RICE UNIVERSITY

High-Performance Data Multicast in Hybrid Data Center Networks

by

Xiaoye Sun

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE

Doctor of Philosophy

APPROVED, THESIS COMMITTEE:

Dr. T. S. Eugene Ng (Chair)
Professor of Computer Science and Electrical and Computer Engineering

Dr. Peter J. Varman
Professor of Electrical and Computer Engineering and Computer Science

Dr. Alan L. Cox
Professor of Computer Science and Electrical and Computer Engineering

Houston, Texas

October, 2018
ABSTRACT

High-Performance Data Multicast in Hybrid Data Center Networks

by

Xiaoye Sun

Nowadays, a significant number of big data processing applications, such as machine learning algorithms and database queries are implemented based on various distributed big data processing frameworks. The distributed computation logic in these applications greatly relies on data multicast, a data transfer pattern with which a piece of data is delivered to multiple destination servers. However, in these distributed frameworks, the state-of-the-art data multicast mechanisms are all based on application-layer multicast, in which data is delivered through unicast flows on top of an overlay network. This thesis proposes high-performance system components that solve the data multicast issue by leveraging hybrid data center networks.

In a hybrid data center network, the racks are connected via a circuit switch (or a circuit-switched network) in addition to the traditional packet-switched network. Circuit switches fundamentally change the multicast communication capability among the servers since they can be extended to support physical layer multicast. This thesis achieves the goal of high-performance from two critical aspects, i.e., multicast data transfer and multicast data scheduling.

In the first part, the thesis presents Republic, a complete platform providing high-performance “data multicast service” for applications running in hybrid data centers.
Republic consists of a Republic agent daemon running on each of the servers and a centralized Republic manager. The Republic agent (1) exposes a unified Republic API for the applications using the data multicast service, (2) talks with the Republic manager to request and return network resources for data multicast, and (3) achieves multicast data transfer efficiently and reliably. The Republic manager takes the multicast data scheduling algorithm as a plug-in module. Republic is implemented and deployed in a hybrid data center testbed. The testbed evaluation shows that Republic can improve data multicast in Apache Spark machine learning applications by as much as $4.0\times$.

In the second part, the thesis tackles the problem of scheduling multicast data transfer in a high-bandwidth circuit switch. The scheduling algorithm adopts the approaches of multi-hopping and segmented transfer. It aims at minimizing the average demand completion time to deliver the most benefit to the applications. The algorithm exhibits up to $13.4\times$ improvement comparing with the state-of-the-art solution.
Foremost, I am profoundly grateful to my advisor Prof. T. S. Eugene Ng for his guidance, inspiration, and support, on my way of getting the doctoral degree. The thesis would have been impossible without his help. I admire Prof. Ng for his unique way of educating his students. He trains his student toward a great thinker, not just a specialist, as the skill of independent thinking is essential for a Ph.D. During my Ph.D. study, I always enjoyed talking with Prof. Ng as he could guide my thoughts towards a better direction. Besides that, Prof. Ng put a significant amount of effort into improving my communication, presentation, and technical writing skills. I can’t imagine any other professor would give me such detailed instructions. I am also deeply grateful to Prof. Ng’s encouragement and patience. So many times I felt being supported when I made mistakes. I feel so lucky to choose Prof. Ng as my Ph.D. thesis advisor. I really enjoyed working with him.

I would like to express my sincere gratitude to the rest of my thesis committee, Prof. Alan L. Cox and Prof. Peter J. Varman, for their insightful comments on my research projects and careful revision suggestions on the thesis.

I would like to thank all members in our research group, Dr. Florin Dinu, Dr. Jie Zheng, Dr. Yiting Xia, Dingming Wu, Ruiqi Liu, Zhaolei Liu, Simbarashe Dinamarira, Xin S. Huang, Sushovan Das, and Afsaneh Rahbar. They kept giving me constructive feedback on my paper drafts and project ideas. I also greatly benefited from the information sharing and the in-depth technical discussion within the group.

My sincere thanks also go to Greg Roelofs, Dr. Zhuang John Wang and Dr. Guohui Wang for their mentorship and help during my internship at LinkedIn and Facebook and during my study at Rice University, respectively.
My heartfelt thanks go to my dearest friends, Dr. Yucheng Liu, Dr. Jacob Gao, Vicky Luo, Dr. Xiwei Yi, Ying Fang, Dr. Li Chen, Tong Zhang, Chen Luo, Ziliang Zhu and Wei Li. They always gave me the strongest emotional and spiritual support during my hardest time.

In the end, I would like to express my deepest love and thanks to my parents for their unconditional love and support, which allows me to enjoy the wonderful and unforgettable journey of becoming a Ph.D.
# Contents

Abstract ................................................................. ii
Acknowledgments .................................................... iv
List of Illustrations ................................................... x
List of Tables ........................................................... xiii

1 Introduction ............................................................. 1
   1.1 Focus and Contribution of the Thesis ................................ 5
       1.1.1 Data Multicast Platform ........................................ 5
       1.1.2 Data Multicast Scheduling ...................................... 7
   1.2 Thesis Outline ....................................................... 7

2 Background ............................................................... 9
   2.1 Data Multicast in Big-data Frameworks .............................. 9
   2.2 Circuit Switch ....................................................... 10
   2.3 Hybrid Rack-level Interconnections ................................. 12

3 Data Multicast Platform for Hybrid Rack-Level Interconnections .... 14
   3.1 Challenges .......................................................... 14
   3.2 Republic .......................................................... 17
       3.2.1 Reliable and Efficient Data Multicast Protocol ............ 20
       3.2.2 Republic Manager .............................................. 30
   3.3 Implementation ...................................................... 31
   3.4 Deployment ........................................................ 33
3.5 Evaluation ................................................................. 35
  3.5.1 Evaluation Workload ............................................... 35
  3.5.2 End-to-end Application Level Improvement with Republic .... 36
  3.5.3 Efficient Data Multicast with Tuned Attempt Sending ....... 40
  3.5.4 High Throughput of Republic Manager .......................... 43
3.6 Related Work ............................................................ 44
3.7 Chapter Summary ....................................................... 45

4 Data Multicast Scheduling in a High-Bandwidth Circuit Switch 46
  4.1 Challenges .............................................................. 46
  4.2 Problem and Approaches ............................................. 47
  4.2.1 Network Model ................................................... 48
  4.2.2 Data Multicast Demands ......................................... 49
  4.2.3 Scheduling Objective ............................................ 49
  4.2.4 The Scheduling Problem and Approaches ..................... 50
  4.3 Related Work .......................................................... 54
  4.4 Scheduling ............................................................. 55
  4.4.1 Create the Circuit Configuration and Choose the Demands to
  be Scheduled in an Epoch ........................................... 56
  4.4.2 Decide the Epoch Duration ...................................... 60
  4.5 Evaluation ............................................................. 61
  4.5.1 Simulation Setup .................................................. 61
  4.5.2 Greatly Improved Average DCT ................................ 64
  4.5.3 Effective Algorithm Features .................................. 65
  4.6 Chapter Summary ..................................................... 65

5 Future Works 67
6 Conclusion

Appendix A Switch Guide

A.1 Switch Architecture .......................................................... 72
  A.1.1 The Two-Layer Network ............................................. 72
  A.1.2 The One-Big-Switch Network ..................................... 75
  A.1.3 The Rice Research Network ..................................... 75
  A.1.4 Servers .................................................................. 75
A.2 Switch Operation .............................................................. 77
  A.2.1 EPS Operation ......................................................... 77
  A.2.2 OCS Operation ......................................................... 85
A.3 Switch Measurement .......................................................... 90
  A.3.1 Measurement System ............................................... 90
  A.3.2 Measurement Instruction ......................................... 92
  A.3.3 Measurement Results ............................................... 93

Appendix B Republic Tutorial

B.1 Republic API ..................................................................... 96
  B.1.1 register .................................................................. 98
  B.1.2 add ....................................................................... 98
  B.1.3 send .................................................................... 100
  B.1.4 read .................................................................... 101
  B.1.5 delete .................................................................... 102
  B.1.6 unregister .............................................................. 104
B.2 Experiment Components .................................................... 104
  B.2.1 Computation Framework – Apache Spark .................. 104
  B.2.2 Cluster Management – Apache Hadoop YARN ............ 106
  B.2.3 word2vec .............................................................. 106
  B.2.4 Latent Dirichlet Allocation (LDA) ............................. 109
B.2.5  TPC-H Benchmark ............................................. 112
B.2.6  Republic Agent .................................................. 113
B.2.7  Republic Manager .............................................. 115
B.3   Experiment Script ............................................... 116

Bibliography  ......................................................... 119
## Illustrations

1.1 Hybrid rack-level interconnections with different circuit switching technologies. Each of the ToR switches also connects to a fully connected packet-switched network, which is not shown for illustration simplicity. .......................................................... 3

2.1 Logical diagram of a circuit switch .................................................. 11
2.2 Example of extending an optical circuit switch (OCS) to support P2MPC .......................................................... 12
2.3 Hybrid data center illustration ...................................................... 13

3.1 System architecture of Republic .................................................. 18
3.2 Timeline of the sender and two receivers in a data multicast example in Republic. ...................................................... 19
3.3 Illustration for detecting packet loss. The ring represent the $sn$ space of a data. The sender sends the fragments in a $sn$ increasing order through multiple rounds. ...................................................... 28
3.4 The deployment of Republic in hybrid data center testbed based on optical circuit switching (OCS) (architecture in Figure 1.1(a)) .... 34
3.5 CDFs of broadcast object reading time in second .......................... 38
3.6 Application running time with different number of executors ......... 39
3.7 Efficient data multicast with attempt sending ............................... 41
4.1 An example of multicast demand scheduling. The example involves ten racks with $b_h=20$ Gbps, $b_l=10$ Gbps (a P2MPC can be shared among at most two multicast flows simultaneously) and a circuit switch with $\delta=1$ ms. (a) The input demands. (b) The optimal scheduling with multi-hopping and segmented transfer. The established P2MPCs are shown on all hops. The multicast flow is only marked on the hop it uses. (c) A comparison between segmented and non-segmented transfer (average DCT 412/6 ms vs 508/6 ms).

4.2 Example of solving conflicting racks. (a) Considering Demand 8 (D8) and D9 during an execution of Alg. 2. The algorithm executes to Iteration $i$ where the P2MPCs rooted from Rack 3 (R3), R4, R7, R8, and R10 are added to the network. In Iteration $i+1$, D8 is added to the schedule by extending the outputs of the P2MPC rooted from R7 to R9 and R10; in Iteration $i+2$, D9 is served by adding a new P2MPC rooted from R1 and connecting to R2, R3, and R4. (b) An example of the loop in the hypergraph. All the racks (nodes) can’t be reached by any racks (nodes) out of the loop by neither extending existing P2MPCs nor adding new P2MPCs.

4.3 Box plot of the speed-up of average DCT using Blast as the baseline. The center quartile and the ‘x’(and numbers) are median and average respectively; the upper(lower) quartiles and the upper(lower) whiskers shows 75(25)- and 95(5)-percentile respectively. Each box represents a hundred sets of input demands in the setting with $n=128$, $b_l=10$ Gbps. (a) Various circuit switch port bandwidth and traffic intensity. $\delta=1$ ms. (b) Various $\delta$ and algorithm policies. $b_h=100$ Gbps, traffic intensity=10 s.

A.1 Testbed switch architecture
A.2 Two-layer network ................................................. 74
A.3 EPS port connection reconfiguration (a) a reconfiguration that results
    in packets being buffered at the sender port. (b) a reconfiguration
    that results in no packet being buffered. ....................... 83
A.4 Timeline of a switch reconfiguration event ...................... 91
A.5 The system diagram for the switch performance measurement .... 91
A.6 EPS measurement results ........................................ 94
A.7 OCS measurement results ......................................... 94

B.1 Republic API message format ..................................... 97
B.2 register message payload ........................................ 99
B.3 add message payload ............................................ 100
B.4 send message payload ........................................... 101
B.5 read_reply payload ............................................. 102
B.6 Content in the RAM disk file of the data ...................... 103
B.7 delete_reply payload ............................................ 103
Tables

3.1 Republic API. Application processes use data multicast service via this API. The calling process provides its appID and processID when making these calls. .................................................. 20

3.2 Republic agent-manager interface. Used by agents and manager to request, response and release multicast paths. ................................. 21

3.3 Control messages in the data multicast protocol. A control message is for a specific multicast data transfer. So a message includes the appID and the dataID. .................................................. 22

4.1 Notations in the chapter. Lowercase and uppercase represent known and unknown variables respectively. ................................. 48

A.1 Server configurations .................................................. 76
A.2 Quanta EPS documents .................................................. 77
A.3 Glimmerglass OCS documents ........................................ 85

B.1 Republic API message type code ...................................... 97
B.2 Reply message status code ............................................ 99
B.3 Repositories used in the Republic experiment ....................... 105
B.4 Placeholders for the security information in the repositories .... 105
B.5 Variables in the command for submitting Spark application to YARN 108
B.6 Variables in the command for input data generator ............... 109
B.7 Variables in the command for running Republic agent . . . . . . . . 114
B.8 Variables in the command for running Republic manager . . . . . . 116
B.9 Critical configurations in the Republic experiment script . . . . . . . 118
Chapter 1

Introduction

We live in a world increasingly driven by big data. Maximizing the value of such a massive amount of data relies on large-scale distributed data processing. Data multicast, i.e. one-to-many data dissemination, is a critical data transfer pattern during the workflow of data processing [1, 2]. For examples, in the data preparation step, executing a database query having a join operation may need one table to be delivered to all the nodes having the partitions of the other table [3]; in the data analysis step, an iterative machine learning algorithm may require the updated training model to be copied to all its computation nodes before each iteration [1].

Data multicast is an expensive operation for traditional data center networks as it generates a high volume of traffic. Unfortunately, the big data processing makes a demanding request for high-performance data multicast. This is because (1) the fan-out of the data multicast is large, as a data processing job may need hundreds of worker nodes; (2) the size of the multicast data is large, as database tables and the models of machine learning jobs, e.g., natural language processing [4] and computer vision, can easily reach hundreds of megabytes or even gigabytes; (3) the data multicast happens frequently, as it happens in each iteration in iterative machine learning jobs and join operations in database queries.

Nowadays, large-scale data processing frameworks heavily rely on application-layer multicast mechanisms due to the lack of in-network multicast support in data center networks. For example, a variable in Spark can be delivered to workers using either
Cornet [1] which is a customized BitTorrent-like overlay protocol, or naive unicast where the sender transmits one copy of the data to each of receivers. However, these unicast-based mechanisms are far from the low cost and low latency requirement as these solutions inject too much traffic into the network especially when the multicast fanout goes up.

The emergence of circuit-switched rack-level interconnections revives the in-network multicast in data centers. These new types of interconnections can be built with circuit switching technologies such as optical circuit switch (OCS) [5, 6, 7], free-space optics (FSO) [8, 9] or millimeter wave (MMW) wireless [10], etc. Some of these interconnections have been extended to support physical-layer multicast. As shown in Fig. 1.1, the circuit switch based on OCS use optical power splitter to divide data signals from the input port to multiple output ports [11, 12, 2]; the circuit switch based on FSO use a cascade of half-reflection switch mirrors to divide light through partial reflection [13]; MMW-based circuit switch can direct wireless signal to a group of receivers through 3D beamforming antennas [14, 15, 16]. These circuit-switched rack-level interconnections and the traditional packet-switched rack-level interconnections altogether form the hybrid rack-level interconnection.

These circuit-switched rack-level interconnections overcome the intrinsic difficulties of in-network multicast in pure packet-switched data center networks [2]. The packet switches in these traditional data centers are organized in layers so the resulting multicast tree can be highly unbalanced, with receivers being at different hops from the sender through different intermediate switches. This creates different levels of bandwidth contention along the paths to different receivers, making congestion control a daunting challenge. Thus retransmission-based reliability mechanisms easily fall into the vicious cycle of generating greater congestion and more packet loss.
Figure 1.1: Hybrid rack-level interconnections with different circuit switching technologies. Each of the ToR switches also connects to a fully connected packet-switched network, which is not shown for illustration simplicity.

However, with a circuit switch, the simple and efficient multicast paths bypass the congested core layer of the packet-switched network. This massively eliminates unnecessary congestion and packet replication in the network. As shown in Fig. 1.1, on the multicast path, only the first and last hops are packet-switched hops on top-of-rack (ToR) switches and these ToR switches are directly connected through the circuit switch. The circuit hops between the ToR switches are dedicated so that there is no bandwidth contention. With judicious data multicast scheduling and network control, the contention in the ToR switches can be minimized [11, 2].

Circuit switches work in a way that significantly different from packet switches. In a circuit switch, the physical-layer signal is passively directed from an input to a configured output port without decoding the content or generating the signal. Such fundamental difference makes the circuit-switched rack-level interconnection massively surpass the rack-level interconnections purely built with packet switches.
in the following aspects. From the perspective of sustainability, first, the per-port power consumption of a circuit switch is lower than the packet switch by at least an order of magnitude. For example, a 48-port 10 GbE electrical packet switch consumes 180 Watt, while a 192-port optical circuit switch only consumes 50 Watt. Second, the circuit switch is almost agnostic to the bandwidth of the signal so that a network-wide link bandwidth upgrade does not require a replacement of the existing circuit switch. This saves a lot of operational costs both on hardware and software. From the perspective of scalability, it is possible to build a single circuit switch with hundreds of or even thousands of ports, which is able to connect all the racks in a medium-scale data center. These superior advantages of circuit switches have driven a number of the emergence of the research in circuit-switched rack-level interconnections [5, 6, 7, 8, 9, 10, 17]. I believe that hybrid rack-level interconnections will eventually be deployed in the next generation of data centers for their potential in leveraging the advantage of both circuit switches and packet switches.

With the confluence of the prosperity of big data and new network innovations, now it is the right time to revisit data multicast to bridge the gap between the highly desirable physical-layer multicast capability provided by the circuit switch and the highly inefficient application-layer multicast mechanisms adopted by the distributed data processing frameworks. However, hybrid rack-level interconnections depart from the old assumptions of pure packet-switched interconnections. Without a dedicated system-level support, the new network infrastructure cannot be used in a plug and play manner. The major challenges in constructing this system are to provide the appropriate interface for the data center applications and hide the underlying details in achieving the high-performance data multicast.

The goal of this thesis is to build a high-performance multicast data transfer
system for the data center equipped with hybrid rack-level interconnections so that the distributed big data processing frameworks can transfer multicast data in a more efficient way.

1.1 Focus and Contribution of the Thesis

This thesis sets itself apart from previous groundbreaking foundation works staying at the level of data center architecture. It is the first effort to solve the system-level issues in leveraging the multicast-capable hybrid rack-level interconnections in data center networks. The thesis proposes the state-of-the-art solutions with the best performance comparing against existing literature. In the long term, the works done in the thesis can be set as the comparison baselines of future following works.

The thesis proposes the solutions from the perspective of the “users” of the multicast-capable data center network infrastructure. The “users” refer to the system researchers and software developers who build the distributed big data processing systems and frameworks. These specialists should take care of the network details as less as possible and focus on the functionality and the efficiency of the distributed data processing logic, while the functionality and efficiency on the network side should be handled by network specialists. Under such a design philosophy, the thesis proposes two works, i.e., a data multicast platform providing a data multicast service to distributed applications in the data centers and a data multicast scheduling algorithm leveraging the high-bandwidth of the circuit-switch rack-level interconnections.

1.1.1 Data Multicast Platform

In order for the distributed data processing frameworks to use the multicast capability of the data center network infrastructure, the most urgent step is to provide an
appropriate interface for these frameworks to call. This interface must be easy to use so that the underlying details on the network side are agnostic to the users. Hence, toward that objective, the first work of this thesis is Republic, a data multicast platform providing data multicast service to the distributed data processing frameworks. Republic abstracts all the underlying network details and complexity in an efficient way and achieves the high-performance data multicast service.

Although Republic provides an easy-to-use interface, it is challenging to design a unified data multicast service that employs the ever-emerging multicast enabling technologies. This is because such a platform must address the following system issues (but not limited to these issues). (1) Different circuit switching technologies have different and unpredictable circuit reconfiguration times. How to efficiently transmit multicast data as soon as the multicast path is established? (2) Circuit switch provides dedicated links with high bandwidth capacity, e.g. 10 Gbps or even higher. How to enable the servers to send and receive multicast data at a high rate? (3) Although circuit switch links are generally reliable, packet loss can still happen. How to achieve reliability at small overhead under the context of hybrid data center networks?

Republic goes one huge step beyond pioneer works [2, 12, 11, 13] which merely demonstrated the potential of supporting data multicast in hybrid data centers. Republic is the first effort towards addressing these system-level challenges and providing a full-fledged solution as a system plug-in for a data center. Republic has been deployed in a hardware testbed having 40 servers. Apache Spark is adopted as an example to use Republic’s data multicast service. Compared to the data multicast mechanisms adopted in Spark, Republic can speed up the end-to-end data multicast performance by as much as $4.0 \times$ in iterative machine learning algorithms and
database queries.

In the long run, Republic is also a system support for researchers and developers to investigate various data multicast scheduling policies and to innovate distributed data processing frameworks adopting in-network multicast support.

1.1.2 Data Multicast Scheduling

In order for the circuit switch to adapt to the current inter-rack traffic demands, a scheduler is a required component because, unlike packet switch, a circuit switch is not able to by itself decide the output ports for the input traffic unless the scheduler configures the circuits. More importantly, the scheduler plays a crucial role in achieving high-performance data multicast because the order and the concurrency in serving the multicast data greatly impact the transfer completion time.

Driven by the urgent need of high-performance data multicast and the desirable physical-layer multicast capability of the circuit switch, the second work of the thesis proposes Monarchy, a scheduling algorithm for multicast data transfer in a high-bandwidth circuit switch.

Monarchy adopts multi-hopping and segmented transfer as the approaches to (1) fully utilize the high bandwidth, (2) overcome the fanout limit of the physical-layer multicast path and (3) effectively reduce the average completion time. The scheduling problem is formulated as an optimization problem. Monarchy out-performs the state-of-the-art by up to $13.4 \times$ in a simulation setup.

1.2 Thesis Outline

The thesis is structured as follows. Chapter 2 introduces the background of data multicast and hybrid data center networks. Chapter 3 presents the data multicast
platform for hybrid data centers. Chapter 4 presents the multicast data scheduling algorithm for high-bandwidth circuit-switched rack-level interconnections. Chapter 5 discusses future works and potential improvement to the work done in the thesis. Chapter 6 concludes the thesis. The thesis also includes the instruction and tutorial for the testbed and the Republic platform. Appendix A is the instruction of using the testbed. Appendix B is the instruction to Republic, the data multicast platform.
Chapter 2

Background

2.1 Data Multicast in Big-data Frameworks

Data multicast is the request that the same data is sent from one server to multiple servers in a data center. Multicast is a prevalence traffic pattern created by big-data processing applications running on these distributed computation frameworks. These applications rely on the distributed computation frameworks to deliver the data to multiple workers. This section presents two data multicast examples in an iterative machine learning application and a database query, respectively. This is to provide a deeper understanding of the multicast data in big data processing so as to demonstrate the importance of optimizing multicast data transfer.

Iterative machine learning application: Most machine learning jobs are to train models iteratively, e.g., artificial neural network, linear regression, and k-means clustering. When the input training data is very large in size, one single server can neither hold the entire dataset in memory nor quickly finish a training iteration for all the data samples. To speed up the training time and scale up the input training data size, big-data processing frameworks support a way to train the machine learning models in a distributed manner. In distributed training, the input data is partitioned into batches, each of which is stored on a server. At the beginning of each training iteration, the model needs to be sent to all the workers so as for them to calculate the update to the model using the data in the local batch.
In these machine learning jobs, the multicast data is the model. The size of the model depends on the number of parameters to be trained and it can reach hundreds of megabytes or even several gigabytes in some really complex models such as many NLP (natural language processing) models. For large datasets, the training job may run on tens of or even hundreds of server. Such large data size and high fan-out make the model delivery an operation with exorbitant cost. Unfortunately, model’s multicast happens in every training iteration. Thus, the machine learning applications have demanding requests to the data multicast capability of data center networks.

**Database query with join operation:** In production, large data sets are usually stored in databases. SQL queries on the database tables are frequently executed for analyzing the data or creating new database tables, such as table storing the machine learning training data. Replicated join (broadcast join) is one of the options in executing the SQL join in a distributed manner. When joining two tables using the replicated join, the partitions of one table are stored on multiple servers. The other table is delivered to these servers for executing a join with the local partition.

In this scenario, the multicast data is a database table. Although this multicast table is usually the smaller table among these two, the “smaller” table could be still large. The fan-out of the multicast is decided by the number of partitions of the other table. Similar to the case in the machine learning applications, in the SQL query, data multicast can be frequently triggered and it could introduce a large amount of traffic to the data center network.

### 2.2 Circuit Switch

A circuit switch works in a way that is fundamentally different from a packet switch. The circuit switch simply forwards the physical layer signal without (de)modulation
and (de)coding it, which means the switch does not look at the content of the packets. Figure 2.1 shows the logical diagram of a typical circuit switch. A circuit switch has multiple input and output ports. When the physical layer signal going into the circuit switch it is converted to the signal type that the circuit switch is able to switch among different output ports. For example, when using an optical circuit switch, the signal going into an input port is converted to a light beam; when using a free-space optics circuit switch, the signal is converted to a free-space laser beam; when using a millimeter wave circuit switch, the signal is converted to millimeter wireless signal. The circuit switches are to direct these signal to its output ports. Thus, logically, the input ports of a circuit switch can be configured to connect to its output ports. However, there is no statistical multiplexing in circuit switches since an output port is just connected to a single input port.

When reconfiguring the circuit connections in the circuit switch from one configuration to another, there is a time cost $\delta$ called circuit reconfiguration time. The reconfiguration time is different in different circuit switching technologies and it varies from tens of $\mu$s to tens of $ms$. Thus, $\delta$ is not negligible, which means that frequently reconfiguring the circuit may degrade the data transfer performance as a significant amount of the time may be spent in circuit reconfiguration.
A circuit switch may have its unique way in building a circuit connecting an input port to multiple output ports simultaneously. Such a circuit connection is called a port-to-multi-port circuit (P2MPC). Figure 2.2 shows an example of how to build a P2MPC based-on an optical circuit switch. The essential point to build the P2MPC on an OCS is to leverage optical splitters. An optical splitter is a passive optical device that is capable of dividing the input optical signal evenly to multiple output ports. The optical splitter is connected to the input and output ports on the OCS. The OCS builds the circuit that forwards the light to the splitter and the circuits that forward the output lights of the splitter to the OCS output ports connecting to the receivers.

2.3 Hybrid Rack-level Interconnections

Nowadays, the servers in a data center are organized in racks. In each rack, there could have tens of servers connecting to a packet switch, which is called the top-of-rack (ToR) switch (usually this switch is put at the top of the rack and the servers are installed beneath the switch). The rest of the data center network is to connect these
ToR switches so as for the servers to transfer inter-rack traffic. Such a network is the rack-level interconnection. Traditionally, the data center network is constructed with pure packet switches. The hybrid rack-level interconnection is to have a circuit switch connecting the ToR switches in addition to the pure packet switched network.

The logical architecture is shown in Figure 2.3. On each ToR, there are links connecting to the packet switched network and a link connecting to the circuit switch. Having a hybrid rack-level interconnection allows the data center network to take advantage of both packet switching and circuit switching. The packet switched network can provide a low latency and fully connected network at all time; the circuit switch can provide high-speed data transfer under low energy consumption. In this thesis, a data center with hybrid rack-level interconnection is called a hybrid data center for short.
Chapter 3

Data Multicast Platform for Hybrid Rack-Level Interconnections

This chapter presents Republic, the data multicast platform for data centers having hybrid rack-level interconnections. The chapter is organized as follows. Section 3.1 discusses the challenges in building the platform. Section 3.2 presents the technical details of Republic. Section 3.4 shows the deployment of Republic to a computation cluster. Section 3.5 evaluates the performance of Republic. Section 3.7 summarizes Republic.

3.1 Challenges

Building a data multicast platform is not straightforward. Leveraging the multicast capability in hybrid data centers faces the following challenges.

**Expertise gap between big-data processing and network:** The rise of big-data processing has fundamentally changed the dynamic between networks and their users. In the past, users who produced large amounts of network traffic also had the expertise to optimize the data transfers in their applications. However, today’s data scientists and machine learning engineers, who often are not network specialists, regularly use cluster computation frameworks to run data processing and machine learning jobs that produce large amounts of traffic. For example, a SQL query on a large data set can easily produce several GBs of multicast data (Section 3.5.1). The prevalence and scale of data multicast in these jobs necessitate more efficient handling
of data multicast. For example, if a job has multiple processes on a server, these processes should share a single data transfer instead of having multiple transmissions. Similarly, multiple senders on the same server require coordination to share network resources. Unfortunately, these desirable features all require network expertise which the software engineers building the cluster computation frameworks do not usually possess. Bridging this expertise gap is essential if big-data processing is to fully exploit emerging network architectures having efficient multicast support. Under such situation, abstraction is usually viewed as a promising bridge. With abstraction on a hybrid data center network, software engineers building the cluster computation framework can keep their focuses, while the network experts work concurrently to guarantee data multicast efficiency. However, it remains a challenge to find the right amount of abstraction that allows effective collaboration while reducing efficiency loss from abstraction.

**High-rate transfer:** On a circuit switch, a circuit is dedicated to the path from the input to the output, i.e., the output of a circuit can only receive traffic from a single input on the other side of the circuit. This property of circuit switch is fundamentally different from a packet switch whose output port can be shared by the flows from multiple input ports through statistical multiplexing. Therefore, transmitting the flow at a high rate is crucial to fully utilize the dedicated circuits [18, 19, 20]. The end-to-end high-rate multicast transmission needs to overcome many obstacles both at the end-point servers and within the network: server bandwidth may be simultaneously shared by multicast and unicast flows; network stack overhead prevents processes from transmitting data at a high rate; congestion may happen to the multicast flow at the receivers’ ToR switches.

**Reliable data delivery:** In hybrid data centers, packets can be lost due to
various reasons. For example, packets can be corrupted due to low signal quality after
power split or during circuit reconfiguration; packets could be dropped if the receiver
cannot process them at the rate of the incoming multicast flow; the output queue of
last hop switch port may drop packets in multicast flows due to the congestion with
other flows. The design of the data multicast protocol should consider and minimize
all sources of packet losses. Once losses occur, how to retransmit lost packets is still
an open question. There are questions such as whether the retransmission should
use the multicast path and whether the retransmission should use a reliable or an
unreliable transfer. The solution should consider the properties of both the multicast
path and the packet-switched unicast paths between the sender and receivers.

Quick coordination between transmission and multicast path setup: Ideally, the transmission should start immediately after the multicast path is set up. This
coordination must be quick because any time lag results in a large bandwidth waste
given the high link capacity offered by circuits. However, for different circuit switch-
ing technologies, the circuit reconfiguration time ranges from tens of $\mu$s [7] to tens of
$ms$ [6]. Even for a single circuit switch, reconfiguration time of each circuit also varies
within a reconfiguration and between reconfigurations. In addition to that, around
the end of a circuit reconfiguration, the circuit may experience a period of transient
state before the circuit is stably connected. During the transient circuit state, the
physical signal strength may be unstable and oscillate due to the ringing effect [21],
which results in an intermittent circuit connection. Even a packet can be delivered to
a receiver, it doesn’t mean the circuit carrying the packet is stable. This makes the
coordination even harder. At this stringent sub-second scale, hardware-level coordi-
nation would be favorable for speed, but no commercial hardware support is available
today as far as I know. A software-level solution is desirable for flexibility, compat-
ibility, and cost, but could be prone to more overhead. The design of an efficient software-level coordination remains unknown. Again, due to the variability in the circuit reconfiguration time, receivers could start receiving packets at different times. This results in different receiving states among receivers. The challenge is to build a data multicast protocol with which a receiver just connected to the multicast path could benefit from the ongoing transfer without interrupting the already connected receivers and without introducing unnecessary data transfer and extra delay.

### 3.2 Republic

Republic addresses all the challenges in leveraging the data multicast capability in hybrid data center networks. Figure 3.1 shows the system architecture of Republic. Republic includes an agent process on each of the servers and a centralized multicast manager (Section 3.2.2). The Republic agent exposes a unified API (Table 3.1) for the data processing applications to request multicast data transfer. The agent handles the transfer using a reliable and efficient data multicast protocol (Section 3.2.1) tailored for hybrid data center environment and requests the multicast path via the Republic agent-manager interface (Table 3.2). The Republic manager is responsible for managing the network resources used for building multicast paths so as to schedule the requested data multicast and configure multicast path. Before diving into the design details of each of the components, the interaction among applications and the Republic components are shown first in order to give a high level impression.

**Interaction between application process and local Republic agent** (Table 3.1): Each application generates a 16-byte universally unique identifier (UUID) as its unique identifier \texttt{appID}, which is known to all the processes of the application. To use Republic, each application process needs to \texttt{register} with the local agent.
Figure 3.1 : System architecture of Republic

The application assigns each multicast data with an application-wide unique 8-byte dataID. Republic decouples data transfer with data reading/writing so that all the multicast data transfer can be handled by the agent in an efficient way and be transparent to the applications. Before requesting for sending data to a set of receivers through send, the application process makes the data accessible by the local agent (Section 3.2.1) and notify the agent through add. The receiver process calls read to request the data from the local agent.

**Interaction between Republic agent and Republic manager** (Table 3.2): In Republic, only the sender’s agent talks with the Republic manager to request and return multicast paths since the sender knows the list of receivers. Allowing the receivers to talk to the manager leads to much more message passings between the agents and the manager, which degrades the throughput of the manager. A data multicast starts with the sender agent requesting a multicast path from the Republic manager through request. The manager replies the agent (via response) with the
Figure 3.2: Timeline of the sender and two receivers in a data multicast example in Republic.

scheduling decision (accepted or denied) made by the scheduling algorithm running on the manager. The manager sends the accepted response to the sender agent only when the scheduling algorithm allows the multicast transfer to start. After the sender agent receives the response, it starts sending the data immediately using the reliable data multicast protocol (Section 3.2.1). The sender agent calls release to return the multicast path back to the manager once the transmission in the multicast path finishes. To support the widest range of different scheduling policies, Republic allows a multicast data transfer to be completed in multiple transmission sessions. This means the scheduling algorithm may accept partial data size for each request [22]. If the sender agent receives a response partially accepting the requested data, the sender agent should send another request for the remaining data immediately.
<table>
<thead>
<tr>
<th>Interface</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>register</td>
<td>Register with the local agent before using the data multicast service.</td>
</tr>
<tr>
<td>unregister</td>
<td>Unregister from the agent. The calling process cannot use the data multicast service after it unregisters.</td>
</tr>
<tr>
<td>add</td>
<td>Add the multicast data to the agent after the process has written the data to the in-memory file system. The process provides <code>dataID</code> of the multicast data.</td>
</tr>
<tr>
<td>send</td>
<td>Request to send the multicast data to a set of receivers. The application process should add the data to the agent before calling <code>send</code>. The process provides <code>dataID</code> and the list of <code>serverIDs</code> of the receivers.</td>
</tr>
<tr>
<td>read</td>
<td>Request to read the multicast data from the agent. Return with file reading instruction when the data is ready to read.</td>
</tr>
<tr>
<td>delete</td>
<td>Delete the multicast data from the in-memory file system. Application processes should coordinate to make the call if the data is no longer needed by the processes on the server. The process provides <code>dataID</code>.</td>
</tr>
</tbody>
</table>

Table 3.1: Republic API. Application processes use data multicast service via this API. The calling process provides its `appID` and `processID` when making these calls.

### 3.2.1 Reliable and Efficient Data Multicast Protocol

The reliable data multicast protocol is a crucial component in Republic since it directly impacts the performance of multicast data transfer. The protocol runs between
<table>
<thead>
<tr>
<th>Interface</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>manager .request</td>
<td>Called by the sender agent to request the multicast path from the manager. The agent provides appID, dataID, datasize, remainingDatasize, serverID of the sender, a set of serverIDs of the receivers and a locally generated unique requestID.</td>
</tr>
<tr>
<td>agent .response</td>
<td>Called by the manager to notify the sender agent about if the requested multicast path is accepted or denied. Besides the scheduling decision, the manager also provides a responseID created by the manager, the requestID from the sender agent, and accepted size.</td>
</tr>
<tr>
<td>manager .release</td>
<td>Called by the sender agent to return the received multicast path back to the manager. The agent provides the requestID in the corresponding agent .request call and the responseID that accepted the request.</td>
</tr>
</tbody>
</table>

Table 3.2: Republic agent-manager interface. Used by agents and manager to request, response and release multicast paths.

The sender and the receivers of each single data multicast. The protocol uses a data channel and a control channel (Figure 3.1) that leverage the properties of the multicast path and the unicast paths respectively. The data channel uses the multicast path to deliver the data content since the multicast path can deliver the data to multiple receivers unidirectionally within a single transmission. The data channel uses UDP packet for efficient connectionless sending and receiving. The control channel is for delivering the protocol control messages (Table 3.3) between the sender and the receivers. The control messages are small unicast messages and require low latency.
### Table 3.3: Control messages in the data multicast protocol

A control message is for a specific multicast data transfer. So a message includes the `appID` and the `dataID`.

<table>
<thead>
<tr>
<th>Control messages</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>INTL_SEQ</strong></td>
<td>Receiver tells the sender the initial <code>sn</code> it receives. The message includes the initial <code>sn</code>.</td>
</tr>
<tr>
<td><strong>DATA_RCVD</strong></td>
<td>Receiver acknowledges the sender that the data has completely received.</td>
</tr>
<tr>
<td><strong>DATA_FNSD</strong></td>
<td>Sender tells the receiver that the data channel has finished data sending.</td>
</tr>
<tr>
<td><strong>PTCH_REQ</strong></td>
<td>Receiver tells the sender the <code>sn</code> range of a detected packet loss. This message contains a pair of (starting <code>sn</code>, ending <code>sn</code>).</td>
</tr>
<tr>
<td><strong>PTCH_DATA</strong></td>
<td>Sender replies <code>PTCH_REQ</code> with the requested data fragment.</td>
</tr>
</tbody>
</table>

The data is logically divided into fragments with a fixed size (except for the last fragment) that fits into the payload of a single Ethernet frame. Fragments are assigned contiguous 8-byte sequence numbers (`sNs` for short) starting from 0. Each of the data packets has a header including the `appID`, `dataID` and `sn` to identify the fragment.

The data size is only put into the data packet with `sn 0` to minimize overhead. The receivers can always get the data size in this way since all data objects have the fragment with `sn 0`. For the case having multiple transfer sessions, the receiver knows the `sn` containing the size for the next transfer session is the next `sn` after the largest `sn` of the previous session.
The sender transmits data packets to the data channel in the increasing order of \textit{sn}. However, the receivers on different branches of the multicast path could begin receiving data at different times due to the unpredictability and variability of the circuit reconfiguration time (Section 3.1). Therefore the sender sends the data packets over the data channel in a wrap-around manner until the data packets are sent to all the receivers. In Figure 3.2's example, the sending start from \textit{sn} 0 to 99. After \textit{sn} 99 is sent, the \textit{sn} goes back to \textit{sn} 0 and continues another round of sequential sending.

The receiver tells the sender the initial \textit{sn} it received via \textbf{INTL_SEQ} and notifies the sender about receiving completion via \textbf{DATA_RCVD}. Only these two control messages are required for each receiver.

**Coordinating Transmission with Circuit Setup**

Network hardware does not have an end-to-end view of the multicast path connectivity. In a naive solution, after the sender receives the \textit{accepted response}, it starts sending after a predetermined amount of waiting time. Unfortunately, this waiting time has to be conservatively large, which leads to unnecessary waiting at the sender. Alternatively, the sending starts right after the \textit{accepted response} is received. However, before the path becomes stable, the sender bandwidth and CPU cycles are wasted in sending data packets to a disconnected or intermittently connected multicast path. Moreover, sending high-rate packets to a circuit in a transient state results in a considerable amount of packet losses. This is even worse than receiving no packet because the received and lost packets are interleaved, which results in a dilemma where either discarding the received packets or retransmitting the lost packets results in extra sender overhead.

Republic adopts a software-based mechanism to detect the connectivity of the
multicast path. It makes a good tradeoff among efficiently utilizing the multicast path, reducing redundant data packets and minimizing packet loss during transient circuit states. The sender starts with an attempt sending phase right after the requested multicast path is accepted, as shown in Figure 3.2. During this phase, data packets are sent at a fixed time interval $i_a$ and are used as probes to test the connectivity of the multicast path.

The value of $i_a$ depends on the reconfiguration time of the circuit switch and can be determined by the network operator. If $i_a$ is too short, the sender sends too many redundant packets during the circuit reconfiguration and the transient circuit state, which results in a considerable amount of packet losses. If $i_a$ is too long, the notification to the sender about multicast path connectivity is delayed, which results in a large delay in starting high-rate transmission and inefficient usage of the multicast path. Deployment experience suggests that an attempt sending interval between 1% and 3% of circuit reconfiguration time achieves a good tradeoff (Section 3.5.3). A carefully chosen $i_a$ only slightly increases the data channel transfer time beyond the theoretical minimum time. For example, in a data center network with 10 GbE server bandwidth, 9KByte jumbo frame and 0.5 ms round-trip time, the extra delay due to attempt sending is only 0.85 ms when $i_a$ is 0.7 ms (10Mbps) and the CPU usage during attempt sending is less than 5% on a single core. For a multicast data with 500 MB, this extra delay only accounts for 0.2% of the ideal transmission time.

When the multicast path is in the transient state, receivers may get the initial $sn$. In Figure 3.2’s example, sender starts attempt sending from $sn$ 0 and receiver 2 gets initial $sn$ 58. However, starting a high-rate transmission (Section 3.2.1) at this moment is too early since most part of the multicast path is still in the transient state. To minimize packet losses due to the transient state, the sender starts high-
rate sending after it collects INTL_SEQs from each of the receivers. In Figure 3.2’s example, the high-rate sending starts after sender gets INTL_SEQ from receiver 1. Collecting the INTL_SEQ from all receivers guarantees that the sender knows which sns each receiver should have received from the data channel, so the sender can stop the data channel transmission after it has sent at least one round to each of the receivers. Therefore, each of the data fragments are sent to each of the receivers via the data channel at least once. In Figure 3.2’s example, the sender loops back to send sn 0 onward and stops after it sends sn 59 in the next round.

The implication of such a coordination mechanism is that after a receiver successfully receives the first data packet, the multicast path from the sender to this receiver will be stable. However, the first data packet may still be received during the transient state of the multicast path. Thus, the Republic agent does not start the high-rate sending right after the sender receives the first INTL_SEQ message. Instead, the sender keeps attempt sending until it collects INTL_SEQ messages from all the receivers. This leaves more time for the multicast path to become stable. In an alternative approach, the receiver can report to the sender when it gets multiple data packets with consecutive sequence numbers. By doing this, at the time the receiver reports to the sender, it has higher confidence in the stability of the multicast path. However, waiting for multiple packets also introduces latency before starting the high-rate sending. Thus, it seems that a more complicated coordination mechanism could only bring marginal benefit. Republic adopts the simplest coordination mechanism and leaves potentially better mechanisms as future research, which could be useful for the case where the signal power is insufficient during the circuit reconfiguration or the physical circuit connection experiences long transient state.
Sending and Receiving Data Packets at High-rate

Republic addresses the challenges in achieving high-rate multicast data transfer in three aspects.

*High-rate in multicast path:* The outgoing unicast flows from the multicast sender and the incoming unicast flows to the multicast receivers may compete with the multicast flow for bandwidth. On one hand, reserving all server bandwidth for multicast flows starves the unicast flows. On the other hand, suppressing the rate of multicast flow decreases the utilization of multicast paths, in which the circuit hops are dedicated for the multicast flow. Republic makes a good trade-off between high circuit utilization and fairness between unicast and multicast flows. At the sender side, the agent sends multicast flows in a best-effort manner and allows the multicast flows share the bandwidth with unicast flows fairly. Therefore, when there is no unicast flow going out of the sender, the multicast flow can be sent at line-rate of the server bandwidth. At the receiver side, congestion could happen between the multicast flow and unicast flows at the ToR switch ports connected to the receiver since there are unicast flows coming from other ToR ports. As the number of receivers increases, the congestion becomes worse because the chance of having flows sharing the receivers’ ToR switch ports increases. Allowing other flows to arbitrarily interfere with multicast flows at the receiver sides causes a large number of multicast packet losses and corresponding retransmissions. Republic gives high priority to the multicast flows on ToR switches to protect the multicast data packets from being dropped due to congestion. This can be achieved by setting a high priority value to the forwarding rules for multicast flows.

*High-rate in forwarding packets between Republic agent and server NIC:* Republic adopts kernel-stack-bypass frameworks [23] to forward the data packets between the
agent process and the server NIC to reduce CPU overhead and the number of memory copies. Republic also uses Ethernet jumbo frames to reduce the number of packets that the agent needs to process, so as to reduce the number of system calls. Republic deployment experience shows that using a kernel-stack-bypass framework can improve the multicast data transmission rate from 5 Gbps to full line-rate in the testbed with a 10 Gbps network.

*High-rate in data reading/writing:* To send/receive data at high-rate, the agent should be able to read/write the data at high-rate as well. The bandwidth of a modern server NIC can be 10 Gbps, or even 40 Gbps on high-end servers. However, the read/write speed of a hard disk drive(HDD) is at most 2 Gbps; even a solid state drive(SSD) cannot reach 10 Gbps. Thus, data cannot be transferred at high-rate if it is stored in on an HDD or an SSD. Nowadays, commodity DDR3 memory can support at least 51.2 Gbps read/write speed and DDR4 memory can even support read/write at 153.6 Gbps. Thus, Republic uses a dedicated in-memory file system and stores the multicast data as an in-memory file to enable high-speed access at memory bandwidth. The sender agent reads the fragments from the in-memory data file added by the application. The receiver agent sequentially writes the data fragments into an in-memory data file in the order they are received from the data channel. In Figure 3.2’s example, the file starts from sn 60 to 99 and then from sn 0 to 59. When reading the data, the application process starts from the position of sn 0 to the end and then from the beginning of the file to the position of sn 0.

**Recovering Lost Packets Efficiently**

According to the deployment experience, three common factors lead to multicast packet losses in hybrid data centers and Republic is designed to avoid and mini-
mize all these types of losses. First, packets are corrupted due to insufficient signal power (type 1 loss). Type 1 loss mostly happens during the transient circuit state where the power of the signal oscillates up and down. These corrupted packets are dropped by the receiving ToR switches so that it affects only the receivers in that rack. Republic reduces type 1 loss using attempt sending (Section 3.2.1). Second, the receiver process may be temporarily too slow to keep up with high packet rate (type 2 loss). The receiver drops the packets due to buffer overflow. Type 2 loss is not correlated because the packets are dropped at individual receivers. The dropped packets usually have contiguous sn values. Republic effectively suppresses type 2 loss by using the kernel-stack-bypass framework to send/receive multicast data packets efficiently (Section 3.2.1). Third, unicast flows sent to the multicast receivers contend with the multicast flow leading to packet losses (type 3 loss). The ToR switches drop packets at the queues of the congested output ports due to overflow. Type 3 loss is not correlated since the packets are dropped at individual switch ports connected to the receivers. Republic eliminates type 3 loss by assigning the multicast flow with a high flow priority (Section 3.2.1). In summary, Republic is designed to suppress these
common sources of packet losses. Republic does not assume a loss-free environment in a multicast data transfer because uncontrollable general packet corruption and type 1 and type 2 losses may occur, albeit very rarely.

Based on the above observations, packet losses in Republic are not correlated and rare, and hence Republic adopts a simple but efficient mechanism to recover the lost fragments in a point-to-point manner. In Republic, recovery of lost fragments is handled by control channel messages. Once the receiver detects a packet loss, it immediately requests the retransmission of the fragments from the sender.

To check packet losses, the receiver maintains three sn pointers to the fragments received from the data channel, i.e., initial, prior and current sn (Figure 3.3). The current sn is the just received sn from the data channel. The prior sn is the sn received prior to the current sn from the data channel. The initial sn and the prior sn divide sn space into two exclusive ranges. The “sent range” contains all thesns that have been sent to the receiver; the “expected range” contains all the sns yet to be sent to the receiver.

The receiver checks for packet loss whenever it receives a new sn from the data channel. The current sn has three possible cases in the sn space. If the current sn is in the expected range and it is the sn right after the prior sn (case 1), then there is no packet loss. If the current sn falls in other places in the expected range (case 2), the receiver knows that sns in the range of (prior, current) are lost. The receiver sends a PTCH_REQ message with the pair (prior, current) to the sender for retransmission, where prior is the starting sn and current is the ending sn of the loss range. If the current sn falls in the sent range (case 3), the receiver knows that sns in the range of (prior, initial) are lost. The receiver sends a PTCH_REQ message with the pair of (prior, initial) to the sender for retransmission and stops receiving the
data from the data channel. In the case where the receiver loses the last s\(n\)s and the sender has stopped sending, the receiver cannot detect such packet losses since there are no more packets being received. To handle this situation, the sender sends a `DATA_FNSD` to the receiver once the sender has sent all the s\(n\)s to the receiver and has not received `DATA_RCVD` from that receiver. The `DATA_FNSD` is sent after a timeout. After receiving the `DATA_FNSD`, the receiver sends (prior, initial) to the sender in a `PTCH_REQ`. Therefore, with such mechanism, *packet losses can always be detected by a receiver*. In Figure 3.2’s example, **receiver 1**’s initial s\(n\) is 60. When the current s\(n\) is 76 the prior s\(n\) is 73. The receiver detects a packet loss since the current s\(n\) falls in the expected range but is not the next s\(n\) after the prior s\(n\). The receiver immediately sends `PTCH_REQ` with the pair of (73,76) to the sender for retransmission.

The sender responds to the `PTCH_REQ` with `PTCH_DATAs`. Each of the `PTCH_DATAs` contains a fragment from the starting s\(n\) to the ending s\(n\) (not including the boundary). Although the receiver doesn’t know the number of lost packets if the detected loss range covers s\(n\) 0 (s\(n\) 0 contains data size), the sender knows exactly the lost packets given the pair of s\(n\) pointers in the `PTCH_REQ`. Therefore, *all the lost packets can always be reliably delivered*. This guarantees that multicast data can be correctly received by the receivers. In Figure 3.2’s example, the sender retransmits s\(n\) 74 and 75 via two `PTCH_DATAs`. To keep writing future fragments to the received files at high-rate, the receiver writes file holes for the lost fragments so that the writing is not blocked. The holes will be overwritten by the fragments received from `PTCH_DATAs`.

### 3.2.2 Republic Manager

The data multicast scheduling decision is made by the scheduling algorithm running on the Republic manager. The scheduling algorithm can be specifically designed for a
particular type of hybrid data center architecture or for different scheduling objectives. For example, the previous work [2] proposed a scheduling algorithm for OCS-based hybrid data centers. Thus, designing data multicast scheduling algorithms for hybrid data centers is out of the scope of this platform. Being a universal framework, Republic allows a scheduling algorithm to run on Republic manager as a plug-in module. This allows Republic to support for various hybrid data centers.

To make scheduling decisions, these algorithms need to know the availability of the network resources for building multicast paths and the data multicast requests issued by the agents. Republic manager provides a library for the scheduling algorithm to access this information. For the network resources, the manager maintains the availability of the ToR ports connecting to the servers and the circuit switch, the circuit switch ports, and the multicast devices (e.g., optical splitter or half-reflection mirror). To avoid circuit reconfiguration overhead, the manager also remembers the circuit connections on the circuit switch and the configuration on the ToR switch (i.e. multicast forwarding rules) so that a new configuration can reuse the existing circuits. The maintained states are updated once the algorithm accepts the request or the agent releases the multicast path. To improve parallelism, the manager simultaneously accepts the request to the sender agent and configures the multicast path so that the starting of attempt sending is not blocked by the multicast path configuration.

3.3 Implementation

**Republic agent**: Republic agent contains (1) the reliable data multicast protocol (Section 3.2.1), (2) the Republic API (Table 3.1) and (3) the agent side of the agent-manager interface (Table 3.2). These modules run in multiple threads to leverage the parallelism in multicore CPUs so that the modules don’t block each other. These
threads communicate with each other via Unix system pipe, which is an efficient and light weighted inter-thread communication mechanism. The Republic agent program is implemented in 6K lines of C, which is efficient in execution. The implementation leverages lock-free data structures, such as hash map, priority queue and list from the Apache Portable Runtime (APR) [24] library for efficient data structures and operations.

In the protocol, the control channel and the data channel are in different threads. The control channel is based on TCP connections between the sender and the receivers. The data channel adopts netmap kernel-stack-bypass framework [23] to send/receive UDP packets efficiently. Netmap is adopted because it is supported by multiple operating systems and compatible with NICs from many vendors. With netmap, a single CPU core can transmit packets at 10 Gbps with low CPU overhead. Multicast data is stored in-memory temporary file system (tmpfs) for fast access.

The Republic API is based on Unix domain socket so that the application processes can talk to the local agent to request multicast service efficiently. Accessing the data through the Unix domain socket is too expensive for bulk transfer, so the data is written to and read from the in-memory file directly by the agents or the application processes. The communication in the agent-manager interface is based on Apache Thrift [25]. Thrift is adopted because it is an efficient and scalable cross-language RPC framework.

**Republic manager**: Republic manager consists of (1) the multicast request management, (2) the multicast path resource management and configuration and (3) the manager side of the agent-manager interface. This requires the Republic manager to talk to many modules including the agents, the scheduling algorithm, the packet switch controller and the circuit switch controller. The Republic manager is
implemented in 2K lines of Java. Java is adopted because it provides rich and efficient data structures and various libraries for inter-module communication. The multicast path configuration module talks to the circuit switch controller and the packet switch controller via RESTful API, which is widely adopted by many OpenFlow controller platforms.

Republic agent [26] and manager [27] is open-source.

3.4 Deployment

An OCS-based hybrid data center testbed in Figure 1.1(a) is built and it is shown in Figure 3.4. The testbed is built with 40 servers, six 48-port 10 GbE OpenFlow [28] switches, one 192×192 OCS, sixteen 1×4 optical splitters and a Republic manager server. Five of the OpenFlow switches are used as ToR switches. Each ToR switch is partitioned into four logical ToR switches. Each logical ToR switch is attached to two servers and connects to the core OpenFlow packet switch. The physical layer portion of a multicast path is established with the optical splitters attached to OCS ports. The reconfiguration time of the OCS is around 70 ms. The physical multicast portion with fanout larger than 4 is achieved by cascading multiple 1×4 splitters. Each of the 40 servers and the Republic manager server has a 6-core Intel Xeon CPU E5-1650 v3 @ 3.50GHz, 128 GB of DDR4 RAM @ 2133 MHz and one 10 GbE NIC. All servers, switches and the manager are connected via an additional 1 GbE management network. The Republic manager configures the ToR switches through a Ryu OpenFlow controller [29] and configures the OCS through a controller talking to the OCS using TL1/telnet commands.
Figure 3.4: The deployment of Republic in hybrid data center testbed based on optical circuit switching (OCS) (architecture in Figure 1.1(a))
3.5 Evaluation

Republic is evaluated in an OCS-based hybrid data center testbed (Section 3.4). The evaluation adopts a variety of realistic applications workload (Section 3.5.1). This section shows how and how much Republic reduce end-to-end multicast data transfer time as well as application running time (Section 3.5.2); justifies the design decisions made in Republic agent (Section 3.5.3); finally shows the performance of Republic manager (Section 3.5.4).

3.5.1 Evaluation Workload

The evaluation uses two popular iterative machine learning algorithms and a widely adopted benchmark for database system. These applications run on top of Apache Spark [30] (Section 3.5.2 explained why the evaluation uses Apache Spark). Details of the applications are as follows.

**Neural word embedding:** This is a machine learning model that takes a text corpus as input and trains the vector representations of words in the corpus. Such word embedding operations are critical techniques commonly used in deep learning and natural language processing. The evaluation uses the Word2Vec [4] implementation in Spark MLlib. The input corpus setting has the same properties as the Wikipedia corpus used in [31]. In this Word2Vec implementation, the multicast data is the training model, which is about 504 MB.

**Latent Dirichlet allocation (LDA):** This is a topic clustering machine learning model widely used in natural language processing. The algorithm assigns the input documents to a topic by training a model that represents the probability of a word appearing in a topic. The Spark LDA implementation is in [32]. The input corpus is the synthetic 20 Newsgroups dataset [33] having one million documents. In this
implementation, the multicast data is the training vocabulary model, which is about 735 MB.

Database management system (DBMS) queries: TPC-H [34] is a widely adopted benchmark for database system. The benchmark contains 22 business oriented database queries designed to have broad industry-wide relevance. TPC-H queries are run on Spark SQL framework [3]. The overall size of the database tables is 16 GB. The multicast data is one of the input tables in the join operation. In a complete benchmark run, there are 58 multicast data whose sizes range from 4.0MB to 6.2GB and 48.3GB in total.

3.5.2 End-to-end Application Level Improvement with Republic

Comparison Methodology

The evaluation uses Apache Spark [30] (v1.6.1) as an example among the distributed data processing frameworks to evaluate Republic. The first reason for choosing Spark is that Spark is a general-purpose, efficient and popular cluster computing framework. A variety of applications including machine learning algorithms, database queries, stream processing, etc., have been implemented in Spark. The second reason is that Spark provides multiple dedicated mechanisms to deliver multicast data (called “broadcast object”).

Spark can easily use the data multicast service provided by Republic. Only the broadcast module in Spark is replaced with a module that uses the Republic API. This module sends the “broadcast object” to the executors once the object is created. This change is completely transparent to Spark user programs. Other data center applications can adopt Republic in a similar way.

In Section 3.5.2, Republic is compared against the state-of-the-art multicast mech-
anisms adopted in Apache Spark, i.e. Torrent (Cornet in [1]) and HTTP, and show that Republic yields a large improvement. The benefits can also apply to other data center applications and distributed computation systems. In Torrent multicast, after the broadcast object is created at the master, the object is partitioned into multiple blocks of 4 MB. For each of the blocks, the receiver randomly chooses the source of the block from the master and other executors having a copy of the block. After all blocks of the object are received, they are reassembled into the original object. Executors fetch the broadcast object on-demand, i.e., data transfer starts when the task in the executor starts using the data. Since Spark runs in Java virtual machine (JVM), there are two layers of serialization/deserialization since the blocks are also objects that need to be serialized/deserialized. In HTTP multicast, the master starts an HTTP server and writes the serialized object to disk. An executor fetches the serialized object via an HTTP GET request when the task using the object is assigned to the executor. The fetch always happens before the task starts using the object.

The experiment runs in a Yarn cluster with 25 worker servers in the testbed. Each worker provides 4 cores/88 GB memory to Yarn. Spark applications are submitted to the Yarn resource manager server. An application request executors from Yarn. For Spark applications, an executor is an independent JVM having dedicated cores and memory. Each of the applications randomly picks one executor of 4 cores/88 GB memory for the application master and $N=10$ or 22 executors of 2 cores/44 GB memory for the application slaves. There are up to four or two applications running concurrently in the cluster when $N=10$ or 22 respectively. Each application is submitted to Yarn 8 times under each configuration. The two machine learning applications run 10 iterations. When using Torrent and HTTP for multicast, the packet-switched core bandwidth between racks are 20Gbps; When using Republic,
Figure 3.5: CDFs of broadcast object reading time in second

the packet-switched core bandwidth is 10Gbps and the circuit switch bandwidth is 10Gbps. So the inter-rack bandwidth in both cases are the same, however, the circuit switch only serves multicast traffic. Attempt sending interval is set to $i_a$ is 2ms in Republic.

Reduced Broadcast Object Reading Time

Figure 3.5 shows the CDF of the broadcast object reading time on all the executors. The broadcast object reading time is defined as the duration from (1) the time when the object reading request is issued by the program running in the executor to (2) the time when the program finishes the deserialization of the object (transferring an
object between JVMs requires the object to be serialized at the sender and deserialized at the receiver), and it includes the circuit reconfiguration time. The broadcast object reading time indicates the application level waiting time when retrieving the broadcast object, which has greater practical meaning than the pure network transfer time.

Republic shows great improvement in the broadcast object reading time because Republic leverages the physical-layer multicast capability in the hybrid rack-level interconnections and has an efficient data multicast protocol. In Republic, the network transfer is much faster than the deserialization, so reading time is dominated by the deserialization time. In Torrent and HTTP, the network transfer is slow, the deserialization needs to wait for incoming bytes from the network. So the reading is dominated by the network transfer.

For the application using 10 executors, in TPC-H queries, comparing with HTTP, Republic achieves 3.72× and 2.85× improvement at the 60th and the 90th percentile, respectively (Figure 3.5(a)); in the machine learning applications, Republic improves the reading time by 2.9× at the 40th and 80th percentile comparing with Torrent.
(Figure 3.5(b)). For the application using 22 executors, the reading time in Republic remains unchanged, and it shows more improvement. This is because in Torrent and HTTP, the amount of traffic sent to the network is proportional to the number of executors, while in Republic data is sent only once. In addition to that, Torrent has much protocol overhead when checking the existence of the blocks on machines as well as the overhead of two-level serialization. For example, comparing with Torrent in the machine learning applications, Republic achieves $4.0 \times$ and $3.6 \times$ improvement at the 40th and the 80th percentile, respectively (Figure 3.5(d)); at 100th percentile in TPC-H, the improvement is $10.7 \times$ (Figure 3.5(c)).

The application running time is also improved due to the high-performance data multicast of Republic (Figure 3.6). For example, the running time of LDA is improved by 32.1% comparing with Torrent when using 22 executors.

### 3.5.3 Efficient Data Multicast with Tuned Attempt Sending

The attempt sending interval ($i_a$) has a great impact on the data multicast performance. So $i_a$ needs to be carefully chosen for a specific hybrid data center (Section 3.2.1). To quantitatively understand the effect of $i_a$, the cases where $i_a$ varies between 70 $\mu$s to 70 ms are examined. These $i_a$s translates to 10% and 0.1% of the 10 Gbps server bandwidth. Figure 3.7 shows the performance metrics in the case having 22 executors (at least 11 and at most 22 receiver servers) with different $i_a$.

For the redundant packets sent to the data channel (Figure 3.7(a)), on the experiment testbed, when $i_a$ is larger than 700 $\mu$s, the redundant bytes is less than 1 MB at the 99th percentile. This is because the redundant packets are the packets sent before the last receiver starts receiving packets from the data channel. The larger the $i_a$ is, the fewer packets sent are redundant. For the lost packets (Figure 3.7(b)),
Figure 3.7: Efficient data multicast with attempt sending
when $i_a$ increases from 70 $\mu$s to 7 ms, the number of lost packets reduces from 10.2K to 8 per data receiving at the 99th percentile and the number of cases having packet losses reduces from 29.4% to 1.4%. This is because, with a large $i_a$, fewer packets are sent during the transient circuit state.

However, an excessively large $i_a$ unnecessarily extends the attempt sending time, which delays the start of the best effort high-rate sending and eventually increases the total sending time. On the experiment testbed, when $i_a$ is between between 700 $\mu$s and 2 ms (1% and 0.3% of the server bandwidth), the attempt sending time is very close to the attempt sending time that achieved when $i_a$ is 70 $\mu$s (Figure 3.7(c)). The cases having attempt sending time less than 20 ms reuse the existing circuits (Figure 3.7(c)). In these cases, the circuit reconfiguration delay (about 50-60 ms in the experiment testbed) is eliminated. The delay is caused by rule installation to the packet switches. The metric total extra time is adopted to show the slow down in transferring a multicast data. The total extra time is defined as the duration from the time when the sender receives approve to the time when the sender collects DATA_RCVDs from all the receivers minus ideal line-rate data transfer time. The total extra time spent on sending multicast data reaches the minimum under the same $i_a$ range (Figure 3.7(d)). However, when $i_a$ is 70 $\mu$s, the extra time is significantly increased. This is because (1) the retransmissions for a large number of lost packets compete for the server bandwidth with the data packets sent to the data channel and (2) the retransmission packets are delivered after the data channel transfer finishes. Therefore, the evaluation suggests that it is reasonable to set $i_a$ between 1% to 3% of the average circuit reconfiguration time of the deployed circuit switch.

As the number of receivers increases, the attempt sending time will increase slightly as the high-rate sending starts after the sender collects INTL_SEQ messages
from all the receivers. More receivers could only slightly increase the time when the
last INTL SEQ is processed. However, a longer attempt sending time may also reduce
the chance of having packet loss during the transient state of the multicast path. So
the total time it takes to finish the entire data transfer will not change a lot as the
number of receiver increases.

Figure 3.7 also compares with an alternative approach which starts the high-
rate sending after the sender receives the first (instead of the last) INTL SEQ from a
receiver connected via the circuit in the multicast path. In the alternative approach,
the high-rate sending starts early (Figure 3.7(c)) due to the large variance on the
circuit reconfiguration time. However, starting high-rate sending early cannot reduce
the extra sending time (Figure 3.7(d)). This is because at a early time, more circuits
are under an unstable transient state so that more packets are prone to lose, which
results in more packet retransmissions. The alternative approach also sends more
redundancy packets under the same \( i_a \), since some of the redundant packets are
actually sent at high-rate.

3.5.4 High Throughput of Republic Manager

The metric manager response time is used to show the achievable throughput of
the Republic manager. Manager response time is the duration from the time when
the sender agent calls request to the time when it receives approve. The average
response time is 1.36 ms. This means that the manager can achieve a throughput of
735 requests per second when the average number of receivers in the request is 16,
which is the value in the experiment. The experiments have about 0.1 request per
second, which implies that the Republic manager can handle $7.35K \times$ more concurrent
applications that are similar to the applications in the experiments.
3.6 Related Work

**Multicast-featured hybrid data centers:** Republic is motivated by previous works building multicast-featured circuit-switched rack-level interconnections. Wang *et al.* proposes c-Through [6], a hybrid data center architecture that leverages 3D MEMS-based OCS for fiber optics. Wang *et al.* [11], Samadi *et al.* [12] and Xia *et al.* [2] extend the 3D MEMS-based OCS with passive optical splitters to enable physical-layer multicast. FireFly [8] introduces transparency switchable mirror and galvo (rotating) mirror to direct the free-space optical (FSO) signals between the racks. FlyCast [13] augments FireFly with partial reflection mirror to enable the physical-layer multicast with FSO. Zhou *et al.* [10] proposes a rack-level interconnection solution based on wireless 3D beamforming at 60 GHz. Technologies such as multi-user 3D beamforming [14, 15, 16] have potential in supporting point-to-multipoint wireless data transfer. However, these prototypes are still far from a complete system that applications can leverage and each of them is specific to a particular architecture. Instead, Republic goes significantly further by building the first full-fledged cross-architecture system that provides a universal data multicast service in hybrid data centers featured with physical multicast capability.

**Multicast in data center applications:** Data multicast is very common in data center applications, especially in big-data processing frameworks. Some of the popular frameworks and their multicast mechanisms are enumerated here. Spark [30] is a general large-scale data processing framework. It provides “broadcast object” as a data type, which allows the worker nodes to retrieve the data through the built-in data multicast mechanisms including Torrent and HTTP (discussed in Section 3.5.2). In HDFS [35], a data block is replicated to multiple storage nodes. The data block is propagated along a chain from the source node to the storage nodes. Tensorflow [36] is
a distributed computation framework for machine learning and deep neural networks. The workers fetch the machine learning model from a group of tasks via unicast transfer. Tez [37] is a data processing framework for a complex directed-acyclic-graph (DAG) of tasks. The mechanism used for broadcasting data to the tasks can be customized by the application developer. These frameworks can adopt Republic’s data multicast service as it is shown in the experience with Spark in the evaluation (Section 3.5.2).

3.7 Chapter Summary

This chapter presents Republic, the first fully-fledged solution to data multicast in hybrid data centers. Republic allows the data center applications to exploit new physical multicast capabilities in hybrid data centers and at the same time achieves reliable and high-performance data multicast. Republic is structured as a service, with a simple unified API, making it easily accessible to expert and non-experts alike. Republic is deployed in an OCS-based hybrid data center testbed. The evaluation shows as much as 4.0× improvement for data multicast. Republic is open-source so as to provide others with an experimental platform for conducting future research on topics such as multicast scheduling algorithms [22] and new inter-rack network architectures [38].
Chapter 4

Data Multicast Scheduling in a High-Bandwidth Circuit Switch

This chapter presents Monarchy, a high-performance data multicast scheduling algorithm for the hybrid rack-level interconnections having high-bandwidth circuit switch. This chapter is organized as follows. Section 4.1 discusses the challenges to be solved by the scheduling algorithm. Section 4.2 formulates the scheduling problem. Section 4.3 discusses the related work. Section 4.4 presents the Monarchy scheduling algorithm. Section 4.5 evaluates Monarchy. Section 4.6 summarizes the chapter.

4.1 Challenges

The design of Monarchy ought to address the following two challenges.

First, in hybrid data centers, the bandwidth of a circuit switch port could be much larger than the bandwidth of a server NIC. This is driven by the following reasons. First, the inter-rack traffic is hungry for bandwidth because it is the aggregation of the traffic from/to tens of servers within the rack. This requires the circuit switch to provide a considerably larger bandwidth that can match the inter-rack traffic demand. Second, connecting a rack and the circuit switch with a high-bandwidth port is much more preferable than using a large number of low-bandwidth ports. Otherwise, it exacerbates the scalability problem of the circuit switch especially for large data centers having hundreds of racks each of which has tens of servers. This is because the number of circuit switch ports is proportional to the number of ToR ports connecting
to it. Third, circuit switch is adopted for its capability in carrying high-bandwidth signal, so restricting the circuit switch port bandwidth contradicts the essential merit of using it. Thus, in a realistic hybrid data center, the traffic from a single server cannot fully take up the bandwidth of the circuit switch port. This means that in order to achieve high utilization of the circuit switch bandwidth, the scheduling algorithm must wisely share the bandwidth among the multicast traffic from multiple servers.

Second, improving the performance of applications should be the ultimate objective of the scheduling algorithm. Thus, to deliver the most benefit to applications creating network flows, the scheduling algorithms ought to be “application-aware”. That is to say, the scheduling algorithm should consider the traffic demands from individual applications and optimize the time it takes to finish the traffic of each application [39, 18] rather than optimize for the aggregated demands no matter which application a demand belongs to [19, 40]. As previous works [18, 39] suggest, “average demand completion time” is the right metric to evaluate the effectiveness of multicast traffic scheduling on applications’ performance. However, how to design a scheduling algorithm that optimizes the average completion time of multicast data demands is an open question. In addition to that, this algorithm should fully exploit the high-bandwidth circuit switch port even though it has much larger bandwidth than the NIC port of a single server.

### 4.2 Problem and Approaches

This section formulates the scheduling problem and introduces the approaches used in the scheduling algorithm.
<table>
<thead>
<tr>
<th>Notation</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$g \in \mathbb{Z}^+$</td>
<td>total number of demands</td>
</tr>
<tr>
<td>$n \in \mathbb{Z}^+$</td>
<td>total number of racks</td>
</tr>
<tr>
<td>$b_i &gt; 0$</td>
<td>bandwidth of the server NIC port</td>
</tr>
<tr>
<td>$b_h \geq b_i$</td>
<td>bandwidth of the circuit switch port</td>
</tr>
<tr>
<td>$f \in \mathbb{Z}^+$</td>
<td>upper limit on the port-to-multi-port circuit (P2MPC) fanout</td>
</tr>
<tr>
<td>$\delta \geq 0$</td>
<td>circuit switch reconfiguration delay</td>
</tr>
<tr>
<td>$demand_k$</td>
<td>a multicast demand composed of a tuple of $s_k$, $r_k$ and $d_k$</td>
</tr>
<tr>
<td>$s_k \in {1, ..., n}$</td>
<td>sender rack index of Demand $k$</td>
</tr>
<tr>
<td>$r_{kj} \in {0, 1}$</td>
<td>indicate if Rack $j$ is a receiver of Demand $k$</td>
</tr>
<tr>
<td>$d_k \in \mathbb{Z}^+$</td>
<td>data size of Demand $k$</td>
</tr>
<tr>
<td>$C_{ij}^t \in {0, 1}$</td>
<td>indicate if a port-to-multi-port circuit (P2MPC) is set up from Rack $i$ to Rack $j$ at Epoch $t$</td>
</tr>
<tr>
<td>$R_{ki}^t \in {0, 1}$</td>
<td>indicate if Demand $k$ uses Rack $i$ as a relay at Epoch $t$</td>
</tr>
<tr>
<td>$D^t &gt; 0$</td>
<td>duration of Epoch $t$</td>
</tr>
<tr>
<td>$T_k \in \mathbb{Z}^+$</td>
<td>epoch index at which Demand $k$ finishes</td>
</tr>
</tbody>
</table>

Table 4.1: Notations in the chapter. Lowercase and uppercase represent known and unknown variables respectively.

### 4.2.1 Network Model

Figure 2.3 shows a data center with $n$ ToRs (notations are in Table 4.1). These $n$ ToRs connect to a circuit switch with bandwidth $b_h$ at each port (e.g. 40 GbE or 100 GbE) and a packet-switched network. The circuit switch is able to build
directed port-to-multi-port circuits (P2MPCs) between an input port and multiple output ports. The P2MPC divides the physical layer signal to multiple beams with reduced power so the maximum fanout is limited to $f$ whose value depends on the transmission power and the sensitivity of the transceiver. The connections in the circuit switch can be dynamically reconfigured with an overhead of reconfiguration delay $\delta$. The model assumes that each ToR provides an exclusive port connecting to the circuit switch for multicast data transfer. Each of the ToRs is connected to multiple servers via links each with bandwidth $b_l$ (e.g., 10 GbE). The model does not require that the ToRs have virtual output queues (VOQs) [41] buffering the outgoing packets. Instead, the scheduling algorithm is based on a more practical and general case where the multicast data is transferred server-to-server.

### 4.2.2 Data Multicast Demands

The scheduling algorithm takes a set of $g$ data multicast demands as input. Each of the demands is identified by its index $k$ and it includes the sender rack index $s_k$, the receiver racks represented by an indicator vector $r_k \in \{0, 1\}^{1 \times n}$, and the size of the multicast data $d_k$. In $r_k$, the $j$th element $r_{kj}$ represents whether rack $j$ is the receiver of demand $k$.

### 4.2.3 Scheduling Objective

To speed up the performance of applications, the objective of the scheduling algorithm is to minimize the average demand completion time (average DCT) for the input demands. The DCT of demand $k$, $DCT_k$, is defined as the time duration from 1) the time when the demand request is received by the scheduler (demand arrival time, denoted as $t_{arr}$) to 2) the time when the data is received by all the receivers (demand
delivery time, denoted as $t_{del}^k$, i.e., $DCT_k = t_{del}^k - t_{arr}^k$. Minimizing average DCT is equivalent to minimizing the sum of all the DCTs, i.e., $\sum_{k=1}^g DCT_k$.

4.2.4 The Scheduling Problem and Approaches

Monarchy adopts two approaches to schedule the multicast data transfer, i.e. multi-hopping and segmented transfer.

**Multi-hopping:** With multi-hopping, a multicast flow can reach a receiver ToR via other ToRs (called relay ToR) as intermediate hops. This brings the following benefits. First, it enriches the reachability of a ToR because each additional hop allows the source ToR to reach the ToRs connected via the P2MPCs rooted from the relay ToRs. In Figure 4.1’s example, at Epoch 1 (epoch is defined later), Rack 5 (R5) is able to reach R1 & R4 via the P2MPC rooted from R5 as the first hop and the P2MPC rooted from R3 as the second hop. Thus, in transferring Demand 2 (D2), R3 acts as a relay ToR to enrich the reachability of R5 so as to cover the destination ToRs of D2 (red dotted line). Second, it increases the utilization of the circuit switch bandwidth. With multi-hopping, the bandwidth of the circuit switch port on a ToR can be shared among the multicast flows from the servers in the rack and the multicast flows using the ToR as a relay. For example, in Figure 4.1(b) the P2MPC rooted from R3 is shared between D1 & D2. Third, it reduces the occurrence of circuit reconfigurations because P2MPCs can be shared by more demands.

On a receiver ToR of a flow (e.g. R1 of D1 Figure 4.1(b)), the ToR is installed with the rules forwarding the flow to the destination servers in that rack. On a relay ToR of a flow (e.g. R3 of D2 Figure 4.1(b)), the ToR is installed with the rules forwarding the flow back to circuit switch port. For the flow going to a ToR which is neither the relay nor the receiver ToR (e.g. R1 of D2 Figure 4.1(b)), the ToR simply discards the
Figure 4.1: An example of multicast demand scheduling. The example involves ten racks with $b_h=20$ Gbps, $b_l=10$ Gbps (a P2MPC can be shared among at most two multicast flows simultaneously) and a circuit switch with $\delta=1$ ms. (a) The input demands. (b) The optimal scheduling with multi-hopping and segmented transfer. The established P2MPCs are shown on all hops. The multicast flow is only marked on the hop it uses. (c) A comparison between segmented and non-segmented transfer (average DCT 412/6 ms vs 508/6 ms).
packets of the multicast flow by installing no rule for the flow, so that the packets are
discarded by the hardware of the ToR. Thus, discarding the packets does not involve
any extra overhead on the switch so it won’t slow down the forwarding of any other
packets.

**Segmented transfer:** With segmented transfer, the scheduling may allocate mul-
tiple transfer sessions for a single multicast demand rather than require the demand
to run to completion in a single session. This is directly beneficial to the minimization
of average DCT because small demands are not blocked by large demands. Thus, the
scheduler creates multiple *epochs* in scheduling the given demand set (Section 4.2.2).
Each epoch includes (1) a fixed circuit configuration \( (C_t \in \{0, 1\}_{m \times n}) \), (2) the
demands served in the epoch and the set of relay racks of each of the served demand
\( (R_t \in \{0, 1\}_{g \times n}) \), and (3) the epoch duration \( (D_t) \). In Figure 4.1’s example, D1 & D2
are transferred in Epoch 1. However, D5 & D6 cannot be served in Epoch 1 due to
the contention at R4 with D1 & D2. In this situation, D3 & D4 are served simultane-
ously with D1 & D2 for better circuit switch bandwidth utilization. With segmented
transfer, a new epoch can be created at 21 ms (lower part in Figure 4.1(c)) to serve
D5 & D6. Otherwise, these two demands have to wait until the completion of D3 &
D4 (101 ms in Figure 4.1(c) upper part), which results in larger average DCT.

With these two approaches, the output of the scheduling algorithm is defined as a
series of epochs \((C^t, R^t \text{ and } D^t)\). Denote \( T_k \) as the index of the epoch where demand
\( k \) finishes. Then, the total number of epochs is \( \max_k(T_k) \). The problem formulation
assumes that the sender transfers multicast flows at the full rate \( b_t \) since this helps
minimize the flow completion time. To handle the case having multiple flows going
to a receiver server, the data can be sent to another server within the same rack and
transferred from that server to the receiver. This can be done efficiently since it only
involves intra-rack traffic. The constraints to the solution are shown in Equ. 4.1.

\[ \text{subject to: } \forall t, j, \sum_{i=1}^{n} C_{ij}^t \leq 1, \quad \text{one circuit port per ToR} \]
\[ \forall t, i, \sum_{j=1}^{n} C_{ij}^t \leq f, C_{ii}^t = 0 \quad \text{P2MPC fanout limit} \]
\[ \forall t, j, k, \sum_{i=1}^{n} R_{ki}^t C_{ij}^t \geq r_{kj} R_{ks}^t, \quad \text{cover receivers} \] (4.1)
\[ \forall t, j \neq s_k, k, \sum_{i=1}^{n} R_{ki}^t C_{ij}^t \geq R_{kj}^t R_{ks}^t, \quad \text{cover relays} \]
\[ \forall t, i, \sum_{k=1}^{g} R_{ki}^t \leq \frac{b_h}{b_t}, \quad \text{bandwidth limit} \]
\[ \forall k, \sum_{t=1}^{T_k} R_{ks}^t D^t \geq \frac{d_k}{b_t}, \quad \text{data transfer completion} \]

The goal of the scheduling can be written as Equ. 4.2.

\[ \text{goal: minimize } \sum_{k=1}^{g} DCT_k \]
\[ = \sum_{k=1}^{g} ((\sum_{t=1}^{T_k} D^t) - ((\sum_{t=1}^{T_k} R_{ks}^t D^t) - \frac{d_k}{b_t}) + T_k \delta) \] (4.2)

NP-hardness of the problem: The scheduling problem can be proved to be NP-hard using contradiction. In a special case of the scheduling problem, each of the multicast demands has only one receiver rack, the data sizes are the same, \( b_h \) equals to \( b_t \), and \( \delta \) is zero. In such special case, the problem is equivalent to the problem of scheduling the unicast flows in a packet switch where a flow exclusive takes the entire bandwidth of the input and output ports when it is transferred. Assume that the data multicast scheduling problem can be solved with a polynomial algorithm. Then, such algorithm can also solve the problems in the special case in polynomial time. However, the special case is a \textit{sum coloring problem} [42] which is NP-hard. This contradicts the assumption. Thus, the original problem is also NP-hard.
4.3 Related Work

**Scheduling algorithm:** The problem of traffic scheduling in hybrid data centers is being actively investigated in recent years. Previous works propose scheduling algorithms for different types of traffic with various optimization goals. These works either deal with unicast traffic patterns or do not optimize for individual demands. Solstice [19] minimizes the makespan of a batch of unicast traffic demands (a demand matrix) in a hybrid data center. It decides the amount of traffic to be transferred through the circuit switch and the packet-switched network and creates a schedule for the circuit switch traffic. Eclipse [40] maximizes the circuit switch utilization within a given time duration for a unicast demand matrix. However, Solstice [19] and Eclipse [40] do not optimize for the demand from individual applications, instead, they achieve optimization goal for the aggregated traffic demands. To bring impactful performance improvement to applications, Sunflow [18] minimizes the average coflow completion time as the improved scheduler beyond Solstice and Eclipse (coflow is defined as a set of unicast demands comes from an application [39]). However, Sunflow is designed for unicast flows as well. For multicast scheduling, Blast [2] picks the multicast demands that should be served by the circuit switch and leaves the rest demands to the packet-switched network in order to maximize the number of multicast demands (or the bytes of multicast demands) being served by the circuit switch. However, Blast does not optimize for the demands of individual applications and does not consider exploiting the high-bandwidth circuit switch.

**High-bandwidth circuit switch:** Composite-path switching [43] in a hybrid data center leverages the high-bandwidth in the circuit switch. A Composite-path is a high-bandwidth link connecting the packet-switched network and the circuit switch. The composite path can be simultaneously shared by multiple unicast flows going to
or coming from the same ToR. Monarchy differs from [43] in that the high-bandwidth link is shared by multicast flows.

### 4.4 Scheduling

**Algorithm 1** Skeleton of the scheduling algorithm

```plaintext
1: procedure SCHEDULE\(\text{demand}_{1,...,g}, \delta\)
2: \(m^{set} \gets \text{Set}(\text{demand}_{1,...,g})\) \hspace{1cm} \triangleright \text{convert to a set of demands}
3: \(t \gets 0\) \hspace{1cm} \triangleright \text{initialize epoch index}
4: \(E^{list} \gets \text{List()}\) \hspace{1cm} \triangleright \text{initialize the returned list of epochs}
5: while not \(m^{set}.\text{isEmpty()}\) do
6: \(C^t_{[n \times n]}, R^t_{[g \times n]} \gets \text{CreateEpochSchedule}(m^{set})\) \hspace{1cm} \triangleright \text{Section 4.4.1 & Alg. 2}
7: \(D^t \gets \text{DecideEpochDuration}(R^t_{[g \times n]}, \delta)\) \hspace{1cm} \triangleright \text{Section 4.4.2}
8: \(E^t \gets \text{Tuple}(C^t_{[n \times n]}, R^t_{[g \times n]}, D^t)\) \hspace{1cm} \triangleright \text{get an epoch}
9: \(m^{set} \gets \text{Serve}(m^{set}, E^t)\) \hspace{1cm} \triangleright \text{update the remaining demands}
10: \(E^{list}.\text{append}(E^t)\) \hspace{1cm} \triangleright \text{add the epoch to the schedule}
11: \(t \gets t + 1\) \hspace{1cm} \triangleright \text{increase epoch index}
12: return \(E^{list}\)
```

Monarchy is a heuristic scheduling algorithm whose skeleton is shown in Alg. 1. The algorithm works in an iterative manner until all the demands have been completely scheduled (Line 5 (L5)). Each iteration creates an epoch. Creating the circuit configuration \(C^t\) and picking the demands to be served \(R^t\) are closely related questions because the demands are served by the circuit. So creating a circuit configuration should consider the senders and the receivers of the demands to be served. Determining the epoch duration \(D^t\) is relatively independent from determining \(C^t\)
and $R^t$. However, given an $R^t$, as will be shown in Section 4.4.2, $D^t$ has a great impact on the effective utilization rate of the circuit switch. Thus, the algorithm first determines $C^t$ and $R^t$ (L6 & Section 4.4.1) and then determines $D^t$ according to $R^t$ (L7 & Section 4.4.2). After the epoch is created (L8), the remaining bytes of the demands are updated so that only the incomplete demands are given to the next iteration (L9). Time complexity of the algorithm is $O(g^2n(g + n))$.

### 4.4.1 Create the Circuit Configuration and Choose the Demands to be Scheduled in an Epoch

Alg. 2 shows the function that creates the circuit configuration and picks the demands to be served in an epoch. The algorithm iteratively considers the demands in the increasing order of the remaining data bytes (L2); in each iteration, the algorithm may assign the circuit resources to a demand (L7,10). By doing this, the demands with smaller remaining data bytes have higher chances in getting the circuit resources, which is beneficial in minimizing average DCT.

Previous work [2] models the circuit configuration as a hypergraph where the vertices are the racks in the data center and the P2MPCs are directed hyperedges. The hyperedge originates from a single vertex in the hypergraph and points to multiple vertices. In the data multicast scheduling problem, each hyperedge has a capacity limit of $b_h/b_t$, which limits the number of multicast demands simultaneously transferred through the P2MPC. In order to serve a multicast demand, the sender vertex must connect to the receiver vertices via hyperedges having free capacities. A demand can be served by multiple cascaded hyperedges since multi-hopping is adopted. Thus, the problem to be addressed in each iteration is to find a set of P2MPCs (hyperedges) that satisfy the connectivity required by the demands and have free capacity. New
Algorithm 2 Create the circuit configuration and choose the demands to be scheduled in an epoch

1: procedure CREATEEPOCHSCHEDULE($m^\text{set}$)
2: $m^\text{list}$←sort($m^\text{set}$, key=$\lambda m_k.m_k$:\text{remain}())
3: $G$←HyperGraph() \Comment{initialize a hyper-graph}
4: $M$←List() \Comment{initialize the list of demands to be served}
5: for $d$ in $m^\text{list}$ do
6: \quad if $p^\text{set}$\text{←}SolveConflict($G$, $d$, True) then \Comment{Line 16}
7: \quad \quad DecideToServeDemand($d$, $M$, $p^\text{set}$) \Comment{Line 13}
8: for $d$ in $m^\text{list}$ do
9: \quad if $d$ not in $M$ and $p^\text{set}$\text{←}SolveConflict($G$, $d$, False) then \Comment{Line 16}
10: \quad \quad DecideToServeDemand($d$, $M$, $p^\text{set}$) \Comment{Line 13}
11: $C, R$←ConvertRepresentation($G$, $M$)
12: return $C, R$
13: procedure DECIDETOSERVEDEMAND($d$, $M$, $p^\text{set}$)
14: $p$.addDemand($d$) for $p$ in $p^\text{set}$
15: $M$.append($d$); $d$.addP2MPC($p^\text{set}$)
16: procedure SOLVECONFLICT($G$, $d$, $\text{loopfree}$)
17: $cr^\text{list}$←List([r for r in $d$.rcvrs() if $G$.vertex(r).indegree==1])
18: $ncr^\text{list}$←$d$.rcvrs()−$cr^\text{list}$
19: $p_{\text{sndr}}$←$G$.vertex($d$.sndr()).P2MPC()
20: $rts^\text{set}$←Set()
21: if $p_{\text{sndr}}$ is not None then \Comment{Case 1}
22: \quad for $c$ in $cr^\text{list}$ do
23: \quad \quad if $G$.isConnected($d$.sndr(), $c$) then
24: \quad \quad \quad if not $G$.freeCapacity($d$.sndr(), $c$) then \Comment{demand skipped}
25: \quad \quad \quad return None
26: \quad \quad else \Comment{demand skipped}
27: \quad \quad \quad $\text{root}$←$G$.rootAncestor($c$)
28: \quad \quad \quad if not ($\text{root}$ and $G$.freeCapacity($\text{root}$, $c$)) then
29: \quad \quad \quad \quad return None \Comment{demand skipped}
30: \quad \quad \quad $rts^\text{set}$.add($\text{root}$)
31: $p_{\text{new}}^\text{set}$, $p_{\text{etnd}}$←extendP2MPC($p_{\text{sndr}}$, $rts^\text{set}$, $ncr^\text{list}$, $G$)
32: else \Comment{Case 2}
33: \quad for $c$ in $cr^\text{list}$ do
34: \quad \quad $\text{root}$←$G$.rootAncestor($c$)
35: \quad \quad if not ($\text{root}$ and $G$.freeCapacity($\text{root}$, $c$)) then
36: \quad \quad \quad return None \Comment{demand skipped}
37: \quad \quad $rts^\text{set}$.add($\text{root}$)
38: $p_{\text{new}}^\text{set}$←$G$.addP2MPC($p_{\text{sndr}}$, $rts^\text{set}$, $ncr^\text{list}$, $G$)
39: if $p_{\text{new}}$ is None then return None \Comment{demand skipped}
40: if $\text{loopfree and}$ adding $p_{\text{new}}^\text{set}$ (and extending $p_{\text{etnd}}$) make $G$ cyclic then
41: \quad return None \Comment{demand skipped}
42: $G$.addP2MPC($p_{\text{new}}^\text{set}$, $p_{\text{etnd}}$) \Comment{add new hyperedges to the graph}
43: return $G$.getP2MPCs($d$.sndr(), $d$.rcvrs()) \Comment{return the set of P2MPCs the demand uses}
Figure 4.2: Example of solving conflicting racks. (a) Considering Demand 8 (D8) and D9 during an execution of Alg. 2. The algorithm executes to Iteration $i$ where the P2MPCs rooted from Rack 3 (R3), R4, R7, R8, and R10 are added to the network. In Iteration $i+1$, D8 is added to the schedule by extending the outputs of the P2MPC rooted from R7 to R9 and R10; in Iteration $i+2$, D9 is served by adding a new P2MPC rooted from R1 and connecting to R2, R3, and R4. (b) An example of the loop in the hypergraph. All the racks (nodes) can’t be reached by any racks (nodes) out of the loop by neither extending existing P2MPCs nor adding new P2MPCs.

P2MPCs (hyperedges) may be created if necessary.

In the function, hyperedges are added to hypergraph $G$ in two stages. In the first stage (L5-7), when a new hyperedge is added to the graph, loop is not allowed. That is to say, a demand is skipped by the first stage if serving the demand results in creating a loop in $G$. This is because having a loop in the graph reduces the chance of sharing the hyperedges in the loop by more demands (explained in following paragraphs and evaluated in Section 4.5.3). After the first stage, the hypergraph forms a forest. In order to increase the utilization of the circuit switch, in the second stage (L8-10), loop is allowed when adding new hyperedges. This is to serve the demands skipped by the first stage.
When considering a demand in either stage, the receiver racks of the demand may already have the output ports of P2MPCs connected. These racks are called as “conflicting racks” which are expected to be connected via a new P2MPC but they have already been occupied by other P2MPCs (L17). For example, in Figure 4.2(a), receiver rack R11 & R15 of D8 and receiver rack R5 & R14 of D9 are conflicting racks. Solving the conflicting racks is the fundamental crux in leveraging the high-bandwidth of the circuit switch. Monarchy solves the conflicting racks by multi-hopping as follows (L16).

There are two cases in solving the conflicting racks. In Case 1 (L21), the sender rack connects to the input of a P2MPC. In this situation, the sender rack is connected to some of the conflicting racks via existing P2MPCs (L23). In Figure 4.2(a)’s example, the sender rack of D8 has connected to the input of a P2MPC, via which the receiver rack R11 can be reached from the sender rack R7. For the conflicting racks that cannot be reached from the sender rack (L26), the algorithm finds the roots of these conflicting racks (L27) and extends the sender’s P2MPC to connect these roots as well as the non-conflicting racks (L31). The roots and the racks on the path from the roots to the receiver racks become the relay racks of the demand. In Figure 4.2(a), the P2MPC rooted from R7 is extended to R10 (root of conflicting rack R15) and the non-conflicting rack R9. Finding the root of a rack fails if the rack is in a loop. In Figure 4.2(b), three P2MPCs form a loop in the hypergraph where all racks cannot be reached from any racks out of the loop by neither adding new P2MPCs nor extending existing P2MPCs since these racks are connected to the outputs of P2MPCs. Thus, once a demand has conflicting racks in loops, the demand cannot be served in the current epoch. This is why the algorithm avoids creating a loop in the first stage when adding P2MPCs.
In Case 2 (L32), the sender rack is not connected to a P2MPC. The algorithm creates a new P2MPC from the sender rack and connects the new P2MPC to all the roots of the conflicting racks as well as the non-conflicting racks (L38). In Figure 4.2(a)'s example, when considering D9, a new P2MPC is created and it connects to R4 (root of conflict rack R14), R3 (non-conflicting rack, root of conflict rack R5), and R2 (non-conflicting rack).

When extending the outputs of a P2MPC or adding a new P2MPC, the expected number of outputs may exceed the upper limits $f$. The algorithm uses the racks not connecting to any P2MPC input/output as relays to expand the reachability of the sender’s P2MPC (L31,38).

4.4.2 Decide the Epoch Duration

Determining epoch duration can greatly affect the performance of the scheduling algorithm because the circuit reconfiguration introduces a non-trivial delay $\delta$ and it varies from 10 $\mu$s to 100 ms in different circuit switching technologies. Specifically, for the circuit switch having large $\delta$, frequently reconfiguring the circuits results in paying too much overhead due to circuit reconfiguration. On the other hand, for the circuit switch having small $\delta$, after some of the demands finishes in the current epoch, keeping the circuit configuration for extra long time results in a sub-optimal circuit configuration for the remaining demands. Thus, the algorithm should wisely choose the duration of an epoch.

Epoch duration has a direct impact on the effective circuit switch utilization rate (denoted as $EU_{(R^t,\delta)}(D^t)$), which is a function of the epoch duration $D^t$ given the demand schedule of the epoch ($R^t$) and $\delta$. $EU$ represents the effective usage of the circuit switch bandwidth in serving the demands in an epoch and it is defined as
Equ. 4.3.

\[
EU_{(R,\delta)}(D^t) = \frac{\text{bytes of the demands transferred in } D^t}{\text{bytes can be transferred by circuit switch in } (D^t + \delta)} = \frac{\sum_k(R_{ks_k}^t \times \min(d_k.\text{remain()}, b_l \times D^t) \times \sum_j r_{kj})}{b_h \times n \times (D^t + \delta)}
\]

(4.3)

It is straightforward to prove that \(EU_{(R,\delta)}(D^t)\) is a continuous and weakly unimodal piecewise linear function. That is to say, \(EU_{(R,\delta)}(D^t)\) has a unique extreme value, which is also the maximum effective utilization rate. To maximize the effective utilization rate of the circuit switch, in determining the duration of an epoch, the duration that maximizes \(EU\) is chosen. This also greatly helps in reducing the average DCT (evaluated in Section 4.5.3).

4.5 Evaluation

4.5.1 Simulation Setup

Multicast demands: The simulation takes input multicast demands synthesized based on the execution of real applications. The distribution of the multicast data size is collected from iterative natural language processing algorithms, i.e. Word2Vec and LDA, and a database query benchmark, i.e. TPC-H, on Apache Spark. The sizes of the multicast models in Word2Vec with Wikipedia corpus input and LDA with 20 Newsgroups dataset input are 480 MB and 700 MB respectively. The size of the multicast data created in the execution of the TPC-H benchmark ranges from 24 MB to 5.9 GB when the aggregated database table size is 16 GB. In the simulation, this empirical distribution is adopted in creating the multicast data sizes. This distribution (size in GB) fits well to a beta distribution with \(\alpha=0.7\) and \(\beta=1.7\). The number of receiver racks is a uniform distribution from 2 to the half of the racks in
the data center. The sender and receiver racks are uniformly distributed among all the racks. The evaluation uses traffic intensity to quantify the total amount of input multicast demands given to the scheduling algorithm. The traffic intensity is defined as $\Sigma_k (d_k \times \Sigma_j r_{kj})$ divided by $b_t \times n$, which shows the finishing time when all the multicast traffic is sent at the aggregated circuit switch bandwidth when $b_h = b_t$. In the simulation, the traffic intensity of the input multicast demands ranges from 1 s, 5 s and 10 s.

Circuit switch properties: In the simulation, the number of racks ranges from 32 to 256, which are typical data center sizes and cover the network scales evaluated in all the related works. The server NIC bandwidth is 10 Gbps and the circuit switch port bandwidth are set to 10 Gbps, 40 Gbps and 100 Gbps. $\delta$ ranges from 10 $\mu$s to 100 ms, which covers all the recently proposed circuit switch designs. The limit on P2MPC fanout, $f$, is set to 16, which can be easily handled by the sensitivity of today’s optical transceivers.

Comparison baseline: Monarchy is compared against the algorithm proposed in Blast [2]. Blast schedules the multicast demands in hybrid data centers and shows 37 times better performance comparing against overlay peer-to-peer multicast. When scheduling demands in the circuit switch, Blast models the problem as a maximum weighted hypergraph matching problem where the weight of an edge is the data size. The algorithm proposed in Blast first sorts the demands in a decreasing order of $d_k / \Sigma_j r_{kj}$, and checks if the demands can be served based on this order. In the simulation, the algorithm runs multiple rounds until all the demands are scheduled. Multi-hopping and segmented transfer are not considered in Blast.
Figure 4.3: Box plot of the speed-up of average DCT using Blast as the baseline. The center quartile and the ‘x’ (and numbers) are median and average respectively; the upper(lower) quartiles and the upper(lower) whiskers shows 75(25)- and 95(5)-percentile respectively. Each box represents a hundred sets of input demands in the setting with $n=128$, $b_i=10$ Gbps. (a) Various circuit switch port bandwidth and traffic intensity. $\delta=1$ ms. (b) Various $\delta$ and algorithm policies. $b_n=100$ Gbps, traffic intensity=10 s.
4.5.2 Greatly Improved Average DCT

From the results of all the settings listed in Section 4.5.1, for different numbers of racks, Monarchy shows similar trends. For the settings having $\delta \leq 1$ ms, the results are similar as well since in these cases, $\delta$ is negligible compared with the time to transfer the data. The results of the settings having 128 racks and $\delta \geq 1$ ms are presented as representative cases. Figure 4.3(a) presents the box plot of the speed-ups in average DCT. Each box plot corresponds to 100 data points in a setting. For each data point, the speed-up is defined as the average DCT of Blast divided by that of Monarchy. So the higher the speed-up is, the larger improvement shown by Monarchy. The observations of the comparison are as follows.

**Monarchy exhibits up to 13.4× speed-up comparing against the algorithm in Blast.** The improvement comes from two aspects. The first aspect is the order in which the demands are considered. In each iteration, Monarchy starts picking the demand having the smallest remaining size, which is beneficial in reducing average DCT. The improvement can be seen in the cases where $b_h = b_l = 10$ Gbps. The speed-up is about 2.0×. The second aspect is that Monarchy is capable of leveraging the high-bandwidth of the circuit switch port, which can be demonstrated by the following observations. (1) Given the same input demands (fixed traffic intensity), as $b_h$ increases, the speed-up increases significantly, e.g., in the case with 10s traffic intensity, the speed-up increases from 2.4× to 13.4× as $b_h$ increases from 10 Gbps to 100 Gbps. (2) Given the same $b_h$, as the traffic intensity increases, the speed-up increases as well, e.g., in the case having 100 Gbps circuit switch port, the speed-up increases from 6.1× to 13.4×.
4.5.3 Effective Algorithm Features

The design of Monarchy is carefully considered, which is shown by comparing Monarchy against an algorithm having some algorithm features turned off.

Requiring loop-freedom in adding P2MPCs in the first stage significantly improves the speed-up. Monarchy is compared against a similar algorithm whose only difference is that loops are allowed in adding P2MPCs to the graph in the first stage. Figure 4.3(b) shows that the loop-freedom requirement exhibits more than 67% increase in speed-up. This is because that maintaining P2MPCs as a forest increases the chance of solving conflicting racks, which effectively reduces average DCT.

Maximizing circuit switch effective utilization rate significantly improves speed-up when $\delta$ is large. Monarchy is compared against a similar algorithm whose only difference is that the epoch duration is always the time used to finish the demand with the smallest remaining size. As $\delta$ increases from 1 ms to 100 ms (Figure 4.3(b)), maximizing $EU$ exhibits increasing benefits. This is because with large $\delta$, frequently reconfiguring the circuit makes $\delta$ dominate the DCT. Maximizing $EU$ effectively helps to reduce the number of reconfigurations so as to minimize average DCT.

4.6 Chapter Summary

This chapter proposes Monarchy, an algorithm scheduling the multicast demands in a high-bandwidth circuit switch capable of building P2MPC connections. Monarchy adopts multi-hopping and segmented transfer as the approaches. The algorithm effectively leverages the high-bandwidth of the circuit switch ports and minimizes the
average DCT of the multicast demands. Monarchy exhibits significant improvement comparing against the state-of-the-art scheduling algorithm.
Chapter 5

Future Works

This chapter points out a series of promising directions for future works that leverages hybrid data center networks in transferring multicast data.

**In-memory multicast data management:** In Republic, the multicast data are stored in memory for high reading and writing throughput. However, on the servers, the memories are usually very limited especially for the servers running computation workloads. Although the application developers have the responsibility to delete their multicast data if it will no longer be used, the multicast data may still take a significant amount of memories on the server since the computation resource on a server are simultaneously shared by multiple applications. This may result in insufficient memory for the multicast data when it is being created or being received. Assume that a fixed amount of memory, e.g. 10 GB, is reserved for storing multicast data, then the following question comes up naturally. What is the best strategy for managing the multicast data stored in the reserved memory in order to maximize throughput of applications running in the data center? It seems that the solution requires a data management system taking care of the in-memory data in Republic. How such data management system should be design and how it should be integrated to Republic are still unkown and interesting problems.

**Segmented data transfer support in data multicast transfer platform:** Republic assumes that multicast data is delivered in a single transmission, while Monarchy achieves a good performance using segmented transfer. Segmented trans-
fer improves the circuit utilization rate and at the same time avoids the head-of-line blocking problem. Thus, the natural next step is to extend Republic to support segmented multicast data transfer to make it compatible with the data multicast scheduling algorithm like Monarchy. To support segmented transfer, additional information should be added to the data packets in order to indicate which segment a data packet belongs to.

**Co-scheduling for both multicast and unicast data:** Due to the superior capability in forwarding physical layer signal from one input port to multiple output ports, circuit switch shows good performance in sending multicast data. Unicast is just a special case of multicast where there is only a single receiver. Thus, Republic already supports unicast data transfer in a hybrid data center. Many previous works have investigated the problem of scheduling unicast data transfer on a circuit switch in order to take advantage of its scalable port count, high bandwidth and high energy efficiency. To support both unicast and multicast data transfer, a naive hybrid data center may have two circuit switches, one for unicast data and the other for multicast transfer. However, from the operation’s point of view, having two circuit switches in a data center seems to be redundant since a single circuit switch can do both unicast and multicast data transfer. This requires a scheduling algorithm that co-schedules the multicast and the unicast data transfer.

**Resilient and scalable scheduling platform:** In Republic, the Republic manager is a centralized control module which may introduce a single point of failure. In addition, the Republic agent is also the only module that the applications on a server connect to. As a network infrastructure, the data multicast platform should be tolerant to the failures in the whole system. Thus, it is necessary to introduce fault-tolerance in the data multicast platform in order to handle more realistic scenarios.
Multicast protocol using programmable switches: The programmable switches introduce high programmability in the data plane of the network. Network engineers can use the P4 programming language to specify the packet processing behavior of a programmable switch. This significantly expands the design space of network protocols. Thus, it is interesting to design the multicast protocol using P4 and programmable switches. The questions to be solved include “can we design a more efficient and reliable data multicast transfer protocol”, “is there a better way in achieving the coordination between data transfer and circuit setup”, etc.
Chapter 6

Conclusion

This thesis proposes Republic and Monarchy, which support data multicast in hybrid
data centers in different aspects.

Republic is a data multicast platform. It provides a simple interface for big-data
processing frameworks to request the data multicast services. Republic handles the
data transfer reliably and efficiently. It greatly surpasses the performance of the state-
of-the-art multicast mechanisms used in today’s big-data computation frameworks.

Republic is not only a data multicast platform but also a platform for the data
center multicast related research. Republic can be used to examine different multi-
cast data transfer protocols by replacing the protocol in Republic. It can be used to
evaluate different multicast scheduling algorithms by using different scheduling mod-
ule. It can be used to evaluate different multicast-capable data center architectures
by replacing the Republic manager logic. It can also be used as the infrastructure
in building an advanced big-data processing framework with in-network multicast
support. Republic is open source software. It is an effort to revive the in-network
multicast in data center networks.

Monarchy is the first multicast scheduling algorithm tailored for the hybrid data
center with the high-bandwidth signal in the circuit switch. Monarchy adopts seg-
mented transfer and multi-hopping to effectively minimize the average multicast data
completion time. The issue of scheduling always draws a lot of attention from re-
searchers as its solutions could have many variations depending on their assumptions
and optimization goals. Monarchy is the first work in this area and it also sets a high standard for the follow-up works.
Appendix A

Switch Guide

A.1 Switch Architecture

The 42 servers (Section A.1.4) in our testbed are connected via three completely independent networks (Section A.1), i.e., “two-layer network” (Section A.1.1), “one-big-switch network” (Section A.1.2) and “Rice research network” (Section A.1.3). The first two networks are 10 GbE networks and they are used to transfer the traffic in different types of experiments. Each of the two 10 GbE NIC ports on a server is connected to either of these two networks*. The two-layer network is used for the experiments that aim at demonstrating network architecture innovations or the experiments that require architectural level assumptions. The one-big-switch network is for the experiments that merely need non-blocking full 10 GbE bandwidth between servers. The Rice research network is used for accessing the testbed servers from a machine outside the testbed.

A.1.1 The Two-Layer Network

This architecture contains 5 edge EPSs, 1 core EPS, and 1 OCS. For each of the edge EPS, it connects to (1) 8 servers via 8 direct-attached cables (port 1-8 on an edge EPS), (2) the core EPS via 8 direct-attached cables (port 9-16 on an edge EPS), and (3) the OCS via 12 fibers. 8 of the fibers are connected to the 10 GbE ports on

*eth3 is connected to the two-layer network, eth2 is connected to the one-big-switch network
Two-layer network

... 40 computation servers

One-big-switch network

40 computation servers

Rice research network

IP range [INTERNET_NETWORK_PREFIX].50.[111-152]

IP range [INTERNET_NETWORK_PREFIX].20.[111-152]

IP range [RESEARCH_NETWORK_PREFIX].[111-152] (eth0), [211-252] (mgmt)

IP range [INTERNET_NETWORK_PREFIX].20.[111-152]

... 2 controller servers

Figure A.1: Testbed switch architecture

the edge EPS via 10 GbE transceivers (port 17-24 on an edge EPS); 4 of the fibers are connected to the 40 GbE ports on the edge EPS via 40 GbE transceivers (port 49-52 on an edge EPS). The OCS is attached with a few splitters. The two controller servers are directly connected to the core EPS.

Usually, the edge EPS are logically partitioned into multiple logical edge EPS in order to expand the scale of the experiments. For example, when partitioning each of the edge EPSs into 4 logical edge EPSs, the whole architecture will have 20 logical edge EPSs. A larger number of logical edge EPSs is beneficial in demonstrating the performance improvement of optical multicast.

This architecture can achieve non-blocking network between the servers, since on each (logical) edge EPS, the bandwidth to the core EPS or the bandwidth to the OCS is no less than the bandwidth to the servers. Moreover, this architecture can be configured with different downlink-uplink over-subscription ratios other than 1:1,
and the logical partition to the EPS may be adjusted at the same time. For example, when partitioning each of the edge EPSs into 2 logical edge EPSs, each of the logical edge EPSs has 4 uplinks to the core EPS. An over-subscription ratio 4:1, 2:1 or 1:1 can be achieved using 1, 2 or 4 of the uplinks respectively.

Many testbed experiments have been done or can be done with this network. For example, the testbed experiments in the project of Republic [44] used this network. The project like Blast [2], HyperOptics [38], SunFlow [18] and When Creek Meets River [22] can use this network to do the testbed evaluation.

**Extending the Two-Layer Network**

This two-layer network can be used as the basis for more complex network architectural experiments. In the basic two-layer network, on each of the edge EPS, only 24 out of 48 of the 10 GbE ports are occupied; on the OCS, only 60 out of 192 of the circuit ports are occupied. This allows a rich amount of extra links to be connected
between the OCS and the edge EPS (even including the core EPS). The testbed experiments in the project like Flat-Tree [45] and ShareBackup [46, 47] are based on an extension of this architecture.

A.1.2 The One-Big-Switch Network

This architecture contains 1 EPS connecting all 42 servers (Section A.1.4). One-big-switch network offers non-blocking full bandwidth between the servers. This network usually remains unchanged since experiments require network changes should be done using the two-layer network A.1.1. The testbed experiments in Ignem used this network.

A.1.3 The Rice Research Network

Rice research network is a private on-campus network. To access the machines in this network, the MAC address of the accessing machine should be registered with Rice’s Office Information Technology (https://oit.rice.edu/get-help). The 42 servers connect to the Rice research network via two 1 GbE NIC ports, i.e., eth0 (IP is [RESEARCH_NETWORK_PREFIX].1XX) and a management port (IP is [RESEARCH_NETWORK_PREFIX].2XX). eth0 is a standard NIC port that can be seen from the server’s OS and it is used to access the OS from a machine outside the testbed. The management port cannot be seen from the server’s OS and it is only used for the power control of the server.

A.1.4 Servers

The 42 servers in the testbed include 40 computation servers and 2 controller servers. The configurations of these two types of servers are slightly different in terms of CPU,
### Table A.1: Server configurations

<table>
<thead>
<tr>
<th></th>
<th>Computation server</th>
<th>Controller server</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CPU</strong></td>
<td>Intel(R) Xeon(R) CPU E5-1650 v3 @ 3.50GHz</td>
<td>Intel(R) Xeon(R) CPU E5-1650 v2 @ 3.50GHz</td>
</tr>
<tr>
<td><strong>DRAM</strong></td>
<td>8 × Samsung DDR4 2133MHz 16GB M393A2G40DB0-CPB</td>
<td>8 × Samsung DDR3 1066MHz 16GB M393B2G70QH0-YK0</td>
</tr>
<tr>
<td><strong>NIC</strong></td>
<td>Intel I350 Gigabit Network Connection</td>
<td>Intel 82599ES 10-Gigabit SFI/SFP+ Network Connection</td>
</tr>
<tr>
<td><strong>Disk</strong></td>
<td>Seagate Constellation.2 ST91000640NS 1 TB slot 0, /dev/sdc for OS</td>
<td>4 × Seagate Constellation ES.3 ST1000NM0033 1TB slot 1-4, /dev/sd[a-d]</td>
</tr>
<tr>
<td></td>
<td>Seagate Constellation.2 ST91000640NS 1 TB slot 1, /dev/sdb</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Seagate ST1200MM0088 1.2 TB slot 2, /dev/sdb</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Samsung SSD 850 PRO 512 GB slot 3, /dev/sda</td>
<td></td>
</tr>
</tbody>
</table>

DRAM, and disk. The computation servers are suggested to run the computation tasks of the experiments, such as simulator, or the master and slave nodes in a distributed computation platform. Table A.1 lists the detail of the configuration of these two types of servers. The SATA slots on the computation server are numbered from 0 to 9 from the top left to the bottom right corner, e.g., the slots on the first row are slots 0-4. The shown disk configuration applies to most of the servers, while a few of the servers don’t follow that rule.
A.2 Switch Operation

A.2.1 EPS Operation

Quanta provides a set of documents of the EPS (Table A.2, link). The Quanta EPSes are installed with OpenFlow™ Data Plane Abstraction (OF-DPA) v1.0 (v1.0.0.3 in Quanta internal version) software. OF-DPA runs on top of Broadcom Ethernet switch devices and enables the switch to be programmable using OpenFlow protocol v1.3.1. However, due to hardware constraints, the flow tables in OF-DPA, and the flow table matching and the forwarding behaviors between the flow tables are not exactly the same as the generalized flow table pipeline specified in OpenFlow specifications. In a nutshell, the OF-DPA on the Quanta EPS achieves a subset of the OpenFlow v1.3.1 features and adds some constraints to the OpenFlow specification due to hardware constraints. All the detailed rules and constraints beyond OpenFlow are specified in OpenFlow™ Data Plane Abstraction (OF-DPA) Specification (called “Specification” for short).

The EPS has two RJ-45 ports. One is an Ethernet port connecting to the Rice Research Network; the other is the management port, which is a serial port connecting to the RS-232 port on a server in the cluster via RJ-45/RS-232 serial cable. The EPS provides two ways to interact with the switch, i.e. maintenance console and OpenFlow controller.
Maintenance Console (Serial Port or Telnet Connection)

The maintenance console allows the user to configure/check the system level configurations of the EPS. The console can be accessed through the management port (serial port). On the servers connecting to the switch via serial cable, the user uses the command “sudo screen /dev/ttyS0 115200” to access the console. The screen session can be reattached after it is detached (after the terminal is closed) using the command “sudo screen -r <session ID>”. To establish a new screen session, the previous session must be terminated using the command “sudo screen -X -S <session ID> quit”. The session ID can be retrieved from the output of the command “sudo screen -ls”. An alternate way to access the console is using telnet via Rice Research Network (Ethernet port). From a server in the Rice Research Network, the user can access the console using the command telnet <switch IP>.

Change the IP address and TCP port of the OpenFlow controller and the switch ID: An OpenFlow switch actively connects to an OpenFlow controller in order for the controller to configure the switch via the OpenFlow protocol. After each switch reboot, the console asks the user to configure the IP address and the TCP port (6633 by default) of the OpenFlow controller that the switch connects to. The user can also set the datapath ID (DPID) of the switch at the same time. DPID is a unique identifier of a switch on an OpenFlow controller. An OpenFlow controller cannot simultaneously connect to more than one switch having the same DPID†.

Configure link scan and output port queue mode: The OF-DPA provides a set of programs that allow the user to configure the switch via the maintenance console. To set the hardware level configurations, the user can use /mnt/application/

†Configurable DPID of the OF-DPA is introduced by Quanta engineer in v1.0.0.3
client_drivshell program. Two most important configurations are listed here.

1. **Link Scan**: By default, the hardware of each switch port periodically checks the connectivity of the port. If the port is not connected to any other port, e.g. NIC port or switch port, the port is physically turned off. In this state, the green LED of that switch port is off. Due to the periodic connectivity check, it takes about 2 seconds from the time (1) the port is physically connected to another port to the time (2) that the port can forward packets. For the applications that 2-second delay is not acceptable, the user can disable the link scan of the switch ports via the following command ./client_drivshell linkscan disable. Once the link scan is disabled, the state of the port does not change according to the connectivity of the port, i.e., the port keeps the state when link scan is disabled until the link scan is enabled again. With the link scan being disabled, the delay can be reduced to no more than 60 ms. Further reducing the delay depending on the type of the transceiver used and the way the connectivity is changed.

2. **Class of Service (CoS) Queue Configuration**: At each output port, there are 8 hardware queues buffering the outgoing packets. By default, packets in these queues are drained in “strict queue scheduling” mode. The priority values that a packet can be assigned to are from 0 to 15 (15 is the highest priority and 0 is the lowest priority). Priority values from 0 to 6 are mapped to Queue 0 to Queue 6, respectively; priority values from 7 to 15 are mapped queue 7 only. In this strict mode, packets in a queue having a higher priority are sent out first, i.e. packets in a queue are sent out if and only if the queues having higher priorities are empty. The strict mode can be set via the command ./client_drivshell
**cos strict.** The queue that a packet goes to can be set via an OpenFlow rule in the ACL table. There are many other queue scheduling modes, such as weighted, weighted fair, round robin, deficit round robin, bounded, etc. The features of these queue scheduling modes are left for the user to explore.

Other useful commands for the CoS queue configuration are listed here: (1) 
`./client_drivshell cos clear` resets the CoS queue configuration to the default value. (2) `./client_drivshell cos simple` sets the CoS queue scheduling to “round robin” mode. (3) `./client_drivshell cos` shows all the supported command for CoS queue configuration. The output of the above commands is written to the file `/mnt/fastpath/ofdpa.log`.

**Dump/modify flow table:** Under the directory of `/mnt/application/` on OF-DPA, there are programs that allows the maintenance console users to dump and modify the flow/group table entries. To dump the flow table entries, the user can use the command `./client_flowtable_dump <table ID>`. `<table ID>` is a number among 10, 20, ..., 60 each representing a flow table, e.g., VLAN table’s ID is 10 and ACL table’s ID is 60. An alternative way to dump the flow table is using the command `./client_<table name> list --count=<number of entries>`. `<table name>` can be acl and vlan etc. By default, this command shows only 1 entry. To dump all entries, `<number of entries>` can be set to a large number, e.g., 5000, which is larger than the capacity of the flow table. `./client_<table name>` also allows the user to add/modify the flow table entries. To list the group table entries, the user can use the command `./client_group list`. `./client_group` can also add/modify group table entries. Please follow the instructions in *OF-DPA Basic Operation Guide* for examples.

**Upgrade switch software/firmware:** To upgrade the software on the switch,
the user first uses the command `reboot` to reboot the switch. The console will be directed to an installation mode if the user presses any key within a few seconds after the `reboot` command is issued. Please follow the instructions in *Quanta OF-DPA Upgrade Guide* to upgrade the switch.

**OpenFlow Control (Ryu Control Program)**

OF-DPA makes the flow tables and the group table on the EPS programmable using an OpenFlow controller. All the details about the constraints of the flow tables and the group table are listed in *Specification*. Here are some important constraints of the tables.

1. *Packets must have a VLAN ID*: The flow tables drop packets having no VLAN ID. Thus, in order for the packet to be matched with the flow tables, the rules in the VLAN table must tag a VLAN ID to the packets having no VLAN ID.

2. *Forwarding actions are specified in group table*: The actions that forward packets to switch ports can only be installed to the group table. This means that in order to forward a packet, the matched flow table rules should direct the packet to a group table rule containing the action of forwarding the packet the switch port. For example, a unicast action can be achieved in a *OF-DPA L2 interface group entry* (page 39 in *Specification*). A multicast action can be achieved in a *OF-DPA L2 multicast group entry* (page 44 in *Specification*). The multicast group entry includes a list of *OF-DPA L2 interface group entries* each forwards the packet to one of the destination ports. There are distinct and strict naming restrictions on each type of the group tables. When installing group table entries, please exactly follow the restriction in *Specification*. 
3. **Install the core forwarding logic in ACL table**: The ACL table has the richest available match fields for packet matching. So, it is highly suggested to install all the core forwarding logic to the ACL table to avoid being constrained by the limited match fields and actions in other tables.

Ryu OpenFlow controller is adopted for programming the EPSes since Ryu is recommended by Quanta engineers. The controller modules that configure the EPSes in the two-layer network (Section A.1.1) and one-big-switch network (Section A.1.2) are `ofdpa_twolayer.py` and `ofdpa_onebigswitch.py` respectively. The command to start the Ryu module is `ryu-manager <module file name> ryu.app.ofctl_rest --wsapi-port 8010 --ofp-tcp-listen-port 6633`.

The controller module for the two-layer network (`ofdpa_twolayer.py`) allows the user to specify the number of logical edge EPSes on each physical EPS and adjust the over-subscription ratio by changing the corresponding entries in the configuration file `cluster_conf.json`.

To dynamically configure the forwarding rules on the EPSes or monitor the statistics of the EPSes, the user can send requests to the RESTful API exposed by the Ryu controller platform. For more detail regarding the RESTful features and message format, please refer to the Ryu RESTful API documentation [48].

**Tricks and Known Issues in Switch Operation**

1. Buffering packet when changing connection using OCS: This discussion assumes the link scan is disabled. When reconfiguring the connectivity between EPS ports using the OCS, the EPS ports may be notified about the temporary connection break during the OCS circuit reconfiguration. Knowing the connection break, the port of the broken connection may buffer its outgoing packets.
Figure A.3: EPS port connection reconfiguration (a) a reconfiguration that results in packets being buffered at the sender port. (b) a reconfiguration that results in no packet being buffered.

In Figure A.3’s example, the destination of the directed connection from the Sender is changed from Receiver 1 to Receiver 2. If the reconfiguration is done in the way shown in Figure A.3(a), the port of Sender buffers the following packets, while the way in Figure A.3(b) results in no packet buffer. In order to not buffering the packet, the input of Sender and the output of Receiver 1 and Receiver 2 should always be connected to other transceivers.

Buffering the packets on the EPS during the circuit reconfiguration can minimize the packet loss due to the temporary connection break. With buffering, only 1 packet is lost; following packets are buffered on the switch until the buffer is full. Not buffering the packets allows a more accurate estimation of the circuit reconfiguration time (Section A.3).

2. Transceiver connection up delay: There are two types of 10 GbE transceivers in our testbed. With transceivers of the first type, the switch port starts sending
packets about 50 ms after the circuit is physically connected. With transceivers of the second type, the switch port starts sending packets almost right after the circuit is physically connected (occasionally, it also shows about 50 ms delay). The explanation to this difference is that the first type notifies the switch port about 50 ms after the circuit connection is set up so the switch releases the packets in the buffer late, while the second type notifies the switch port right after the circuit connection is set up.

3. Output priority queue for multicast forwarding rule: Rules in the group table can forwarding packet to a specific output queue of a port. However, under the “strict priority” mode, a multicast rule cannot put packets in a queue having priority value larger than 4.

4. Issues when accessing maintenance console via telnet: The telnet server process is loaded to the switch OS only after the IP address and TCP port of the OpenFlow controller is configured after each switch reboot. That is to say, after a switch reboot, the user must connect to the maintenance console via the serial port to configure the IP address and TCP port of the OpenFlow controller so as to access the maintenance console using telnet.

5. Group table entry deletion: Empirically, entries in the group table cannot be successfully/effectively removed through either the maintenance console or the OpenFlow controller. The only way to reliably delete group table entries is rebooting the switch.
A.2.2 OCS Operation

Glimmerglass provides a set of documents of the OCS (Table A.3, link). These documents include very detailed and comprehensive descriptions of the OCS hardware and various ways to configure the OCS via the user interfaces exposed by the OCS software. The aim of this section is to highlight the most important features of the switch and provide an instruction for accessing and configuring the OCS. Reference pointers to the document sections are given when more details can be found in the documents.

The Glimmerglass OCS (System 500) in the testbed has 192×192 LC input-output port pairs. The OCS itself is able to monitor the power of the input and the output light as well as the switching time from the switching hardware’s point of view. The OCS has 4 dedicated variable optical attenuation (VOA) installed on the output port 1 to 4 internally. For more details about the power monitoring and the dedicated VOA, please refer to Input Power Detection (page 7) and Dedicated VOA (page 16) of the System Installation and Maintenance Guide (called “Guide” for short), respectively.

The OCS has dual Ethernet ports and an RS-232 serial port for a controller
machine to access and configure the OCS. One of the Ethernet ports is connected
to the Rice Research Network with IP [RESEARCH_NETWORK_PREFIX].88. The other
Ethernet port is not used. The serial port allows a machine to access the maintenance
console without using Ethernet. For a detailed software interface overview of the OCS,
please refer to *Standard Software* (page 19) in *Guide*.

**Maintenance Console (via SSH Connection and Serial Port)**

Maintenance Console is used to configure the system-level attributes of the OCS. The
system-level configuration includes but not limited to the date and time of the system,
the IP address of the Ethernet ports, firewall and login password, etc. For more details
about the features of the maintenance console, please refer to *Maintenance Console
Overview* (page 91) in *Guide*. The maintenance console can be accessed via either
SSH or serial port.

For SSH access, the connecting machine must be in Rice Research Network. The
command to access the maintenance console via SSH is `ssh [OCS_USERNAME]@[RESEARCH
_NETWORK_PREFIX].88`, followed by typing the password. For more details about SSH
access, please refer to *Log in Using SSH* (page 99) in *Guide*.

For serial port access, the connecting machine must directly connect to the OCS
via a serial cable and serial ports. Serial port access is only suggested when the
OCS is inaccessible via SSH since SSH is much more convenient and it has the same
feature as the serial port access. On a testbed server, the connecting command is
`sudo screen /dev/ttyS0 9600` . For more details about serial port access, please
refer to *Log in Using the Serial Port* (page 97) in *Guide*. 
ClickFlow™ Graphical User Interface (GUI)

ClickFlow™ GUI allows the OCS user to configure the switch using a GUI. The GUI runs as a web server listening to the access to [RESEARCH_NETWORK_PREFIX].88 on the OCS and can be accessed via a browser. ClickFlow™ GUI can configure most of the system-level attributes of the OCS. Table on page 92 in Guide shows the attributes that can be configured with the GUI. ClickFlow™ GUI can configure the optical circuit connections. For a detailed instruction of using the ClickFlow™ GUI, please refer to ClickFlow™ GUI Manual.

ClickFlow™ GUI is good for temporarily reconfiguring the optical circuit connections since the user does not need to write a program to reconfigure the circuit and the circuit connections are shown in a “what you see is what you get” manner.

TL1 User Interface

Transaction Language 1 (TL1) is a general management protocol based on TCP/IP. The OCS software extends the TL1 with a command set that enables command-line and programmatic operation and monitoring of the OCS. With the TL1 user interface, the OCS can be dynamically configured and monitored via programs.

To interact with the OCS via TL1 UI, the control program needs to first connect to the TL1 server via TCP port 11034 and then use ACT-USER command (page 89 in TL1 Manual) to log into the TL1 server and create a TL1 session. To log out and terminate a TL session, the program should use CANC-USER command (page 95 in TL1 Manual).

To setup circuit connections between the OCS ports, the control program use ENT-CRS-FIBER (page 208 in TL1 Manual). The circuit connections command and some other corresponding command can be configured to be executed synchronously
(sync, the default) or asynchronous (async) (page 52 in TL1 Manual). In sync mode, the reply to the command is delayed until the hardware update is confirmed (replied in about 160 ms) so that when the reply is received by the program, the circuit connection is set up completely. In async mode, the TL1 server replies the program without waiting for the confirmation from the hardware (replied in about 20 ms). The async mode is preferable in scenarios where the OCS needs to frequently reconfigured. To minimize the TL1 server response time, the `PowerMonitoringPeriod` (Power Monitoring (page 48) in TL1 Manual) can be set to a lower value, such as 15 ms, using `ED-PARAM` command (page 186 in TL1 Manual).

Please refer to TL1 Manual and TL1 Quick Reference Guide for the details of the TL1 UI.

To dynamically configure the circuit connections in the OCS, we developed a controller platform (`glimmerglass_controller.py`) to interact with the OCS via TL1 language. This platform initially logs in the OCS TL1 server and sets up the default connections on the OCS. The default connections create self-loop connections for all the transceivers, which the user to disable the link scan on the EPSes while maintaining the state of the EPS ports turned on. The platform exposes a RESTful API for the user program to reconfigure the circuit connections. The reconfiguration request is an HTTP POST message carrying a JSON data structure. The URL to the HTTP server is `http://<platform IP>:<platform port>/fiber/ent_crs`. The JSON includes the input/output port pairs and its format is `"inport":[<list of inports>],"outport":[<list of outports>]`, where `<list of inports (outports)>` is a list of the port indices separated by commas.
Tricks and Known Issues in Operation

1. Session timeout: By default, the OCS TL1 server terminates a user session having no activity for 10 minutes (page 2 in TL1 Manual). ED-CID-SECU command can disable or adjust the timeout (page 134 in TL1 Manual). The timeout can also be modified via ClickFlow™ GUI through the following operations: System → System Configuration → Account Options → Inactivity Policy → No-Activity Timeout (minutes).

2. TL1 command length: The incoming command buffer in the TL1 server on the OCS only has about 1024 bytes, which means that one ENT-CRS-FIBER command can only issue about 82 concurrent circuit reconfigurations if the input and the output of the circuit ports are explicitly specified. Fortunately, there exists a trick to effectively reduce the length of the command by omitting the input ports and consecutive ports. For example, ENT-CRS-FIBER::INPUT,OUT5&OUT6&OUT7:1; connects IN1-OUT5, IN2-OUT6, IN3-OUT7. The effect internally is to traverse the input list from IN1 through IN192 and the connected output would be derived from the output list. The above command can also be written as ENT-CRS-FIBER::INPUT,OUT5&&OUT7:1;. The && range can be used to collapse the list further.

3. Signal band assignment (1550nm vs 1310nm): The power monitors of the OCS ports are calibrated according to the wavelength of the input signal (Signal Band Assignment on page 64 in TL1 Manual). To get accurate power measuring of the input signal, the signal band should be set to 1310 nm since the transceivers in the testbed operate at single mode, which generates the signal at the band of 1310 nm. The signal band can be set via the TL1 UI (page 281 in TL1 Manual)
as well as the ClickFlow\textsuperscript{TM} GUI (Configure $\rightarrow$ Signal Types).

A.3 Switch Measurement

This section presents the system built to measure the circuit reconfiguration delay of the OCS and the OpenFlow rule insertion delay of the EPS. The system also measure the switch software delay in handling the reconfiguration request or the rule insertion request. The program and the script performing the measurement is here https://github.com/sunxiaoye0116/republic_manager/tree/master/switch_controller/switch_measurement.

A.3.1 Measurement System

The measurement system is based on the two-layer network architecture (Section A.1.1). When performing a switch measurement, either the OCS or the core EPS is tested. The basic principle of measuring the delay is to use packets as probes to test the end-to-end connectivity of the path between the sender and the receiver. Figure A.4 shows the timeline of the events happening during a switch reconfiguration. During the circuit reconfiguration on the OCS and the rule update on the EPS, the forwarding path between the sender and the receiver breaks temporarily. During this transient state, the tested switch drops the packets. The delay can be calculated by identifying the dropped packets and analyzing the time when the received packets are sent. The duration between the time when the reconfiguration command is issued and the time when the old configuration is removed is called the “software delay”. The duration between the old configuration is removed and the time when the new configuration becomes effective is called the “hardware delay”. The switch measurement system aims at measuring the software delay and the hardware delay of the EPS and the
The diagram of the system measuring the delay are shown in Figure A.5. In terms of hardware, the system includes two computation servers (not under the same logical edge EPS), one (two) edge EPS(es) (one edge EPS if two testing servers are under the same physical edge EPS, two if not), a controller server and a tested switch (the OCS or the core EPS). In terms of software, the system includes (1) a packet sender running on one of the computation servers, (2) a packet receiver running on the other computation server, (3) a configure controller running on the controller server that installs the fixed forwarding rules on the edge EPS(es), (4) a reconfigure controller running on the sender server.
The packet sender keeps sending raw Ethernet packets at a fixed time interval between the packets. The packets have a special Ethernet type so as to differentiate from other IP packets. The time interval between the packets should much smaller than the expected switching time of the tested switch. Each packet carries an increasing sequence number, and a timestamp when the packet is sent to the network. The packet receiver checks if the sequence number of a received packet is not consecutive from the sequence number of the previous packet. The receiver writes the sequence numbers and the timestamps of the beginning and ending packets of the sequence number gap to a file. The difference between the timestamp from the packet at the beginning of the gap and the timestamp from the packet at the ending of the gap is the rule insertion delay of the EPS or circuit reconfiguration delay of the OCS (hardware delay). The configure controller installs the fixed forwarding rules on the edge EPS(es) for the packets sent from the sender to the controller. The reconfigure controller periodically updates the OpenFlow rules in the EPS or the circuit connections in the OCS and switches the two paths from the sender to the receiver back and force. The reconfigure controller writes the timestamp when the reconfiguring request is sent to the switch. The difference between this timestamp and the timestamp of the packet at the beginning of the corresponding sequence number gap is the switch software delay. In this measurement, all the timestamps are generated from the sender server, so there is no need to sycn the time between the servers.

A.3.2 Measurement Instruction

To start the switch measurement, run the script run.py on a machine within the Rice Research Network. The machine has not to be any of the controller, sender or the receiver machines and should be able to access these machines through ssh
Algorithm 3 run.py procedure for switch measurement

1: Set measured devices (EPS or/and OCS)
2: Set a list of number of concurrent reconfigurations \( cr \) for the OCS measurement
3: Ask user to reboot and then configure the EPSes
4: Terminate all controller, sender and receiver programs on testing machines
5: Download experiment files to testing machines
6: Compile the sender and the receiver programs on testing machines
7: Start Edge EPS Configure Controller on controller machine
8: if measure OCS then
9: Start OCS TL1 Controller on controller machine
10: Disable link scan on the edge EPS(es)
11: for \( c \) in \( cr \) do
12: Start receiver program on receiver
13: Start OCS reconfigure controller with \( c \) concurrent reconfigurations on sender
14: Start sender program on sender
15: Wait until no file size change to the reconfigure controller output file
16: Terminate OCS reconfigure controller, sender and receiver programs on testing machines
17: Get \( delay_{software} \), \( delay_{hardware} \) from analyzing the output file of the reconfigure controller and the receiver program
18: add \( c \), \( delay_{software} \), \( delay_{hardware} \) to \( delay_{ocs} \)
19: Terminate OCS TL1 Controller on controller machine
20: else if measure EPS then
21: Start receiver program on receiver
22: Start EPS reconfigure controller on sender
23: Start sender program on sender
24: Wait until no file size change to the reconfigure controller output file
25: Terminate EPS reconfigure controller, sender and receiver programs on testing machines
26: Get \( delay_{software} \), \( delay_{hardware} \) from analyzing the output file of the reconfigure controller and the receiver program
27: add \( delay_{software} \), \( delay_{hardware} \) to \( delay_{eps} \)
28: Terminate Edge EPS Configure Controller on controller machine
29: return \( delay_{ocs} \), \( delay_{eps} \)

without a password. The procedure of the script is shown in Algorithm 3. The script first prepares the testing machines (Line 3-Line 6), then measures the OCS delay (Line 9-Line 19) followed by the measuring of the EPS delay (Line 21-Line 27).

A.3.3 Measurement Results

The measurement results of the EPS and the OCS are shown in Figure A.6 and Figure A.7 respectively. Each of the boxplot represents about 50 measurements. For
Figure A.6: EPS measurement results

Figure A.7: OCS measurement results
the EPS, the switch software delay is about 950 $\mu$s and the hardware delay is about 650 $\mu$s. For the OCS, the switch software delay increases linearly as the number of concurrent reconfigurations increases. The OCS software delay can be expressed as $0.33 \times n + 6.5 \text{ ms}$, where $n$ is the number of concurrent reconfigurations. The OCS hardware delay roughly keeps constant (11.3 ms) when the number of concurrent reconfiguration changes. In Figure A.7(b), the outliers are due to the delay from the transceivers (Section A.2.1).
Appendix B

Republic Tutorial

B.1 Republic API

This section provides a detailed instruction about how an application uses Republic. Republic assumes the application consists of a set of processes running on multiple servers. On a single server, there could be multiple processes belonging to the same application. To use Republic, these processes should talk to the Republic agent through the Republic API shown in Table 3.1. In the current implementation, these interfaces can be used as functions. When making a call to an interface, the application process provides the arguments and receives the return results. The application process makes these interface calls by passing messages with the local Republic agent. The rest of the section shows the message header format the payload format of each message.

In making an interface call, the arguments are provided in a request message and the results are returned in a reply message. Thus, there are two messages (a request message and the reply message) associated to each interface. These messages are forwarded through the Unix domain socket. The path to the file associated with the socket is specified in the Republic agent.

The format of the messages passed through the socket is shown in Figure B.1. The message header contains appID (16 bytes), dataID (8 bytes), message type (4 bytes), payload length (4 bytes), timestamp (16 bytes, for logging and debugging
appID is the globally unique identifier of the application running in the data center. dataID is the identifier of the data within an application. Thus, an appID and a dataID together identify a multicast data within the entire data center. message type is represented as an integer listed in Table B.1. payload length is the number of bytes in the message (including the header and the payload). Each of the message types has its own payload format.

For each interface in Table. 3.1, the application process sends the request message and waits for the reply message, except for the unregister. The rest of this subsection gives a more detailed description of each interface and message type.
B.1.1 register

After the processes are connected to the Republic agent through the Unix domain socket, all these processes should first call the register interface. The purpose of this interface call is to let the agents know the existence of the processes. Among the processes of the application, one process can be specified as a “sender”; others are “receivers”. Only the sender process can send data to the receiver processes. The role (sender or receiver) of a process is provided to the register interface. Once registered, the roles of the processes are not exchangeable.

The payload of register request and register reply message is shown in Figure B.2. Republic assumes all the processes know (1) the IP address of the server where the sender process locates (sender IP) and (2) the name of the application (application name). sender IP is provided to the agent so that the agent can create the control channel with the agent on the server running the sender process. application name is provided by the application process for logging purpose (can be different from appID). sender IP and application name must be the same for all the processes belonging to the same application. processID is the unique identifier of the processes belonging to the same application. processID of the sender process is -1; processID of the receiver processes can be any unique non-negative numbers. The register reply contains the status of the interface call. The code of the status are listed in Table B.2. This code also applies to other reply messages. dataID field in the header is not used in the register interface.

B.1.2 add

Whenever the sender process needs to send data to the receiver processes, the sender process should first write the data to the RAM disk as a file. The root directory of
the RAM disk is specified by the agent. The agent creates a folder under this root directory for this application when the process registers with the agent. The name of this folder is \texttt{broadcast-[4 bytes]-[2 bytes]-[2 bytes]-[2 bytes]-[6 bytes]}, where \texttt{[X bytes]} should be replaced with the corresponding bytes of \texttt{appID} in the HEX representation. For example, if \texttt{appID} is 0x0123456789ABCDEF9876543210FEDCBA, the folder name will be \texttt{broadcast-01234567-89AB-CDEF-9876-543210FEDBCA}. All the multicast data of this application should be written to and read from this folder.
under the root directory. The file name of the multicast data is broadcast-[dataID]. For example, if dataID of a multicast data is 4, then the file name of the data is broadcast-4.

The payloads of add_request and add_reply message is shown in Figure B.3. The sender process makes the add interface call after it has written the broadcast data into the RAM disk. The add_request contains the size (number of bytes) of the multicast data (8-bytes). The add_reply contains the status of the interface call.

B.1.3 send

After the sender process calls add, it can call send to ask the Republic agent to send the data to the list of receivers specified in the send interface call.

The payloads of send_request and send_reply messages are shown in Figure B.4. The payload of send_request contains the number of receivers (8 bytes) of the data transfer, and a list of receiver IPs each with 16 bytes in the string format. The payload of send_reply contains the status of the interface call.

After the sender process makes the send interface call, the age ts of the receiver processes will store the data as a file in the RAM disk. The path to the file on the receiver server is the same as the one on the server running the sender process.
Figure B.4: \texttt{send} message payload

\textbf{B.1.4 read}

The receiver processes may make the \texttt{read} interface call to the Republic agent once the sender process has called \texttt{add} for the corresponding data. The receiver process calls \texttt{read} to the Republic agent to get the instruction about how the data file in the RAM disk should be read to recover the original data.

\texttt{read\_request} has no payload. The payload of \texttt{read\_reply} is shown in Figure B.5. It contains the status of the interface call and three offset pointers (\texttt{start}, \texttt{jump}, \texttt{end}) to the data file in the RAM disk. The receiver process should read the file in three steps: (1) from \texttt{start} to \texttt{jump}, (2) from 0 to \texttt{start}, and (3) from \texttt{jump} to \texttt{end}. Figure B.6 shows three example cases of the data file.

In the case shown in Figure B.6(a), the sequence number of the first packet received by the receiver is 0. The bytes order in the received file is exactly the same as the sender file. To read the file, the receiver process should read from \texttt{start} to \texttt{jump}. 
There is no byte between 0 and \texttt{start} and between \texttt{jump} and end. In the case shown in Figure B.6(b) and Figure B.6(c), the sequence number of the first packet received by the receiver agent is not 0. In Figure B.6(b), data packet having sequence number 0 is not lost during the data channel receiving. In this case, the receiver process should read from \texttt{start} to \texttt{jump} and then from 0 to \texttt{start}. There is no byte between \texttt{jump} and \texttt{end} (\texttt{jump} and \texttt{end} are the same) In Figure B.6(c), data packet having sequence number 0 is lost during the data channel receiving. In this case, the receiver process should read from \texttt{start} to \texttt{jump}, then from 0 to \texttt{start} and finally from \texttt{jump} to \texttt{end}.

\textbf{B.1.5 delete}

The application processes may call the \texttt{delete} interface to remove the data file from the RAM disk in order to save space. In the case where there are multiple processes belonging to the same application make the \texttt{delete} for the same data, the first call will delete the data and following calls will have no effect.

\texttt{delete} request message has no payload. The payload of \texttt{delete_reply} is shown in Figure B.7. It contains the status of the interface call.

<table>
<thead>
<tr>
<th>status (4 bytes)</th>
<th>padding (4 bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>start (8 bytes)</td>
<td>jump (8 bytes)</td>
</tr>
<tr>
<td>end (8 bytes)</td>
<td></td>
</tr>
</tbody>
</table>

Figure B.5: \texttt{read_reply} payload
(a) sequence number of the first received data packet is 0

(b) sequence number of the first received data packet is not 0, packet 0 is not lost

(c) sequence number of the first received data packet is not 0, packet 0 is lost

Figure B.6: Content in the RAM disk file of the data

Figure B.7: delete_reply payload
B.1.6 unregister

Each of the application processes should unregister with the Republic agent when it finishes. To unregister, the process could simply terminate the Unix domain socket connection with the Republic agent. There is no message passing through the socket.

B.2 Experiment Components

This section provides more details about the experiment run in Section 3.5. This includes the introduction about the computation framework (Apache Spark in Section B.2.1), the cluster management platform (Apache Hadoop YARN in Section B.2.2), application workload (Section B.2.3, B.2.4 and B.2.5), Republic agent (Section B.2.6) and Republic manager (Section B.2.7) used in the experiment. Table B.3 lists the repositories mentioned in this section. The usernames, passwords, and IP address prefixes are replaced with the placeholder listed in Table B.4 for security purpose. These placeholders should be replaced with the correct value before using the code.

B.2.1 Computation Framework – Apache Spark

The experiment in the evaluation of Republic uses Apache Spark (version 1.6.1) as an example use case of Republic. Spark uses “broadcast variable” to broadcast an object to all executor processes in the Spark cluster. To use Republic, the broadcast mechanism calling Republic API is added to Spark and the modified Spark is published (https://github.com/sunxiaoye0116/spark_private/tree/v1.6.1_dev-republic/). The majority of the changes is the new BoldBroadcast.scala and corresponding libraries. To compile the modified Spark, run the command “build/mvn -T 6 -Pyarn -Phadoop-2.6 -Dhadoop.version=2.7.4 -DskipTests
<table>
<thead>
<tr>
<th>Repository name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>spark_private</td>
<td>Modefied Spark (Section B.2.1), word2vec library (Section B.2.3), and LDA application (Section B.2.4)</td>
</tr>
<tr>
<td>spark_perf</td>
<td>word2vec application (Section B.2.3)</td>
</tr>
<tr>
<td>data_generator</td>
<td>LDA input corpus generator (Section B.2.4)</td>
</tr>
<tr>
<td>tpch-spark</td>
<td>TPC-H queries on Spark SQL and TPC-H database table generator (Section B.2.5)</td>
</tr>
<tr>
<td>republic_scripts</td>
<td>Example YARN configuration (Section B.2.2)</td>
</tr>
<tr>
<td>republic_agent</td>
<td>Republic agent (Section B.2.6)</td>
</tr>
<tr>
<td>republic_manager</td>
<td>Republic manager (Section B.2.7)</td>
</tr>
</tbody>
</table>

Table B.3: Repositories used in the Republic experiment

<table>
<thead>
<tr>
<th>Placeholder name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[SERVER_USERNAME]</td>
<td>Non-root username of the servers</td>
</tr>
<tr>
<td>[ROOT_USERNAME]</td>
<td>Root username of the servers</td>
</tr>
<tr>
<td>[SERVER_PASSWORD]</td>
<td>Password of the servers</td>
</tr>
<tr>
<td>[OCS_USERNAME]</td>
<td>Username of the OCS</td>
</tr>
<tr>
<td>[OCS_PASSWORD]</td>
<td>Password of the OCS</td>
</tr>
<tr>
<td>[EPS_USERNAME]</td>
<td>Username of the EPS</td>
</tr>
<tr>
<td>[EPS_PASSWORD]</td>
<td>Password of the EPS</td>
</tr>
<tr>
<td>[RESEARCH_NETWORK_PREFIX]</td>
<td>IP address prefix of the experimental network</td>
</tr>
<tr>
<td>[INTERNAL_NETWORK_PREFIX]</td>
<td>IP address prefix of the research network</td>
</tr>
</tbody>
</table>

Table B.4: Placeholders for the security information in the repositories
package" under the root directory of the spark.private repository.

B.2.2 Cluster Management – Apache Hadoop YARN

The experiment uses Apache Hadoop YARN (version 2.7.4) as the cluster management platform. YARN is responsible for monitoring the availability and the usage of the computation resources (including CPU and memory) in the cluster of multiple servers. To run a Spark application, the application should be submitted to the YARN cluster resource manager with the requested number of CPU cores and the amount of memories. The manager will then allocate corresponding cluster resources for the submitted application.

YARN should be configured before it starts. The critical configurations include (1) the list of servers considered to be in the YARN cluster, (2) the number of CPU cores and the amount of memories that can be allocated to the applications on each server, (3) the HDFS directory, (4) the scheduling policy for the submitted applications.

Here is an example YARN configuration. The directory to the configuration files is /etc/ under the root directory of Hadoop. To start a YARN cluster, the first step is to format the HDFS directory using command “hadoop namenode -format”; the second step is to run the command “start-dfs.sh” (start HDFS for logging) and “start-yarn.sh” (start YARN).

B.2.3 word2vec

The word2vec application (https://github.com/sunxiaoye0116/spark-perf/blob/master/mllib-tests/v1p5/src/main/scala/mllib/perf/TestRunner.scala) used in the experiment comes from a Spark benchmark (spark-perf). The application first generates a random input corpus and trains a word2vec model (a skip-gram model).
The class that achieves the model training comes from Spark MLlib.

To compile `spark-perf`, run “`sbt clean; sbt package`” under the root directory of the `spark-perf` repository. To submit the `word2vec` application to the YARN cluster manager, run the following command. The meanings of the variables in the command are shown in Table B.5.

```
${SPARK_HOME}/bin/spark-submit --class mllib.perf.TestRunner --master yarn-cluster --conf spark.executor.heartbeatInterval=50 --conf spark.storage.memoryFraction=0.66 --conf spark.serializer=org.apache.spark.serializer.JavaSerializer --conf spark.shuffle.manager=SORT --conf spark.locality.wait=60000000 --conf spark.broadcast.factory=org.apache.spark.broadcast.BroadcastFactory --conf spark.cores.max=${TOTAL_EXECUTOR_CORES} --conf spark.driver.cores=${CORES_ON_DRIVER} --conf spark.driver.memory=${RAM_ON_DRIVER} --conf spark.executor.instances=${TOTAL_EXECUTORS} --conf spark.executor.cores=${CORES_PER_EXECUTOR} --conf spark.executor.memory=${RAM_PER_EXECUTOR} --conf spark.app.name=A:word2vec_E

TOTAL_EXECUTOR_CORES]:_P:#{BROADCAST_MECHANISM}:_R:#{RUN_INDEX} --conf spark.broadcast.compress=false ${SPARK_PERF_HOME}/mllib-tests/target/mllib-perf-tests-assembly.jar word2vec --num-trials=1 --inter-trial-wait=3 --num-partitions=${TOTAL_EXECUTOR_CORES} --random-seed=5 --num-sentences=${NUM_SENTENCES} --num-words=${NUM_WORDS} --vector-size=${VEC_DIM} --num-iterations=${NUM_ITERATIONS} --min-count=1
```

In `word2vec` model, the broadcast variable size is proportional to `NUM_WORDS` and `VEC_DIM`. 
<table>
<thead>
<tr>
<th>Variable name</th>
<th>Meaning and examples</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SPARK_HOME</strong></td>
<td>Directory to the spark_private repository, e.g., ${HOME}/github/spark_private/</td>
</tr>
<tr>
<td><strong>SPARK_PERF_HOME</strong></td>
<td>Directory to the spark-perf repository, e.g. ${HOME}/github/spark-perf/</td>
</tr>
<tr>
<td><strong>TPCH-SPARK_HOME</strong></td>
<td>Directory to the tpch-spark repository, e.g. ${HOME}/github/tpch-spark/</td>
</tr>
<tr>
<td><strong>BROADCAST_MECHANISM</strong></td>
<td>Broadcast mechanism used in the application, e.g., “Torrent”, “Http”, or “Bold” (Bold uses Republic)</td>
</tr>
<tr>
<td><strong>TOTAL_EXECUTOR_CORES</strong></td>
<td>Total number of cores used by all the executors, e.g., 28</td>
</tr>
<tr>
<td><strong>CORES_ON_DRIVER</strong></td>
<td>Number of cores used by the driver, e.g., 4</td>
</tr>
<tr>
<td><strong>RAM_ON_DRIVER</strong></td>
<td>RAM used by the driver, e.g., 40g</td>
</tr>
<tr>
<td><strong>TOTAL_EXECUTORS</strong></td>
<td>Total number of executors used by the application, e.g., 14</td>
</tr>
<tr>
<td><strong>CORES_PER_EXECUTOR</strong></td>
<td>Number of cores in an executor, e.g., 2</td>
</tr>
<tr>
<td><strong>RAM_PER_EXECUTOR</strong></td>
<td>RAM in an executor, e.g., 40g</td>
</tr>
<tr>
<td><strong>RUN_INDEX</strong></td>
<td>The index of the application run (for the case that needs to run the same application multiple times repeatedly), e.g., 0, 1, or 2, ...</td>
</tr>
<tr>
<td><strong>NUM_SENTENCES</strong></td>
<td>Number of sentences in the corpus (word2vec specific), e.g. 4000000</td>
</tr>
<tr>
<td><strong>NUM_WORDS</strong></td>
<td>Number of unique words in the corpus (word2vec specific), e.g. 210000</td>
</tr>
<tr>
<td><strong>VEC_DIM</strong></td>
<td>Dimension of the vector for each word (word2vec specific), e.g. 300</td>
</tr>
<tr>
<td><strong>NUM_ITERATIONS</strong></td>
<td>Number of training iterations (word2vec and LDA specific), e.g. 10</td>
</tr>
<tr>
<td><strong>VOCABULARY_SIZE</strong></td>
<td>Number of unique word in LDA input corpus (LDA specific), e.g. 10000, 918725 or 2527943 for 1-gram, 2-gram or 3-gram respectively</td>
</tr>
<tr>
<td><strong>QUERY_INDEX</strong></td>
<td>Index of the query among 22 queries (TPC-H specific), e.g. 1, ..., 22</td>
</tr>
<tr>
<td><strong>TOTAL_TABLE_SIZE</strong></td>
<td>Total database table size in GB (TPC-H specific), e.g. 16</td>
</tr>
</tbody>
</table>

Table B.5: Variables in the command for submitting Spark application to YARN
<table>
<thead>
<tr>
<th>Variable name</th>
<th>Meaning and examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>INPUT_CORPUS</td>
<td>Directory to the original corpus, e.g. ./corpus/input</td>
</tr>
<tr>
<td>NGRAM</td>
<td>Number of gram, e.g. 2</td>
</tr>
<tr>
<td>NUM_DOCS</td>
<td>Number of documents in the original corpus to use, e.g. 16923</td>
</tr>
<tr>
<td>NUM_PARTITIONs</td>
<td>Number of partitions made to the entire synthesized corpus, e.g. 28</td>
</tr>
<tr>
<td>NUM_SYN_DOCS</td>
<td>Number of documents in the synthesized corpus, e.g. 10000</td>
</tr>
<tr>
<td>NUM_TOPICS</td>
<td>Number of topics in the LDA model, e.g. 100</td>
</tr>
<tr>
<td>INPUT_NGRAM_CORPUS</td>
<td>Directory to the input n-gram corpus, e.g. ./corpus/input_2_gram/</td>
</tr>
<tr>
<td>OUTPUT_NGRAM_CORPUS</td>
<td>Directory to the output synthesized and partitioned n-gram corpus, e.g. ./corpus/output_2_gram/</td>
</tr>
</tbody>
</table>

Table B.6: Variables in the command for input data generator

### B.2.4 Latent Dirichlet Allocation (LDA)

The LDA model training application used in the experiment comes from [32]. Its implementation has been added to the spark_private repository.

The input data to the LDA application is a synthesized corpus. The input corpus needs to be generated using the data_generator repository before the LDA application runs. There are two steps in generating the synthesized input corpus. The first step is to generate the n-gram corpus; the second step is to synthesize the input corpus from the n-gram corpus from the first step and partition the synthesized n-gram corpus so that a distributed task can read its own partition.

Before running the input data generator, the library file jsc.jar should be installed to the local maven library using the command “mvn install:install-file -Dfile=jsc.jar -DgroupId=uk.co.nildram -DartifactId=jsc -Dversion=1.0
Then, compile the repository using the command "mvn package".

The n-gram corpus generator, i.e. NGramDataGenerator, takes the original unigram corpus [33] and converts it to an n-gram corpus. The original corpus has 16923 documents. The command to generate n-gram corpus is as follow. The meanings of the variables in the command are shown in Table B.6.

```java
java -classpath ./target/classes:./jsc.jar edu.rice.republic.spark_data_generator.NGramDataGenerator ${INPUT_CORPUS} ${NGRAM} ${NUM_DOCS}
```

The output n-gram corpus will be written to the directory of `${INPUT_CORPUS}-${NGRAM}_gram`. The vocabulary for the 1-gram, 2-gram, and 3-gram corpus are 10000, 918725 and 2527943 respectively. In the experiment of Republic, 2-gram corpus is used.

The synthesized corpus generator and partitioner, i.e. DataGenerator, synthesizes the input corpus of the LDA application using the n-gram corpus generated by NGramDataGenerator and then partitions the synthesized corpus. The command to generate the synthesized input corpus is as follow. The meanings of the variables in the command are shown in Table B.6.

```java
java -classpath ./target/classes:./jsc.jar edu.rice.republic.spark_data_generator.DataGenerator lda ${NUM_PARTITIONS} ${NUM_SYN_DOCS} ${NUM_TOPICS} ${INPUT_NGRAM_CORPUS} ${OUTPUT_NGRAM_CORPUS} spark
```

The partitioned synthesized n-gram corpus will be stored under `${OUTPUT_NGRAM_CORPUS}`.

The output file should be copied to the directory of `${SPARK_HOME}/data/` on all
servers in the YARN cluster so that the LDA application tasks can access the input corpus.

The LDA application is compiled together with Spark in the spark_private repository. To submit the LDA application to the YARN cluster manager, run the following command.

```
${SPARK_HOME}/bin/spark-submit --class org.apache.spark.examples.Lda --master yarn-cluster --conf spark.executor.heartbeatInterval=50 --conf spark.storage.memoryFraction=0.66 --conf spark.serializer=org.apache.spark.serializer.JavaSerializer --conf spark.shuffle.manager=sort --conf spark locality.wait=6000000 --conf spark.broadcast.factory=org.apache.spark.broadcast.BroadcastFactory --conf spark.cores.max=${TOTAL_EXECUTOR_CORES} --conf spark.driver.cores=${CORES_ON_DRIVER} --conf spark.driver.memory=${RAM_ON_DRIVER} --conf spark.executor.instances=${TOTAL_EXECUTOR} --conf spark.executor.cores=${CORES_PER_EXECUTOR} --conf spark.executor.memory=${RAM_PER_EXECUTOR} --conf spark.app.name=A:lda_E${TOTAL_EXECUTOR_CORES}:P:${BROADCAST_MECHANISM}R:${RUN_INDEX} --conf spark.broadcast.compress=false \ ${SPARK_HOME}/examples/target/scala-2.10/spark-examples-1.6.1-hadoop2.7.4.jar \ ${SPARK_HOME}/data/output_${NGRAM}_${gram/l}da/WORD_IN_DOC_${TOTAL_EXECUTOR_CORES}.tbl ${VOCABULARY_SIZE} ${NUM_TOPICS} ${NUM_ITERATIONS}
```

For some reason, the LDA application cannot run more than 10 iterations. So NUM_ITERATIONS must be no larger than 10. In the LDA model, the broadcast variable size is proportional to NUM_TOPICS and VOCABULARY_SIZE.
B.2.5 TPC-H Benchmark

TPC-H benchmark is a set of 22 SQL queries used to evaluate the performance of an SQL engine. The benchmark provides a program to generate a set of database tables as the input data. The `tpch-spark` repository provides the support to run these SQL queries on Spark SQL.

To create the database tables, first, compile the database table generator (`dbgen`) using the command `make all`; then, generate the tables using the command `./dbgen -f -q -s ${TOTAL_TABLE_SIZE}`. The database tables with extension `.tbl` will be put in the same directory as `dbgen`. The database tables must be copied to all the servers in the cluster under the directory `${TPCH-SPARK_HOME}/dbgen` in order for the tasks to read the tables locally.

To compile the `tpch-spark` repository, run `sbt package` under `${TPCH-SPARK_HOME}`.

To submit the TPC-H queries to the YARN cluster manager, run the following command.

```
${SPARK_HOME}/bin/spark-submit --packages com.databricks:spark-csv_2.10:1.2.0 --class main.scala.TpchQuery --master yarn-cluster --conf spark.executor.heartbeatInterval=50 --conf spark.storage.memoryFraction=0.66 --conf spark.serializer=org.apache.spark.serializer.JavaSerializer --conf spark.shuffle.manager=sort --conf spark.locality.wait=60000000 --conf spark.broadcast.factory=org.apache.spark.broadcast.$ {BROADCAST_MECHANISM}BroadcastFactory --conf spark.cores.max=${TOTAL_EXECUTOR_CORES} --conf spark.driver.cores=${CORES_ON_DRIVER} --conf spark.driver.memory=${RAM_ON_DRIVER} --conf spark.executor.instances=${TOTAL_EXECUTORS} --conf spark.executor.cores=${CORES_PER_EXECUTOR} --conf spark.executor.memory=${RAM_PER_EXECUTOR} --conf spark.app.name=A:tpch-E:${TOTAL_EXECUTOR_CORES}_P:${BROADCAST_MECHANISM}_R:${RUN_INDEX} --conf
```
spark.broadcast.compress=false --conf spark.default.parallelism=${TOTAL_EXECUTOR_CORES} --conf spark.sql.shuffle.partitions=${TOTAL_EXECUTOR_CORES} --conf spark.sql.autoBroadcastJoinThreshold=10485760 --conf spark.sql.broadcastTimeout=3000 ${TPCH-SPARK_HOME}/target/scala-2.10/spark-tpch-h-queries-2.10-1.0.jar ${QUERY_INDEX} ${TOTAL_TABLE_SIZE} ${TOTAL_EXECUTORS} ${BROADCAST_MECHANISM} ${RUN_INDEX} local

B.2.6 Republic Agent

Before using Republic agent, the netmap kernel module and the corresponding netmap compatible NIC driver should be loaded in the Linux operating system. In addition to that, the network stack should be tuned to make the agent runs efficiently. The config_netmap.sh script sets up the corresponding configurations for the agent. It includes (1) loading netmap and NIC driver, (2) adjusting the MTU of the ethernet interface, (3) configuring LSO, TSO, GSO and LRO offloading options, (4) setting up RSS (Receive-Side Scaling), RPS (Receive Packet Steering) and RFS (Receive Flow Steering), (5) adjusting numbers and sizes of the NIC buffer and rings, (6) mounting RAM disk directory and so on.

To compile Republic agent, run the command make under ./protocol/transceiver.

The command to run Republic agent is as follow.

```
./protocol -e ${DC_NIC} -o ${CC_NIC} -s ${DC_SCHE_POLICY} -S ${DC_SCHE_PRIORITY} -d ${DC_CORE} -c ${CC_CORE} -i ${API_CORE} -q ${NETMAP_QUEUE} -r ${DC_RATE_GBP} -b ${DC_BATCH_SIZE} -a ${DC_ATTEMPT_RATE} -j ${DATA_PAYLOAD_LEN}
```
<table>
<thead>
<tr>
<th>Variable name</th>
<th>Meaning and examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>DC_NIC</td>
<td>Name of the NIC used by the data channel, e.g. <code>eth3</code></td>
</tr>
<tr>
<td>CC_NIC</td>
<td>Name of the NIC used by the control channel, e.g. <code>eth3</code></td>
</tr>
<tr>
<td>DC_SCHE_POLICY</td>
<td>the scheduling policy for the data channel thread, e.g. 1 (SCHED_RR)</td>
</tr>
<tr>
<td>DC_SCHE_PRIORITY</td>
<td>the scheduling priority for the data channel thread, e.g. 99</td>
</tr>
<tr>
<td>DC_CORE</td>
<td>Index of the core used by the data channel thread, e.g. 1</td>
</tr>
<tr>
<td>CC_CORE</td>
<td>Index of the core used by the control channel thread, e.g. 6</td>
</tr>
<tr>
<td>API_CORE</td>
<td>Index of the core used by the Republic API thread, e.g. 6</td>
</tr>
<tr>
<td>NETMAP_QUEUE</td>
<td>Index of the NIC queue used by the data channel, e.g. 1</td>
</tr>
<tr>
<td>DC_RATE_GBP S</td>
<td>Max rate that the data channel can use to send data, e.g. 10.0</td>
</tr>
<tr>
<td>DC_BATCH_SIZE</td>
<td>Max number of packets to send to NIC in each system call in data channel, e.g., 128</td>
</tr>
<tr>
<td>DC_ATTEMPT_RATE</td>
<td>Portion of the <code>DC_RATE_GBP S</code> to send packet at the attempt sending stage, e.g. 0.003</td>
</tr>
<tr>
<td>DATA_PAYLOAD_LEN</td>
<td>Number of bytes of data (payload) that at a data packet contains, e.g. 8822</td>
</tr>
</tbody>
</table>

Table B.7: Variables in the command for running Republic agent
The meanings of the variables in the command are shown in Table B.7.

### B.2.7 Republic Manager

The `republic_manager` repository includes three programs related to the experiment, i.e. (1) Republic manager, (2) OCS controller, and (3) EPS controller. The Republic manager talks to the OCS controller and the EPS controller to configure the OCS and the EPS respectively.

To compile the Republic manager, run `mvn package` under the root directory of the repository. The command to start the Republic manager is as follow.

```bash
java -cp ${REPUBLIC_MANAGER_HOME}/target/classes:${HOME}/.m2/repository/org/apache/thrift/libthrift/0.9.3/libthrift-0.9.3.jar:${HOME}/.m2/repository/org/slf4j/slf4j-api/1.7.12/slf4j-api-1.7.12.jar:${HOME}/.m2/repository/org/apache/httpcomponents/httpclient/4.5.2/httpclient-4.5.2.jar:${HOME}/.m2/repository/commons-logging/commons-logging/1.2/commons-logging-1.2.jar:${HOME}/.m2/repository/commons-codec/commons-codec/1.9/commons-codec-1.9.jar:${HOME}/.m2/repository/org/slf4j/slf4j-log4j12/1.7.16/slf4j-log4j12-1.7.16.jar:${HOME}/.m2/repository/log4j/log4j/1.2.17/log4j-1.2.17.jar:${HOME}/.m2/repository/com/googlecode/json-simple/json-simple/1.1/json-simple-1.1.jar:${HOME}/.m2/repository/org/apache/httpcomponents/httpcore/4.4.4/httpcore-4.4.4.jar:${HOME}/.m2/repository/org/jgroups/jgroups/3.1.0.Final/jgroups-3.1.0.Final.jar:${HOME}/.m2/repository/org/apache/commons/commons-lang3/3.4/commons-lang3-3.4.jar:${HOME}/.m2/repository/commons-cli/commons-cli/1.3.1/commons-cli-1.3.1.jar edu.rice.bold.server.BedController -m ${REPUBLIC_MANAGER_PORT} -t ${TOR_PORT} -o ${OCS_PORT} -s -a -c -p
```
### Variable name | Meaning and examples
---|---
OCS_PORT | TCP port of the OCS controller, e.g. 8080
TOR_PORT | TCP port of the EPS controller, e.g. 8010
OFP_PORT | TCP port of the OpenFlow protocol connecting with the EPS, e.g. 6633
REPUBLIC_MANAGER_PORT | TCP port of the agent-manager interface, e.g. 10880
REPUBLIC_MANAGER_HOME | Directory to the republic_manager repository, e.g. `${HOME}/github/republic_manager`

Table B.8: Variables in the command for running Republic manager

The command to start the OCS controller is as follow.

```
python ./glimmerglass/glimmerglass_controller.py --api port ${OCS_PORT}
```

The command to start the EPS controller is as follow.

```
ryu-manager ./quanta/ofdpa_broadcast.py ryu.app.ofctl_rest --wsapi-port ${TOR_PORT} --ofp--tcp--listen--port ${OFP_PORT}
```

### B.3 Experiment Script

The script to reproduce the Republic experiment is provided in the republic_scripts repository.

The main script is `run.py`. At a high level, the script allows the user to specify (1) the type of the applications to run, (2) the broadcast mechanism (and attempt sending rate if uses Republic), (3) the number of executors used by an application,
(4) the number of runs for each application, (5) the list of servers in the YARN cluster and so on. Table B.9 explained all these configurations. For the first three configurations, the user can provide a list of values. This allows the script to launch multiple “experiments” that are the permutation of all the valid experiment settings. In one experiment, a new YARN cluster is started; applications belonging to the same type run certain times using the same broadcast mechanism; all these applications use the same number of executors. After the applications finish, the script parses the performance metrics from the log. The parsed results are stored under the directory of ./log/. run_app.py is called by the main script to submit applications to YARN cluster manager. parse_all.sh is called by the main script to parse the log.

The republic_scripts repository also provides the scripts to generate the hadoop configuration files (hadoop_config.py) and download all the required files on the servers (prepare_files.sh).
Table B.9: Critical configurations in the Republic experiment script

<table>
<thead>
<tr>
<th>Configuration &amp; variable name</th>
<th>What it is and how to use?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application type</td>
<td>There are two types of applications. One is “tpch” (22 TPC-H queries) the other is “ml” (word2vec and LDA). User can specify a list of application type here, e.g. ['tpch', 'ml'].</td>
</tr>
<tr>
<td>cluster_purpose_l</td>
<td></td>
</tr>
<tr>
<td>Broadcast mechanism protocol</td>
<td>There are three broadcast mechanisms, “Http”, “Torrent”, and “Bold” (i.e. Republic, attempt sending rate is used). User can specify a list of broadcast mechanism, e.g. ['Http', 'Bold']</td>
</tr>
<tr>
<td>attempt_sending_rate</td>
<td>How much portion of the line rate bandwidth is used for sending data packet at the attempt sending stage, only used in Bold(Republic) broadcast mechanism, User can specify a list of attempt sending rate, e.g. [0.01, 0.003]</td>
</tr>
<tr>
<td>republic_attempt_sending</td>
<td></td>
</tr>
<tr>
<td>Number of executors</td>
<td>Number of executors that can be used by each application, e.g. 14</td>
</tr>
<tr>
<td>num_executor</td>
<td></td>
</tr>
<tr>
<td>Number of application runs</td>
<td>Number of repeated runs for each application, e.g. 10</td>
</tr>
<tr>
<td>num_repeat</td>
<td></td>
</tr>
<tr>
<td>Servers</td>
<td>This is the name of the file containing a list of servers, e.g. ‘parallel_hosts’</td>
</tr>
<tr>
<td>cluster_file</td>
<td></td>
</tr>
</tbody>
</table>
Bibliography


