.

It is not hard to check that problem (3.6) is a special case of the general form (3.12), when there is no second term (i.e., $l_j = 0$ for all $j$), and (3.7) is also a special case of (3.12) when each index group $I_j$ corresponds to all the trees of a certain depth $d$ in a certain session $i$. However, the convergence in the case of problem (3.1) is much harder to analyze, which involves a different $L_{ij}$ inside each max function.
based on OpenFlow, the table size is limited in hardware switches, which can not scale well to support a large number of unicast and multicast rules as the session number grows. Furthermore, not all hardware switches are OpenFlow-enabled. In contrast, by implementing the full control plane and data plane at the application layer, our implementation can leverage the advantage of intelligent traffic engineering offered by SDN, without changing any current infrastructure. Moreover, our system can still take advantage of all the latest technologies in Layer 2 and/or Layer 3 network components.

By implementing the core principles of the SDN paradigm, which is the separation of the control plane from the data plane, we are able to preserve the control flexibility of SDN at the application layer. Fig. 3.2 illustrates the difference between the existing SDN solution and our application-layer SDN. The application layer provides a much simpler abstraction; it hides complexities of handling flows with lower layer components and avoids fatal issues that are easy to encounter. For instance, lower-layer messages, such as ARP packets, will not be one of our concerns since they are handled naturally by the lower layer. Moreover, building our system completely at the application layer makes it readily deployable in the inter-datacenter network of any public cloud provider without making assumptions on their underlying infrastructure. Furthermore, by using virtual machines (VMs) as forwarding devices, we do not have a hard limit on the number rules that can be supported, which means that our application-layer solution can scale to a large number of sessions.

There are two major components in our system: a centralized controller implemented in Python to compute routing decisions, and forwarding devices implemented in C++ to forward traffic based on the rules instructed by the controller. Both the centralized controller and forwarding devices are operated as software systems running in virtual
instances launched at different datacenters. As the brain of our system, the controller is launched in a virtual machine in one of the datacenters. It offers the abstraction and programmability over the network. The forwarding device for each datacenter in the inter-datacenter network is launched as one or multiple VMs in that datacenter.

The controller and forwarding devices interact in intricate ways. Each forwarding device is connected to the controller using a *long-lived* TCP connection. Whenever a new forwarding device comes online, the controller will save its information and send it to other existing forwarding devices. When a forwarding device has obtained information about other devices in the network, it will start measuring the one-way delays from itself to other devices in the network, and reports the measured values to the controller to be used in routing optimization. In addition, whenever a new flow emerges in the network, the first forwarding device that has received it will send the information of this flow, such as the delay-sensitivity weight and the requested target rate, to the controller. Therefore, the controller always has the up-to-date knowledge of the entire network.

Given the information of the sessions, link latencies and bandwidth capacities gathered from forwarding devices, our controller written in Python will run the proposed Sparse Traffic Routing algorithms to determine the best routes as well as rate assignments on the selected routes for each session according to their delay requirements. There are two steps to be conducted by the controller. *First*, it will enumerate the feasible multicast trees for each session, based on the network topology constructed from the collected information as well as the source and destination of each session reported by the forwarding devices. Combining such information, the feasible multicast trees can be readily generated using the method mentioned in Sec. 3.2.3, and stored in a matrix to serve as the input of the next step. *Second*, by using CVXPY and NumPy packages, our program
solves the optimization problem (i.e., problem (3.1), (3.6), (3.7) or (3.8) depending on different objectives) to obtain sparse routing decisions and optimized rate assignments within several seconds.

To reduce unnecessarily redundant computations, our routing optimization does not run whenever a new flow joins; it will be carried out every two minutes or whenever the number of new joins has exceeded a threshold. We always allow a new flow to join the system according to simple protocols, such as a greedy shortest-path-first routing based on the current network conditions. Whenever the controller has computed a new routing scheme, it will make a global re-routing decision and send out the new forwarding rules for all existing flows to all the forwarding devices.

The forwarding devices are completely managed by the controller, and are responsible for buffering and forwarding traffic. Like traditional SDN, these forwarding devices do not have control intelligence; they need to ask the controller for forwarding rules whenever they see packets of a new flow coming in. In addition to basic storing and forwarding, we have also implemented a multi-path forwarding and traffic splitting functionality in each forwarding device. Specifically, whenever a traffic splitting decision requires a proportion of the allocated flow rate to be sent on a path or tree, the forwarding device will send packets onto the path or tree according to the corresponding probability. Since our optimization algorithms may only use traffic splitting at the source, there is no need to split an incoming flow.

3.5 Performance Evaluation

We have deployed our real-world implementation with delay-sensitive Sparse Traffic Routing on 6 Amazon EC2 datacenters distributed globally, whose locations are shown in
3.5. PERFORMANCE EVALUATION

Figure 3.3: The 6 Amazon EC2 datacenters used in our deployment and experiments.

Figure 3.4: Our experiment to verify the independence of link latency measurements on the throughput.

Figure 3.5: The relationship between average RTT and throughput on each link in the experiment in Fig. 3.4.
3.5. PERFORMANCE EVALUATION

Fig. 3.3. We have launched 6 XLarge compute instances, each with 4 compute units and 15 GB memory. Each compute instance is launched in a different datacenter and hosts our forwarding device. The centralized controller is hosted in Oregon, sharing the same compute instance as one of our forwarding devices.

3.5.1 Bandwidth and Latency Measurements

Link bandwidth capacities between the 6 datacenters can be readily measured. Measurement results have demonstrated that the available bandwidth is highly stable on all the links in the Amazon EC2 inter-datacenter network. To avoid queuing delays and congestion, we have been conservative on link capacity measurements: we use the 10th percentile of the measured available bandwidth to indicate the capacity on each link, as shown in Table 3.1, i.e., 90% of available bandwidth values we measured are higher than the 10th percentile values used in Table 3.1. By using a low capacity value, we make sure that our optimization will never make a decision that may potentially congest a link. During our experiments, all link latency measurements are logged once every 5 seconds.

We now show that latencies $L_{ij}$ used as inputs in our optimization framework will not depend on rate allocations and thus can be deemed as constants during the relatively short period of a single optimization run. We conduct a simple experiment shown in Fig. 3.4 to verify this fact. The target rate of each session is 10 Mbps, and the available capacity on each link is listed in Table 3.1. 15 sessions come into the network at random times in a 3-minute period. Therefore, the throughput on each link accumulates as more sessions join and start transferring data on it. Fig. 3.5 plots the average round-trip times (RTTs) versus the measured throughput on three links, respectively. It is obvious that the measured RTT is stable as the rate allocated to each particular link varies. Since
each link’s capacity is not exceeded, the queuing delay of the corresponding link will remain negligible and will not accrue, while the propagation delay still plays the most important role on each inter-datacenter link. Therefore, we can conclude that $L_{ij}$ does not depend on rate allocation decisions, and thus can be treated as a constant within a single run of optimization.

### 3.5.2 Schemes for Comparison

Due to the nature of inter-datacenter transfers, we bundle small sessions (including both unicast and multicast sessions) of similar delay-sensitivity together to form larger and longer persistent session groups. The rationale is that traffic engineering is meaningless for short sessions, as they may have already left the network before an optimized rerouting decision is made. Therefore, in our experiment, we bundle sessions that are originated from the same datacenter to be sent to the same destination datacenter and have similar delay sensitivities, e.g., all live video streams of a certain quality, into a same session group. Inside a session group, we might have different smaller sessions joining and leaving from time to time, while the overall session group is still kept alive and will exist in the network for a long time. A similar technique of flow groups has also been adopted in B4 [8] to avoid overhead of overly fine-grained flow management in inter-datacenter networks. In the following, we may abuse the term session to indicate session groups.

In our experiment, we will compare the following different routing solutions, in terms of achieved end-to-end delays, fine service differentiation/prioritization among sessions of different delay-sensitivities, paths sparsity and overhead of traffic split, as well as throughput when bandwidth capacity is insufficient. The latter two schemes, i.e., Shortest Path Routing and Multi-Commodity Flow, will serve as baseline schemes which have been
adopted in current inter-datacenter routing solutions or in recent literature.

**Delay-Sensitive Sparse Traffic Routing (1), (6) and (7).** Our delay-sensitive Sparse Traffic Routing (1), (6) and (7) are all designed to reduce packet delays for sessions with high $w_i$s, yet using different objective functions. It is worth noting that we have proved in the Sec. 3.3 that Algorithm 3.1 for formulations (6) and (7) is guaranteed to converge in theory. Moreover, problem (7) measures the delay only by counting the number of hops and does not even require to measure link delays, which reduces the measurement overhead of system. In other words, problem (7) provides a sparse routing solution to minimize packet delays for delay-sensitive sessions, without relying on accurate latency measurements.

**Shortest Path Routing.** As the first baseline scheme, we have implemented the constrained shortest path first (CSPF) algorithm [19], which is commonly adopted in MPLS TE today. Specifically, it chooses the shortest path with available bandwidth for each session subject to several constraints, such as bandwidth requirements, hop limitations and session priorities.

**Multi-Commodity Flow (MCF).** As the second baseline scheme, MCF has been adopted in SWAN [5] to allocate rates in three classes of flows one after another, first processing interactive flows, followed by the processing of elastic flows and background flows. MCF is mainly used to increase the network utilization or total throughput while preferring shorter paths. MCF, as an LP, can be expressed as

$$\text{maximize} \sum_{i=1}^{k_i} r_{ij} - \beta_i \sum_{j=1}^{k_i} w_i r_{ij}$$

subject to the same target rate and capacity constraints as in problem (8). In fact, MCF
can be equivalently represented via problem (3.8), with

\[ U_i(\{r_{ij}\}) := \sum_{j=1}^{k_i} r_{ij} - \beta_i \sum_{j=1}^{k_i} L_{ij}r_{ij}. \]

Our evaluation mainly consists of three parts. First, we compare the performance of delay-sensitive Sparse Traffic Routing with that of Shortest Path Routing in over-provisioned networks. As randomly synthesized sessions join, Sparse Traffic Routing will be carried out on demand to globally re-route the sessions that already exist in the network according to their delay-sensitivities. Second, we compare different versions of delay-sensitive Sparse Traffic Routing solutions, i.e., problems (1), (6) and (7). Third, under the rare case of insufficient bandwidth, we evaluate the throughput and delay performance of our system under flow competition, as compared to Shortest Path Routing and MCF.

### 3.5.3 Sparse Traffic Routing vs. Shortest Path Routing

In this experiment, we consider the common case of over-provisioned inter-datacenter networks, where the total inter-datacenter capacity is able to support all the sessions. However, due to Shortest Path Routing, some direct links may be locally occupied by
background flows or delay-insensitive sessions which have arrived earlier. This may influence the performance of subsequently arrived delay-sensitive sessions.

We conduct our experiment with 40 unicast/multicast sessions randomly joining the network within a 10-minute period. The weight $w_i$ of each session takes one value out of 10 predetermined values, which are randomly drawn between 1 and 39 from an exponential distribution. The target request rate $R_i$ for each session is chosen uniformly at random from $4 - 10$ Mbps. When a session comes, it will be immediately directed onto the shortest path with available bandwidth, using Shortest Path Routing. Delay-sensitive Sparse Traffic Routing will be triggered under two situations, i.e., either when 10 new sessions have joined after the last optimization, or when 2 minutes have past since the last optimization. After re-routing decisions are made by the optimization, the controller will update the new forwarding rules for all the existing sessions to be installed in the forwarding devices instantly. Under this scenario, Sparse Traffic Routing has been triggered 4 times during the entire experiment.

Fig. 3.6(a), 3.6(c) and 3.6(e) show the average packet delays for sessions with different weights, right after the 2nd, 3rd and 4th Sparse Traffic Routing, respectively, as compared to always using Shortest Path Routing. From these figures, it is obvious that for sessions with higher weights $w_i$, our algorithm can effectively reduce their average packet delays compared to Shortest Path Routing. In Fig. 3.6(a), the delay benefit of Sparse Traffic Routing is relatively less significant, since at this early point the shortest links are not fully occupied yet, and most sessions can still go onto their desired paths. In Fig. 3.6(c), the advantages of Sparse Traffic Routing becomes more salient, especially for delay-sensitive sessions with higher $w_i$s. Fig. 3.6(e) evidently shows that our solution results in considerably lower average packet delays than Shortest Path Routing for delay-sensitive
Figure 3.6: A dynamic comparison of Shortest Path Routing and Sparse Traffic Routing (1), as 40 sessions join progressively over time.
sessions with high $w_i$s. It is worth noting that some low-weight sessions actually have longer delays under our scheme. This conforms to our intention of service differentiation, as background delay-insensitive flows have been placed on alternative longer paths to yield way to delay-sensitive sessions. Such a fine-grained service differentiation dictated by session weights is not possible with MLPS TE using Shortest Path Routing.

Fig. 3.6(b), 3.6(d) and 3.6(f) compare the average packet delays of the newly joined sessions before and after the next re-routing decisions made by Sparse Traffic Routing. Clearly, after each optimization, the average packet delays of high-weight sessions decrease significantly. Similarly, the benefit of our solution becomes more significant as time passes, which complies with the observations in Fig. 3.6(a), 3.6(c) and 3.6(e). Meanwhile, the sessions that join between two optimizations only need to suffer a relatively high delay for a short period of time. As soon as the next optimization has been executed, delay-sensitive sessions will be directed to shorter paths.

Fig. 3.7 plots the CDF of measured end-to-end packet delays for sessions with different weights. Fig. 3.7(a) shows that the packet delays of delay-sensitive sessions with $w_i \geq 26$, are much reduced due to the prioritization effect of our scheme. In contrast, Fig. 3.7(b) confirms our expectation that for delay-insensitive sessions with $w_i \leq 7$, our algorithm has a greater chance to incur a larger packet delay than Shortest Path Routing to make direct paths available for delay-sensitive sessions. And Fig. 3.7(c) illustrates that our algorithm evidently outperforms Shortest Path Routing.

To evaluate the sparsity of our solution in terms of path selection, Table 3.2 lists the number of sessions with traffic splitting, i.e., the sessions that have been routed on multiple paths/trees by the optimization. Recall that traffic splitting not only incurs overhead for traffic engineering and routing, but can also lead to additional packet reordering
Figure 3.7: End-to-end packet delays for Sparse Traffic Routing (1) and Shortest Path Routing.
3.5. PERFORMANCE EVALUATION

Table 3.2: Number of Trees/Paths Selected in Experiment 1 with Sparse Traffic Routing (1).

<table>
<thead>
<tr>
<th>Weight</th>
<th>1</th>
<th>4</th>
<th>7</th>
<th>8</th>
<th>13</th>
<th>18</th>
<th>21</th>
<th>26</th>
<th>29</th>
<th>39</th>
<th>total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Optimization 1</td>
<td>0/1/1</td>
<td>0/3/3</td>
<td>0/0/0</td>
<td>0/0/0</td>
<td>0/1/1</td>
<td>0/3/3</td>
<td>0/1/1</td>
<td>0/0/0</td>
<td>0/1/1</td>
<td>0/0/0</td>
<td>0/10/10</td>
</tr>
<tr>
<td>Optimization 2</td>
<td>0/2/2</td>
<td>1/6/7</td>
<td>0/0/0</td>
<td>0/1/1</td>
<td>0/2/2</td>
<td>0/5/5</td>
<td>0/5/5</td>
<td>0/0/0</td>
<td>0/1/1</td>
<td>0/0/0</td>
<td>1/20/21</td>
</tr>
<tr>
<td>Optimization 3</td>
<td>0/3/3</td>
<td>0/6/6</td>
<td>0/1/1</td>
<td>0/2/2</td>
<td>1/5/5</td>
<td>0/5/5</td>
<td>0/4/4</td>
<td>0/1/1</td>
<td>0/4/4</td>
<td>0/1/1</td>
<td>1/30/32</td>
</tr>
<tr>
<td>Optimization 4</td>
<td>0/3/3</td>
<td>1/7/8</td>
<td>1/3/4</td>
<td>0/3/3</td>
<td>0/5/5</td>
<td>1/7/8</td>
<td>2/6/8</td>
<td>0/2/2</td>
<td>0/4/4</td>
<td>0/2/2</td>
<td>5/40/45</td>
</tr>
</tbody>
</table>

Note: the values 1)/2)/3) are: 1) the number of sessions with traffic splitting, 2) the number of sessions in this weight range, 3) the total number of paths/trees adopted by these sessions.

Therefore, it is desirable to select a single path/tree for each delay-sensitive session. From Table 3.2, we can see that at most 5 out of all the 40 sessions have traffic splitting, while most sessions are still routed on a single path or tree. Furthermore, in each optimization, Sparse Traffic Routing does not choose the same sessions to routed on multiple paths. In other words, all the sessions are treated fairly, taking turns in terms of traffic splitting. In general, the results in Table 3.2 have demonstrated the effectiveness of using \{0, 1\} sparsity regularizers \(\phi(\cdot)\) in our optimization formulation.

3.5.4 Latency-Based vs. Hop-Based Optimizations

In this experiment, we have randomly generated all the parameters including requested target rates and weights of the sessions. We also have 40 sessions coexisting in the Amazon EC2 inter-datacenter network. However, we only focus on the results from one optimization this time, since we aim to compare the performance of different Sparse Traffic Routing solutions (1), (6) and (7) with different objective functions.

Fig. 3.8 shows a similarity of the formulations (1), (6) and (7) in terms of the measured average packet delays. From Fig. 3.8, we can easily tell that packet delays achieved by (1) mostly lie very close to those achieved by (6) and (7). It is worth noting that the tree-depth-based formulation (7), although only based on counting hops without measuring
3.5. PERFORMANCE EVALUATION

Figure 3.8: Comparison among different Sparse Traffic Routing schemes (1), (6) and (7). Link latencies, can yield good performance as compared to formulations (1) and (6) which require latency measurements, verifying the usefulness of hop-based formulation (7). This is because the latency on each path is highly correlated to the number of hops along the path, especially in inter-datacenter networks. Fig. 3.9 further plots the CDF of packet delays under formulations (1), (6) and (7). The formulation (6) performs better at the lower end of packet delays, while (7) outperforms the other two schemes at the higher end. Generally speaking, their performance is similar.

Figure 3.9: Packet delays of different Sparse Traffic Routing schemes (1), (6) and (7).

Table 3.3 shows that all three formulations (1), (6) and (7) have the ability to generate sparse path selection, yielding only a small number of sessions with traffic splitting, while most sessions have only adopted one tree/path.
### 3.5. PERFORMANCE EVALUATION

Table 3.3: Number of Trees/Paths Selected in Experiment 2 with Sparse Traffic Routing (1), (6) and (7).

<table>
<thead>
<tr>
<th>Weight</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sparse Traffic Routing (1)</td>
<td>0/3/3</td>
<td>0/3/3</td>
<td>0/4/4</td>
<td>0/2/2</td>
<td>0/5/5</td>
<td>0/3/3</td>
<td>0/2/2</td>
<td>3/6/9</td>
<td>0/4/4</td>
<td>0/8/8</td>
<td>3/40/43</td>
</tr>
<tr>
<td>Sparse Traffic Routing (6)</td>
<td>0/3/3</td>
<td>0/3/3</td>
<td>0/4/4</td>
<td>0/2/2</td>
<td>0/5/5</td>
<td>0/3/3</td>
<td>1/2/3</td>
<td>0/6/6</td>
<td>0/4/4</td>
<td>0/8/8</td>
<td>1/40/41</td>
</tr>
<tr>
<td>Sparse Traffic Routing (7)</td>
<td>0/3/3</td>
<td>0/3/3</td>
<td>0/4/4</td>
<td>0/2/2</td>
<td>1/5/6</td>
<td>1/3/4</td>
<td>0/2/2</td>
<td>0/6/6</td>
<td>0/4/4</td>
<td>0/8/8</td>
<td>2/40/42</td>
</tr>
</tbody>
</table>

Note: the values 1)/2)/3) are: 1) the number of sessions with traffic splitting, 2) the number of sessions in this weight range, 3) the total number of paths/trees adopted by these sessions.

#### 3.5.5 Flow Competition under Insufficient Bandwidth

In the third experiment, we consider the rare scenario of insufficient bandwidth and aim to maximize the total allocated rates according to session priorities while minimizing packet delays for high-weight sessions. In order to simulate insufficient bandwidth (which rarely happens in the Amazon EC2 inter-datacenter network), we assume that the available bandwidth on each link in the Amazon EC2 inter-datacenter is capped by half of its actual available bandwidth. We assume that there exist some bandwidth-hungry sessions, which are large in size, but have little requirements on latency performance. Therefore, their weights are set close to 0, which means that they are delay insensitive.

We test three routing algorithms, Shortest Path Routing, Sparse Traffic Routing (8) and MCF, when 40 sessions are transferring data in the network at the same time. Here, we want to compare delay, throughput, and path sparsity of these three schemes.

Fig. 3.10(a) clearly shows that the routing decision made by Sparse Traffic Routing (8) performs the best for high-weight sessions (with $w_i \geq 1$) in terms of packet latency. It is worth noting that some background flows, with $w_i$ close to 0, actually suffer longer packet delays under Sparse Traffic Routing (8) and MCF. The reason is that, even if background sessions come into the network first and use up direct paths, Sparse Traffic
3.5. PERFORMANCE EVALUATION

Figure 3.10: Performance comparison among Shortest Path Routing, MCF and Sparse Traffic Routing (8).

Table 3.4: Number of Trees/Paths Selected in Experiment 3 with Sparse Traffic Routing (8) as compared to Multi-Commodity Flow (MCF).

<table>
<thead>
<tr>
<th>Weight</th>
<th>close to 0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sparse Traffic Routing (8)</td>
<td>6/10/16</td>
<td>0/3/3</td>
<td>0/4/4</td>
<td>0/4/4</td>
<td>0/1/1</td>
<td>1/3/4</td>
<td>0/4/4</td>
<td>0/2/2</td>
<td>0/3/3</td>
<td>0/3/3</td>
<td>7/40/47</td>
<td></td>
</tr>
<tr>
<td>MCF</td>
<td>10/10/30</td>
<td>3/3/6</td>
<td>0/4/4</td>
<td>0/4/4</td>
<td>0/1/1</td>
<td>3/3/6</td>
<td>0/4/4</td>
<td>0/2/2</td>
<td>0/3/3</td>
<td>0/3/3</td>
<td>16/40/66</td>
<td></td>
</tr>
</tbody>
</table>

Note: the values 1)/2)/3) are: 1) the number of sessions with traffic splitting, 2) the number of sessions in this weight range, 3) the total number of paths/trees adopted by these sessions.

Routing and MCF algorithm will re-route background sessions to other paths, so that the shortest path is reserved for delay sensitive sessions.

A similar prioritization phenomenon is observed for the allocated rates. Fig. 3.10(b) shows the rate allocated to each session when they compete for bandwidth. While Shortest Path Routing may allocate 0 rates to some delay-sensitive sessions with high $w_i$s that arrive later than background flows, Sparse Traffic Routing (8) and MCF will always satisfy high-weight sessions first and then allocate the remaining capacity to accommodate low-weight background flows as much as possible.

Though there are not too much differences between Sparse Traffic Routing (8) and MCF in terms of the average packet delay and rate allocation, Sparse Traffic Routing (8) greatly outperforms MCF in terms of path sparsity. As shown in Table 3.4, for all
the 40 sessions, Sparse Traffic Routing (8) only splits traffic onto 2 paths for 7 sessions, while 6 of them are background flows. This is reasonable since wherever there is available bandwidth, we would split background flows onto different paths to maximize their throughput and fully utilize network resource. In contrast, MCF splits 16 sessions onto multiple paths, while only 8 of them are background flows. In other words, MCF does not minimize the number of paths assigned for each session; 8 delay-sensitive sessions have experienced traffic splitting and had to suffer from packet reordering. This demonstrates the unique strength of our solution to generate sparse routing solutions, while achieving even lower packet delays than MCF (for delay-sensitive sessions) and the same rate allocation as MCF, as shown in Fig. 3.10.

3.6 Summary

Our focus in this chapter is to study the problem of optimal inter-datacenter wide area network traffic routing, targeting a large volume of delay-sensitive traffic with stringent yet diverse latency requirements, transmitted among other types of flows between datacenters. Since inter-datacenter wide area networks are usually over-provisioned to accommodate the aggregate requested flow rates at most circumstances, we explicitly focus on minimizing the packet delays according to the diverse delay-sensitivity levels of different sessions, which can prioritize and improve the performance of delay-sensitive traffic. We propose a sparse traffic routing scheme that selects a single path for most delay-sensitive sessions to limit the overhead of fine-grained traffic splitting and packet reordering at the destination.

We have implemented our proposed delay-optimized sparse routing technique in an application-layer software-defined network, which enables traffic engineering decisions
made from a global point of view without modifying the current network infrastructure. Unlike switch-level software-defined networking, our system can scale to a large number of forwarding rules and thus accommodates a large number of sessions. Extensive experiments performed on Amazon EC2 datacenters have shown that our solution can effectively reduce packet delays for time-sensitive sessions according to various urgency levels, even when they are transmitted among a diverse range of other types of flows, with little overhead.
Chapter 4

Joint Routing and Batch Sizing in Wide Area Stream Processing Systems

Real-time big data analytics has become increasingly important, such as real-time analytics in social networks. As the growing data volumes generated and stored globally, there is a pressing need for processing them across multiple distributed servers. However, modern batch-based stream processing systems, such as Spark Streaming, are not designed to take into account bandwidth variations on wide area network links, leading to prolonged query response times and accumulated queuing delays when bottleneck links exist. In this chapter, we study the joint issue of path selections and achievable batch sizes for streaming analytics applications to intelligently exploit detoured high-bandwidth links.

The remainder of this chapter is organized as follows. To motivate our work, we show a simple example in Sec. 4.1. In Sec. 4.2, we formulate the problem of joint path selection and batch sizing for multiple queries sharing a same wide area network, with bandwidth
4.1 Motivation Example

Figure 4.1: The topology of motivation example. The red line represents the bottleneck link.

constraints. We also show how to learn the basis functions which are parts of the input required in our proposed problem. Since our problem is a non-convex optimization problem, we present an ADMM algorithm to decouple non-convex constraints in Sec. 4.3.

To support flexible routing decisions made by our bandwidth aware routing scheme, we have implemented a routing functionality on Spark Streaming. In Sec. 4.4, we introduce the implementation details. With extensive experiments conducted on Amazon EC2, we clearly show the effectiveness of our algorithm in reducing completion time of each micro-batch. We finally summarize this chapter in Sec. 4.6.

4.1 Motivation Example

Take a simple example to motivate our work. Suppose that a streaming analytics application running on the wide area network as displayed in Fig. 4.1. Its input data comes from three different sources, node A, B and C. The raw data will be processed locally at its source, and then be sent to the destination. Node D is the destination to collect intermediate data from all sources, finally process the data and submit the results. In this case, there exist significant link bandwidth variations on the wide area network. The bandwidth of link $A \rightarrow D$ is much lower than that of other links, which can be seen as
4.2 System Model and Problem Formulation

With the intuition obtained from our example, we now formulate the problem of joint query routing and batch sizing for multiple queries sharing a same wide area network with bandwidth constraints as a non-convex integer program, which is however bi-convex in routing path selection and batch sizing, i.e., given one set of variables fixed, the optimization of the other set is a convex problem. Therefore, we can propose an ADMM algorithm to solve the problem.
4.2.1 System Model

Similar to the previous chapter, we model the network of a set of geo-distributed regions as an undirected graph $G = (V, E)$, with $N = |V|$ denoting the number of regions. For each link $e \in E$, we use $C_e$ to denote its available bandwidth capacity. Suppose there are $Q$ streaming analytics queries on $G$. Each query $i$ monitors the data generated from a set of source nodes $S_i \subset V$, and collects output at one destination node $D_i \in V$. Each query $i$ has a target batch interval $\tau_i$ (in terms of seconds), specifying how often new data should be processed in query $i$. The target batch interval $\tau_i$ corresponds to the batch size in Spark Streaming.

Let $M_i(\tau_i)$ denote the output size of query $i$ per batch at the collecting node $D_i$ as a function of the batch interval $\tau_i$. Then, according to the data generation rate $r_{vi}$ at each source $v \in S_i$, the amount of output per batch $M_i(\tau_i)$ is a function of the amount of total input data generated in this batch, i.e.,

$$M_i(\tau_i) = U_i \left( \sum_{v \in S_i} r_{vi} \tau_i \right),$$

where we will show in Sec. 4.2.3 that $U_i$ is a function that only depends on the data characteristics of a particular application and query. It can be learned through measurement data for each application. Also, we will show through some real application cases that $U_i$ is often a linear function.

We can then define the goodput of query $i$ as

$$R_i(\tau_i) := M_i(\tau_i)/\tau_i,$$

which is the amount of useful information collected by the system in query $i$ at the
4.2. SYSTEM MODEL AND PROBLEM FORMULATION

destination node $D_i$ per unit time. Apparently, if the input-output relationship $U_i$ is linear in a certain application, $M_i(\tau_i)$ is linear in terms of $\tau_i$, and $R_i(\tau_i) := M_i(\tau_i)/\tau_i$ will be linear in terms of the query frequency $1/\tau_i$. Assume the batch size $\tau_i$ of each query $i$ has a lower bound $\tau_i^l$.

Our objective is to jointly determine the path selections and batch sizes for all $Q$ queries, that is, to select the optimal path $p_{vi}^*$ on $G$ from each source $v$ in each query $i$ to its destination site $D_i$, under which the total goodput $\sum_{i=1}^{Q} R_i(\tau_i)$ of all $Q$ queries in the network can be maximized.

4.2.2 Maximizing the Query Goodput

Now consider a particular query $i$ with source nodes $S_i \subset V$ and destination node $D_i \in V$. For each source $v \in S_i$, it is not difficult to find out all the paths $P_{vi}$ from $v$ to the destination site $D_i$, on which data generated from $v$ can be transmitted. Choosing one path from $P_{vi}$ for each source $v$ and combining them over all the sources $v$ in query $i$ will lead to a tree from the sources $S_i$ to $D_i$. For query $i$, we denote all these $J_i$ feasible trees as $T_{i1}, \ldots, T_{ik_i}$, where each tree $T_{ij}$ ($j = 1, \ldots, J_i$) actually represents the smallest subgraph to deliver the data flow from $S_i$ to $D_i$, without traffic splitting at any node. To limit the variable space, we only consider trees up to a number of hops, e.g., two hops.

It will soon be clear that by considering the trees (instead of paths) for each query, we can model aggregation operations at intermediate nodes. For example, in a tree $T_{ij}$ of query $i$, if the path $p_{v_1,i}$ of a source node $v_1$ shares a link $e$ with the path $p_{v_2,i}$ of another source $v_2$, then an aggregation operation (e.g., ReduceByKey, Union, Join, etc.) can be performed at the corresponding intermediate node before data is forwarded onto link $e$.

Similar to $M_i(\tau_i)$ defined above, we let $M_{ij}^e(\tau_i)$ denote the amount of data transmitted
4.2. SYSTEM MODEL AND PROBLEM FORMULATION

on link \( e \) for query \( i \) if the tree \( T_{ij} \) is selected. Apparently, just like \( M_i(\tau_i) \), each \( M^e_{ij}(\tau_i) \) can also be predetermined as a function of \( \tau_i \), i.e.,

\[
M^e_{ij}(\tau_i) = U_i \left( \sum_{v \in S_i \cap \text{Descendants of } e \text{ in } T_{ij}} r_{vi} \tau_i \right).
\]

Similar to the goodput \( R_i(\tau_i) \), if the tree \( T_{ij} \) is selected, the throughput on link \( e \) due to query \( i \) is given by

\[
R^e_{ij}(\tau_i) := \frac{M^e_{ij}(\tau_i)}{\tau_i}.
\]

Let \( x_{ij} \in \{0, 1\} \) be binary variables denoting tree selection decisions, where \( x_{ij} = 1 \) indicates that the tree \( T_{ij} \) is selected and \( x_{ij} = 0 \) indicates otherwise. Our objective is to find \( \{x_{ij}\} \) and \( \{\tau_i\} \) that can maximize the total goodput in the network, i.e.,

\[
\text{maximize} \quad \sum_{i=1}^{Q} R_i(\tau_i) \quad \text{(4.1)}
\]

subject to

\[
\sum_{i=1}^{Q} \sum_{j, x \in T_{ij}} R^e_{ij}(\tau_i) \cdot x_{ij} \leq C_e, \quad \forall e \in E, \quad \text{(4.2)}
\]

\[
\sum_{j=1}^{k_i} x_{ij} = 1, \quad i = 1, \ldots, Q, \quad \text{(4.3)}
\]

\[
x_{ij} \in \{0, 1\}, \quad j = 1, \ldots, J_i, \quad i = 1, \ldots, Q, \quad \text{(4.4)}
\]

\[
\tau_i \geq \tau^l_i, \quad i = 1, \ldots, Q, \quad \text{(4.5)}
\]

where the constraints (4.3) and (4.4) enforce that only one tree will be chosen for each query \( i \), while constraint (4.2) ensures that the total throughput on link \( e \) will not exceed the link capacity (or the allotment provided by the cloud operator) \( C_e \). Since \( R^e_{ij}(\cdot) \) and \( R_i(\cdot) \) can be learned from data offline, they serve as predetermined basis functions in our
4.2. SYSTEM MODEL AND PROBLEM FORMULATION

Figure 4.2: The input-output relationship of WordCount base on 12 GB of Wikipedia data. Linear regression reveals the fit of the model for the output size $U(I) = 0.40I + 0.43$ as a function of the input size $I$.

optimization problem (4.1). The decision variables are all the $x_{ij}$ and $\tau_i$.

Apparently, problem (4.1) is a hard non-convex optimization problem, since 1) $x_{ij}$ is a 0−1 valued integer, and 2) there is a multiplication between $R_{ij}^e(\tau_i)$ and $x_{ij}$ that leads to non-convexity. In Sec. 4.3, we will propose an efficient ADMM algorithm to solve problem 4.1, through an innovative reformulation to decouple path selection and batch sizing. We also adapt the ADMM algorithm to incorporate an iteratively reweighted sparsity regularizer to generate a sparse tree selection for each query.

4.2.3 Learning the Basis Functions

Our proposed Algorithm 4.1 requires the basis functions $R_i(\cdot)$ and $R_{ij}^e(\cdot)$ as parts of the input. As has been shown in Sec. 4.2.1, in each query $i$, $R_i(\cdot)$ and $R_{ij}^e(\cdot)$ depend on the input-output relationship $U_i$ in the specific application of interest as well as all the data generation rates $r_{vi}$ in query $i$, which can be monitored at each input source $v$.

For a variety of typical queries commonly used in MapReduce-like batch processing applications, the input-output relationship $U$ actually demonstrates a linear and stable shape over different input sizes [31], and thus can be readily profiled before running the actual streaming analytics application. For example, WordCount calculates the count of
4.2. SYSTEM MODEL AND PROBLEM FORMULATION

each distinct word in documents, where the number of distinct words linearly increases as there are more input documents. This is true at least in the regime of micro-batches, as will be verified through the profiling on Wikipedia data. Grep finds all the input strings that match a particular pattern, e.g., all the lines containing a specific word in a system log. Again, more lines will match the pattern as the amount of input logs increases. The Top-
$k$ Count gathers the counts of the most popular $k$ elements (e.g., URL, keywords), and thus the output remains constant as the input data size increases. Furthermore, in surveillance video camera networks, video clips (compressed to a fixed video bitrate, e.g., 720p) are sent to a central site for further processing. The output size in a certain period after final processing at the central site is always linearly proportional to the amount of data collected.

In fact, in Facebook’s production big data analytics cluster, the ratio of intermediate to input data sizes is 0.55 for a median query, with 24% of queries even having this ratio greater than 1 [45]. Since Spark Streaming is based on micro-batches, it exhibits a similar input-output scaling pattern, although in the regime of small batch sizes.

The particular $U$ can be learned for each application either based on benchmarking data or adaptively from historical data. Here we specifically characterize the input-output relationship $U$ of WordCount, which is a widely used benchmarking query for big data processing systems, based on a publicly available text dataset from Wikipedia [46] of 12 GB. To focus on the micro-batch regime, we split the Wikipedia dataset into small chunks of different sizes ranging from 5–50 MB, and randomly choose 10 chunks of each size to serve as the inputs of WordCount in Spark. The input-output relationship is shown in Fig. 4.2, which shows that the output size is roughly $U(I) = 0.40I + 0.43$ for a input size of $I$ between 5 MB and 50 MB, a range applicable to micro-batches in streaming.
analytics.

Once a linear regression $U_i(I)$ is learned for a query $i$, with the data generation rate $r_{vi}$ monitored at each source $v$, the output size $M_i(\tau_i)$ of the query and the intermediate output sizes $M_{ei}^c(\tau_i)$ on each link $e$ in the selected tree $T_{ij}$ in a micro-batch can be derived as shown in Sec. 4.2.1 and Sec. 4.2.2.

In particular, recall that if the input-output relationship $U_i$ is linear, $M_i(\tau_i)$ is linear in terms of $\tau_i$, and $R_i(\tau_i) := M_i(\tau_i)/\tau_i$ will be a linear function of the query frequency $1/\tau_i$. Similarly, in this case $R_{ei}^c(\tau_i)$ will be a linear function of $1/\tau_i$. Hence, in the rest of this chapter, we may abuse the notation and use $R_i(\frac{1}{\tau_i})$ to equivalently represent $R_i(\tau_i)$ and $R_{ei}^c(\frac{1}{\tau_i})$ to equivalently represent $R_{ei}^c(\tau_i)$. This way, we can utilize the linearity of $R_i(\frac{1}{\tau_i})$ and $R_{ei}^c(\frac{1}{\tau_i})$ to simplify the problem.

4.3 An ADMM Algorithm for Joint Tree Selection and Batch Sizing

In this section, we propose an ADMM algorithm to tackle the joint batch sizing and sparse path selection problem. Instead of solving problem (4.1) with integer constraints, we relax $x_{ij}$ to be a fractional number and solve the following problem penalized by a sparsity regularizer:
4.3. AN ADMM ALGORITHM FOR JOINT TREE SELECTION AND BATCH SIZING

\[
\begin{align*}
\text{maximize} & \quad \sum_{i=1}^{Q} R_i \left( \frac{1}{\tau_i} \right) - \lambda \sum_{i=1}^{Q} \sum_{j=1}^{J_i} \mathbb{1}(x_{ij} > 0) \quad (4.6) \\
\text{subject to} & \quad \sum_{i=1}^{Q} \sum_{j \in T_{ij}} R_{ij} \left( \frac{1}{\tau_i} \right) \cdot x_{ij} \leq C_e, \quad \forall \ e \in E, \quad (4.7) \\
& \quad \sum_{j=1}^{k_i} x_{ij} = 1, \quad \forall \ i, \quad 0 \leq x_{ij} \leq 1, \quad \forall \ (i, j), \\
& \quad 0 < \frac{1}{\tau_i} \leq \frac{1}{\tau_{il}}, \ i = 1, \ldots, Q,
\end{align*}
\]

where \( \mathbb{1}(x_{ij} > 0) \) is an indicator function defined as

\[
\mathbb{1}(x > 0) = \begin{cases} 
1, & \text{if } x > 0, \\
0, & \text{otherwise.}
\end{cases} \quad (4.8)
\]

The regularizing term \( \sum_{i=1}^{Q} \sum_{j=1}^{J_i} \mathbb{1}(x_{ij} > 0) \) is a penalty function to yield sparse tree selections by pushing most \( x_{ij} \) to zero.

Note that problem (4.6) is still a non-convex problem involving an integer penalty term \( \sum_{i=1}^{Q} \sum_{j=1}^{J_i} \mathbb{1}(x_{ij} > 0) \) and non-convex multiplicative constraints in (4.7). In the following, we first propose an ADMM algorithm to solve problem (4.6) without the sparsity penalty. Then, to handle the sparsity penalty, we propose an iterative reweighted procedure to solve the linearization of a smooth surrogate of the sparsity penalty in each iteration, and merge this procedure into the proposed ADMM algorithm to yield sparse tree selections \( \{x_{ij}\} \).
4.3. AN ADMM ALGORITHM FOR JOINT TREE SELECTION AND BATCH SIZING

4.3.1 An ADMM Algorithm to Decouple Non-Convex Constraints

Problem (4.6) without the sparsity penalty is a bi-convex problem, i.e., convex for path selections given fixed batch sizes and convex for batch sizes given all the path selections. Through innovative reformulation, we can decouple this non-convex problem into alternated minimizations of two quadratic programs (QPs), one for path selection, the other for batch sizing.

A general bi-convex problem [16] is of the form

$$\begin{align*}
\text{minimize} & \quad F(x, z) \\
\text{subject to} & \quad G(x, z) = 0,
\end{align*}$$

where $F : \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}$ is bi-convex, i.e., convex in $x$ for a given $z$ and convex in $z$ for a given $x$, and $G : \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}^p$ is bi-affine, i.e., affine in $x$ given $z$, and affine in $z$ given $x$. This problem can be solved by the scaled form of ADMM in the following iterative updates [16]:

$$\begin{align*}
x^{k+1} & := \arg \min_x \left( F(x, z^k) + \frac{\rho}{2} \|G(x, z^k) + u^k\|_2^2 \right) \\
z^{k+1} & := \arg \min_z \left( F(x^{k+1}, z) + \frac{\rho}{2} \|G(x^{k+1}, z) + u^k\|_2^2 \right) \\
u^{k+1} & := u^k + G(x^{k+1}, z^{k+1}).
\end{align*}$$

We now reformulate problem (4.6) without the sparsity penalty into the general bi-convex problem above. Introducing the auxiliary variable $z_{ije}$ to decouple the coupled constraint (4.7), problem (4.6) without the sparsity penalty is equivalent to the following
4.3. An ADMM Algorithm for Joint Tree Selection and Batch Sizing

Problem:

\[
\begin{align*}
\text{maximize} & \quad \sum_{i=1}^{Q} R_i \left( \frac{1}{\tau_i} \right), \\
\text{subject to} & \quad \sum_{j=1}^{Q} \sum_{e \in T_{ij}} z_{ije} \leq C_e, \quad \forall \; e \in E, \\
& \quad \sum_{j=1}^{Q} x_{ij} = 1, \quad \forall \; i, \quad 0 \leq x_{ij} \leq 1, \quad \forall \; (i, j), \\
& \quad 0 < \frac{1}{\tau_i} \leq \frac{1}{\tau_l}, \quad \forall \; i,
\end{align*}
\]  

(4.9)

which is equivalent to

\[
\begin{align*}
\text{maximize} & \quad \sum_{i=1}^{Q} R_i \left( \frac{1}{\tau_i} \right) + h\left( \left\{ \frac{1}{\tau_i} \right\} \right) + g\left( \{x_{ij}\}, \{z_{ije}\} \right) \\
\text{subject to} & \quad R_{ij} \left( \frac{1}{\tau_i} \right) \cdot x_{ij} - z_{ije} = 0, \quad \forall \; (i, j), \quad \forall \; e \in E,
\end{align*}
\]

where \( g(\{x_{ij}\}, \{z_{ije}\}) \) is an indicator function of all the uncoupled constraints of \( \{x_{ij}\}, \{z_{ije}\} \) in problem (4.9), i.e., it is zero if the second and third constraints of problem (4.9) are all satisfied and is negative infinity otherwise. Similarly, \( h\left( \left\{ \frac{1}{\tau_i} \right\} \right) \) is zero if \( 0 < \frac{1}{\tau_i} \leq \frac{1}{\tau_l}, \quad \forall \; i, \) and is negative infinity otherwise.

Now we have converted problem (4.9) into the same form as the general bi-convex problem above with a bi-affine constraint, where all \( \{x_{ij}\} \) and \( \{z_{ije}\} \) are treated as one set of variables and \( \left\{ \frac{1}{\tau_i} \right\} \) are treated as another set of variables. Then we can solve problem (4.9) with ADMM as described by Algorithm 4.1, which updates the two sets of variables alternately. Apparently, steps 4 and 5 of Algorithm 4.1 only involve quadratic
programming (QP), which can be efficiently solved by a number of existing QP solvers such as interior point methods.

**Algorithm 4.1** An ADMM algorithm for Problem (4.6) without the sparsity penalty.

1: **Input**: Basis functions $\{R_i(\cdot)\}, \{R_{ij}^e(\cdot)\}$; link capacities $\{C_e\}$.
2: **Output**: $\{x_{ij}^k\}, \{\tau_i^k\}$ when the algorithm stops.
3: $k := 0$. Initialize $\{x_0^{ij}\}, \{z_0^{ije}\}, \{u_0^{ije}\}$.
4: Solve the following subproblem to obtain $\{x_{ij}^{k+1}\}, \{z_{ije}^{k+1}\}$

\[
\min_{\{x_{ij}, z_{ije}\}} \sum_{(i,j,e)} \left( R_{ij}^e \left( \frac{1}{\tau_i^k} \right) x_{ij} - z_{ije} + u_{ije}^k \right)^2
\]

subject to
\[
\sum_{i=1}^{Q} \sum_{j:e \in T_{ij}} z_{ije} \leq C_e, \quad \forall \ e \in E,
\]
\[
\sum_{j=1}^{k_i} x_{ij} = 1, \quad \forall \ i, \quad 0 \leq x_{ij} \leq 1, \quad \forall \ (i, j)
\]

5: Solve the following subproblem to obtain $\{\tau_i^{k+1}\}$:

\[
\min_{\{\tau_i\}} - \sum_{i=1}^{Q} R_i \left( \frac{1}{\tau_i} \right)
\]

\[
+ \frac{\rho}{2} \sum_{(i,j,e)} \left( R_{ij}^e \left( \frac{1}{\tau_i^k} \right) x_{ij}^{k+1} - z_{ije}^{k+1} + u_{ije}^k \right)^2
\]

6: Update $u_{ije}^{k+1}$ by

\[
u_{ije}^{k+1} := u_{ije}^k + R_{ij}^e \left( \frac{1}{\tau_i^{k+1}} \right) x_{ij}^{k+1} - z_{ije}^{k+1}.
\]

7: $k := k + 1$, repeat until the stop criterion is met.

**4.3.2 Generating Sparse Tree Selection**

When the sparsity penalty in problem (4.6) is considered, we need to add the sparsity penalty $\lambda \sum_{i=1}^{Q} \sum_{j=1}^{k_i} 1(x_{ij} > 0)$ to the objective function in subproblem (4.10). Then,
4.4. IMPLEMENTATION

subproblem (4.10) becomes a typical $\ell_0$-norm regularized least squares problem, which can be efficiently solved by the LASSO [47] in statistical learning, which replaces the $\ell_0$-norm sparsity penalty by $\ell_1$-norm, i.e., a linear term $\lambda \sum_{i=1}^Q \sum_{j=1}^{J_i} x_{ij}$.

It has been shown [48] that reweighted $\ell_1$ minimization can further enhance the sparsity of the solutions, where to solve subproblem (4.10), we should iteratively penalize its objective by $\lambda \sum_{i=1}^Q \sum_{j=1}^{J_i} x_{ij} / (x_{ij}^t + \delta)$ for iterations $t = 1, \ldots, T$ until convergence, similar to the method we mentioned in Sec. 3.3. Such an algorithm can achieve fast convergence and minimize the linearization of a concave log-sum penalty [11] iteratively.

Here, to speed up convergence, we merge the iterations of reweighted $\ell_1$ minimization with the iterations of ADMM, leading to a revised ADMM algorithm. This algorithm is the same as Algorithm 4.1, except that in step 4 of each iteration, we replace the objective function (4.10) by

$$\sum_{(i,j,e)} \left( R_{ij}^e \left( \frac{1}{x_{ij}^k} \right) x_{ij} - z_{ije} + u_{ije}^k \right)^2 + \lambda \sum_{i=1}^Q \sum_{j=1}^{J_i} \frac{x_{ij}}{x_{ij}^k + \delta},$$

which places a weight $1/(x_{ij}^k + \delta)$ for each $x_{ij}$ when solving for $x_{ij}^{k+1}$. Intuitively speaking, the larger the $x_{ij}^k$, the smaller the weight, and the larger the produced $x_{ij}^{k+1}$, generating a sparse solution. After the procedure above is done, most queries will select only a single tree, if a query $i$ still has multiple nonzero $x_{ij}$, we simply choose the tree with the largest $x_{ij}$.

4.4 Implementation

In streaming analytics, routing the data stream is equivalent to placing tasks onto workers at desired locations. In Spark Streaming, the generator and the consumer of a data stream
is determined by application itself, but the routing decisions in between are completely made by the TaskScheduler, which is responsible for task placement.

For example, consider a WordCount streaming analytic application generated at node $A$ and consumed at node $B$. It has two computation tasks, including a map and a reduce. If we intend to route the mapped data stream to node $C$, all we need to do is to place the reduce task on node $C$.

However, Spark Streaming cannot natively support stream detours, because it only supports direct data transfers between placed tasks. To detour a data stream, an intermediate node should be employed as a simple relay. However, there is no such a computation task available by default.

To realize detoured transfer, we have to make modifications to Spark streaming. One intuitive approach of realizing our detoured transfer is to replace the default TaskScheduler with an implementation of our optimization algorithm. Unfortunately, though the required modifications are too intrusive and aggressive, we are still not able to realize the stream customized routing features.

Alternatively, we take a non-intrusive approach, making no modification to the existing internal behavior of Spark Streaming. In our system implementation, we first realize the stream routing features with a new transformation addRoute() on Spark Streaming, and then enforce the scheduling decisions by adding this transformation to the application workflow.

Overview. From a high level, addRoute() can explicitly route the data stream through a specified worker node, adding relay tasks when necessary. The function is implemented as an additional method on DStream, the data stream abstraction in Spark Streaming. It takes a single parameter, which specifies the hostname of which the data
stream should be routed. Like other transformations, `addRoute()` returns a new instance of DStream, representing the data stream that has been routed to the specific worker.

**Usage.** The usage of `addRoute()` can be explained with the previous WordCount example. To route the mapped data stream to node $C$ for the reduce task, we can simply add a `addRoute(''C'')` transformation between `map()` and `reduce()` in the application. Thus, the `reduce()` transformation is performed on the data stream that has already been routed to node $C$, which is semantically intuitive.

With `addRoute()` as a fundamental building block, the routing decisions made by our optimization algorithm can be enforced in Spark Streaming. For example, the optimization results can be fed to the `DAGScheduler`, the application workflow analyzer in Spark Streaming. The `DAGScheduler` is fully aware of all transformations made on the data stream, and it is able to automatically insert `addRoute()` to the original application workflow as additional transformations, realizing the desired stream routes.

Suppose a data stream for WordCount generated at node $A$ and consumed at node $C$. If the optimized route is $A$-$B$-$C$, where node $B$ is the desired place for the reduce task, the `DAGScheduler` can thus process the sequence of stream transformations. The processed workflow would be equivalent to `map(...).addRoute(''B'').reduceByKey(...)`.

**Benefits.** Because `addRoute()` is non-intrusive and provided as an additional transformation, it automatically enjoys several benefits. For example, it ensures full compatibility with the original Spark Streaming framework. All existing applications, customizations and code patches can still work without any change.

**Implementation Details of `addRoute()`**. In our system implementation, the routing decisions made by our optimization algorithm only serve as a request of task placement.
These requests are fed to the default TaskScheduler, which makes centralized task placement decisions with the awareness of resource availability.

In Spark Streaming, each mini-batch of DStream is processed separately, as a series of computation tasks. Each task has an important attribute, preferredLocations, which specifies its placement preferences using a list of candidate worker nodes. When being scheduled, the default TaskScheduler will strive to satisfy the preferences, by assigning a task to one of its available candidates.

addRoute() leverages this mechanism to explicitly specify the route for a data stream. To this end, it modifies the preferredLocations of the subsequent task. In the WordCount example, addRoute() changes the preferredLocations of the reduce task, forcing it to be placed on node C. The change is possible due to the “lazy evaluation” feature in Spark Streaming.

Realizing Stream Detours. Chaining addRoute() triggers a special use, where a stream detour is defined. For example, a statement dStream.addRoute(‘C’).addRoute(‘B’) defines a stream detour via node C.

On the relay node C, no transformation on data is applied by the application. In this case, extra computation tasks are generated. These tasks simply serve as forwarders of the data stream without transforming it. The incurred data transfers, i.e., receiving results from the preceding tasks and sending them to the succeeding tasks, are thus automatically handled by the Spark Streaming framework itself.

4.5 Performance Evaluation

We perform real experiments of Spark Streaming running our intelligent routing, batch sizing and task scheduling algorithm in a bandwidth-constraint environment on Amazon
4.5. PERFORMANCE EVALUATION

EC2 to emulate the scenario in wide area networks. We compare our modified Spark Streaming framework with the default Spark task scheduler as well as a common streaming topology currently adopted in most wide area stream processing systems, where each input source sends its locally aggregated DStreams to the central collecting site for processing.

4.5.1 Experimental Setup

We have launched an emulation testbed of six compute instances in the US East region of Amazon EC2, yet with bandwidth limiter implemented to emulate bandwidth constraints that would exist in wide area networks. We running Spark Streaming applications on this launched cluster, where one instance serves as the master node, while the other five instances are worker nodes. In reality, since the cost of storage and processors is decreasing at a faster pace than that of provisioning bandwidth in existing infrastructures [49], [34], bandwidth is the bottleneck in wide area streaming analytics.

Therefore, in the deployed emulation testbed, we assume compute capacity and memory are relatively adequate as compared to bandwidth availability. Hence, all the instances we used are m4.4xlarge, with 16 compute units and 64 GB memory. We use the tc command to limit the link bandwidths among the five workers. The bandwidth connections in the launched testbed are shown in Fig. 4.3, where the link bandwidth is chosen randomly from 5-35 Mbps.

We run five recurring WordCount queries in Spark Streaming in the cluster concurrently, as WordCount is a typical recurring query used in the benchmarking of big data processing systems. For each query, three input sources are placed on three randomly chosen nodes and there is one randomly chosen output node where the query is answered.
Figure 4.3: The wide area network emulation testbed launched for streaming experiments. In addition, the input data generation rate for each input-output pair is chosen uniformly at random from 5-10 Mbps as shown in Table 4.1.

4.5.2 Batch Sizing and Evaluation Methodology

In our experiments, we compare the following different task scheduling strategies on Spark Streaming, representing different routing and batch sizing decisions:

**BW-Aware Routing.** We use the proposed Algorithm 4.1 to jointly determine the proper batch size $\tau_{BW}$ for each query as well as the routing paths, i.e., the aggregation tree, for each query. Run the application flow generated by Algorithm 4.1 with the automatically inserted `addRoute()` transformations.

**Star Topology.** As the first baseline scheme, each input source node of a query sends its locally aggregated input data to the central collecting site, where the data is further processed and aggregated to produce final query output. We run Algorithm 4.1 with such a star topology as the tree selected for each query to obtain their optimal batch sizes $\tau_{star}$. The built-in data locality mechanisms in Spark are used to collocate each Receiver task with its corresponding input source and to enforce all the reduce tasks be placed at the output node. This strategy represents a common industrial practice to enforce data locality.
4.5. PERFORMANCE EVALUATION

Table 4.1: Input data generation rate used in our experiment.

<table>
<thead>
<tr>
<th>Input</th>
<th>Query 1</th>
<th>Query 2</th>
<th>Query 3</th>
<th>Query 4</th>
<th>Query 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input 1</td>
<td>6 Mbps</td>
<td>8 Mbps</td>
<td>5 Mbps</td>
<td>10 Mbps</td>
<td>8 Mbps</td>
</tr>
<tr>
<td>Input 2</td>
<td>5 Mbps</td>
<td>9 Mbps</td>
<td>4 Mbps</td>
<td>4 Mbps</td>
<td>5 Mbps</td>
</tr>
<tr>
<td>Input 3</td>
<td>8 Mbps</td>
<td>10 Mbps</td>
<td>8 Mbps</td>
<td>8 Mbps</td>
<td>7 Mbps</td>
</tr>
</tbody>
</table>

Note: Each query has three input sources, and three corresponding input generation rates.

**Default Spark.** As the second baseline scheme, we only use the built-in data locality mechanisms in Spark to collocate each Receiver task with its corresponding input source, yet rely on the default task scheduler in Spark to place all the reduce tasks.

To obtain the batch sizes for queries, we obtain the optimal batch sizing for BW-Aware Routing and Star Topology, respectively, based on the above-mentioned experimental setup, as shown in Table 4.2. For BW-Aware Routing, Algorithm 4.1 also produces a sparse solution for tree selection, in which most queries simply select a single tree. In the rare cases, that a query \( i \) selects two trees, we choose the tree with the largest \( x_{ij} \).

Apparently, with bandwidth constraints, the computed \( \tau_{BW} \) is significantly smaller than \( \tau_{star} \) for each query. This implies that BW-Aware Routing, which intelligently selects detours to avoid bottleneck links, can support a potentially higher query rate in this bandwidth-constrained scenario.

For validation and performance evaluation, we run all the queries for 10 minutes with the same batch sizes \( \tau_{BW} \) under all three different schemes to measure the stability of the system and the processing time per batch, broken down into the computation time, transfer time, and scheduling delay. We then run the same queries with the batch sizes \( \tau_{star} \) under BW-Aware Routing and Star Topology to see if both of them are stable and if BW-Aware Routing still has a benefit in this case.
4.5. PERFORMANCE EVALUATION

Table 4.2: Batch size generated by BW-Aware Routing and Star Topology scheme.

<table>
<thead>
<tr>
<th></th>
<th>Query 1</th>
<th>Query 2</th>
<th>Query 3</th>
<th>Query 4</th>
<th>Query 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\tau_{BW}$</td>
<td>2s</td>
<td>2s</td>
<td>2s</td>
<td>2s</td>
<td>2s</td>
</tr>
<tr>
<td>$\tau_{star}$</td>
<td>6s</td>
<td>9s</td>
<td>6s</td>
<td>9s</td>
<td>6s</td>
</tr>
</tbody>
</table>

4.5.3 Experimental Results

First, we run queries with the batch sizes $\tau_{BW}$ for all three schemes. Fig. 4.4(a), Fig. 4.4(c) and Fig. 4.4(e) illustrate the average computation time, transfer time and processing time per batch in each query for different schemes. As we can see that, the computation time is comparable between different schemes, since we have adequate computation resources, and is (roughly) proportional to the data generation rates in each query.

However, BW-Aware Routing can significantly reduce the transfer time in Queries 2-5, leading to a substantial decrease on the processing time per batch for these queries. While the Star Topology is slightly better than Default Spark, they both fail to avoid the bottleneck links effectively. For example, Query 2 selected the route C-D-E in our scheme to avoid the bottleneck link C-E picked in Star Topology.

Default Spark uses a random task placement mainly taking load balancing into account, and the decision could be different from time to time. It is worth noting that the Query 1 under Default Spark has good performance, possibly because the tasks of Query 1 have been randomly placed at the nodes without bottleneck links.

Furthermore, we can observe that the processing times of BW-Aware Routing scheme are smaller than the batch sizes, which means that the streaming system is stable. In contrast, the processing time under the other two schemes (except Query 1 with Default Spark) is greater than the batch size, which in fact lead to system failure in around 300 seconds.
4.5. PERFORMANCE EVALUATION

Figure 4.4: The average computation time, transfer time and processing time of two sets of experiments. Fig. 4.4(a), Fig. 4.4(c) and Fig. 4.4(e) show the results of the first set of experiments, Fig. 4.4(b), Fig. 4.4(d) and Fig. 4.4(f) represent the results of the second set of experiments.
Due to the space limit, we only show the scheduling delay of Query 2 under three schemes in Fig. 4.5(a). We can see that the scheduling delay of our scheme is consistently low, which means the micro-batches are processed faster than they are created. In contrast, the scheduling delays keep increasing for the other two schemes until failure, which means they cannot support such a query rate as high as once per 2 seconds.

In the second experiment, we run queries with the batch sizes $\tau_{\text{star}}$ computed for Star Topology under the routes optimized by our BW-Aware Routing and Star Topology. We omit Default Spark, because here we are interested in whether and how well our routing trees involving detours can support the batch sizes $\tau_{\text{star}}$ that is optimized for traditional star topology.

The results are shown in Fig. 4.4(b), Fig. 4.4(d) and Fig. 4.4(f). Again, there is not much difference between the two schemes in terms of the computation time. Furthermore, the processing time per batch of all the queries is lower than the set batch sizes $\tau_{\text{star}}$. Therefore, both schemes are stable, with scheduling delays plotted in Fig. 4.5(b) for Query 2 as an example.

However, our BW-Aware Routing scheme still obviously outperforms Star Topology in terms of processing time per batch. Though Star Topology is stable, the pressure on the bottleneck link still exists. In this case, judiciously choosing a proper detour can save
the transfer time, effectively avoiding congested links.

4.6 Summary

With the observation that wide area network bandwidths can be highly heterogeneous, and existing streaming processing frameworks are not optimized for such variations. Therefore, in this chapter, we propose a sparsity-penalized ADMM algorithm to tackle this problem. By jointly selecting path and deciding batch sizing for each query, our algorithm can take advantage of detoured route with sufficient bandwidth to effectively reduce the completion time of each micro-batch.

We have implemented our bandwidth aware routing technique on the basis of Spark Streaming. By adding a routing functionality on Spark Streaming, we extend the system to enforce detoured transfers and allow additional tasks placed at desired location. The experiments performed on Amazon EC2 clearly imply that our solution can support more frequent query response rate, while keeping the stream processing system stable.
Chapter 5

Conclusion

5.1 Concluding Remarks

Our focus in this thesis is to improve network performance for applications deploying over wide area networks, where the bandwidth limitation is a major issue. However, there is no general solution to effectively share link bandwidth for applications with different requirements under diverse scenarios. With the awareness of such diversity, we formulated optimization problems and designed corresponding algorithms for different types of applications specifically.

Delay-sensitive applications have to compete bandwidth with all other types of applications which may not have stringent requirements on latency. This leads to detoured paths with longer packet delays. To appropriately allocate network resource among flows with different service priorities, we formulate multiple optimization problems applied to two scenarios. The objective is to minimize packet delays for high priority flows in over-provisioned network or under insufficient bandwidth. Since these problems were
not convex, we proposed an algorithm to solve them iteratively. We realized our delay-optimized traffic routing scheme based on the software-defined networking framework implemented at the application layer. With extensive experiments conducted on Amazon EC2, we have shown that, by selecting paths for every flows running concurrently, our solution is particularly tailored to reduce packet delays for applications which have greater sensitivity to latency.

Still focusing on boosting wide area network performance, we turned to study streaming analytics application, which is a hot issue in recent years. Existing batch-based stream processing systems are mainly designed to operate in the context of a single datacenter where data streams are processed inside a high-bandwidth network. However, since more and more data are generated globally, the limited bandwidth makes the existing systems impractical to perform well over wide area networks. We resolved this problem by taking the significant bandwidth variation into consideration. On the basis of existing system, Spark Streaming, we find that simply choosing path is not able to tackle the problem thoroughly. Therefore, we formulated the optimization problem of query routing and batch sizing jointly. To solve the non-convex problem, we decoupled it as a bi-convex problem, and proposed an ADMM algorithm to efficiently work it out. We extend Spark Streaming by adding a routing functionality to enforce data streams transmitting onto the paths determined by our optimization and placing additional tasks at desired locations. Through emulations performed on Amazon EC2, we have demonstrated that, our proposed solution can reduce job completion time of each micro-batch while supporting higher query response rates.
5.2 List of Publications

The work described in this thesis has been presented in the following publications:

Journal Publication


Conference Paper


The work is collaborated with Professor Di Niu, who is an Assistant Professor in the Department of Electrical and Computer Engineering at University of Alberta. Thanks for him to provide all the helpful guidance, discussions and suggestions towards this thesis.
Bibliography


