Republic: Data Multicast Meets Hybrid Rack-Level Interconnections in Data Center

Xiaoye Steven Sun — *Rice University*
Yiting Xia — *Facebook Inc.*
Simbarashe Dzinamarira — *Rice University*
Xin Sunny Huang — *Rice University*
Dingming Wu — *Rice University*
T. S. Eugene Ng — *Rice University*

***Abstract*—Data multicast is a crucial data transfer pattern in distributed big-data processing. However, due to the lack of network and system level support, data processing relies on unicast-based application layer multicast. In recent years, there has been a surge in interest in using various emerging circuit switching technologies to build data centers having hybrid packet-circuit switched rack-level interconnections, i.e., hybrid data centers. These physical layer innovations fundamentally change the inter-rack communication capability, especially the capability of multicast communication. We propose Republic, a complete system that addresses the challenging issues in achieving high-performance data multicast in hybrid data centers. Republic abstracts the underlying network complexity as a data multicast service and provides a unified Republic API for data center applications requesting data multicast. Republic is implemented and deployed in a hybrid data center testbed. Testbed evaluation shows that Republic can improve data multicast in Apache Spark machine learning applications by as much as 4.0×.**

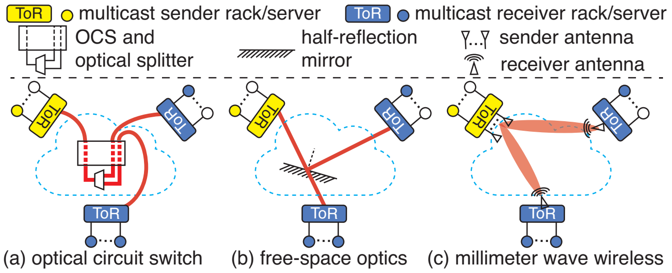
***Index Terms*—data center networks, multicast, circuit switch**

I. Introduction

We live in a world increasingly driven by big-data. Maximizing the value of such massive amount of data relies on large-scale distributed data processing. Data multicast, or one-to-many data dissemination, is a critical data transfer pattern during the workflow of data processing [14], [35]. 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 [13]; 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 [14].

Data multicast is an exorbitant operation for traditional data center networks as it generates large traffic volume. Unfortunately, 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 [23] 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* [14] which is a customized BitTorrent-like overlay protocol, or *naive unicast* where the sender transmits one copy of the data to each receiver. 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.



(a) optical circuit switch (b) free-space optics (c) millimeter wave wireless

Fig. 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.

The emergence of *hybrid rack-level interconnections* revives the in-network multicast in data centers. In addition to the traditional packet-switched network connections, circuit-switched rack-level interconnections built with optical circuit switch (OCS) [16], [25], [32], free-space optics (FSO) [17], [18] or millimeter wave (MMW) wireless [38] are introduced. Some of these technologies have been extended to support physical-layer multicast. As shown in Fig. 1, solutions based on OCS use optical power splitters to divide data signals from the input port to multiple output ports [27], [33], [35]; solutions based on FSO use a cascade of half-reflection switch mirrors to divide light through partial reflection [11]; MMW-based solutions can direct wireless signal to a group of receivers through 3D beamforming antennas [28], [29], [36].

These circuit-switched rack-level interconnections overcome the intrinsic difficulties of in-network multicast in pure packet-switched data center networks [35]. 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. 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, 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 [33], [35].

Different from packet switch, a circuit switch passively directs the signal from an input port to a destinated output port without generating signals. From the perspective of sustainability, such fundamental difference makes the circuit switch massively surpass the interconnection purely built with packet switches. First, the per-port power consumption of a circuit switch is lower than the packet switch in at least an order of magnitude. For example, a 48-port 10GbE electrical packet switch consumes 180 Watt, while a 192-port optical circuit switch only consumes 50 Watt. Second, the circuit switch is 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. These superior properties have driven a number of emergence of the research in circuit-switched rack-level interconnections [16]–[18], [22], [25], [32], [38]. We believe that hybrid rack-level interconnections will eventually be deployed in the next generation of data centers.

Hybrid rack-level interconnections depart from the old assumptions of pure packet-switched interconnections and for the first time make in-network multicast a promising solution in data centers. Accordingly, we claim that 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 far-from-efficient application-layer multicast used by applications, such as the cluster computation frameworks. The major difficulty lies in the lack of abstraction through which application programmers and system engineers can easily leverage the physical-layer technologies. We propose Republic as a *data multicast service* for data centers equipped with hybrid rack-level interconnection ("hybrid data centers" for short).

It is challenging to design a unified data multicast service that employs the ever-emerging multicast enabling technologies. Different circuit switching technologies have different circuit reconfiguration times. How to transmit multicast data as soon as the multicast path is established? 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 packets at a high rate? 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?

Although previous works [11], [27], [33], [35] have demonstrated the potential of supporting data multicast in hybrid data centers, Republic goes one huge step further. Republic is the first effort towards addressing these system-level challenges and providing a full-fledged solution. We view Republic as a system plug-in for a data center. Republic has been deployed in our testbed cluster having 40 servers. We adapt Apache Spark as an example to use Republic's data multicast service. Compared to the state-of-the-art data multicast mechanisms, Republic can speed up the end-to-end data multicast performance by as much as 4.0× in iterative machine learning algorithms and database queries.

II. Challenges

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 traffic also had the expertise to optimize the data transfers in their applications. But today's data scientists, who often are not network specialists, regularly use cluster computation frameworks to run data processing jobs that produce large amounts of traffic. A simple SQL query on a large data set can easily produce several GBs of multicast data (Sec. VI-A). The prevalence and scale of data multicast in these jobs necessitate more efficient handling of data multicast. For example, if a server has multiple processes of a job, 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 data scientists 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. We find *abstraction* as a promising bridge, so that data scientists keep their focuses, while the network experts work concurrently to guarantee 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 [19]–[21]. 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 last packet-switched hop.

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 switching technologies, the circuit reconfiguration time ranges from tens of μs [25] to tens of ms [32]. 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 [15], 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 coordination would be favorable for speed, but no commercial hardware support is available today as far as we know. A software-level solution is desirable for flexibility, compatibility, 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.

III. Republic

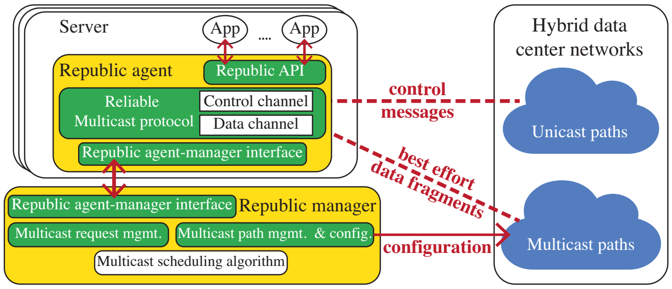

Server
App
....
App
Republic agent
Republic API
Reliable Multicast protocol
Control channel
Data channel
Republic agent-manager interface
Republic agent-manager interface
Republic manager
Multicast request mgmt.
Multicast path mgmt. & config.
Multicast scheduling algorithm
control messages
best effort data fragments
configuration
Hybrid data center networks
Unicast paths
Multicast paths

Fig. 2. System architecture of Republic

Republic addresses all the challenges in leveraging the data multicast capability in hybrid data centers. Fig. 2 shows the system architecture of Republic. Republic includes an agent process on each of the servers and a centralized multicast manager (Sec. III-B). The Republic agent exposes a unified API (Table I) for the data processing applications to request multicast data transfer. The agent handles the transfer using a reliable and efficient data multicast protocol (Sec. III-A) tailored for hybrid data center environment and requests the multicast path via the Republic agent-manger interface (Table II). 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, we briefly show the interaction among applications and the Republic components.

Interaction between application process and local Republic agent (Table I): Each application generates a 16-byte universally unique identifier (UUID) as its unique identifier `appID`, which is known to all the processes of the application. To use Republic, each application process needs to `register` with the local agent. 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 (Sec. III-A2) 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 II): 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 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 (Sec. III-A). 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 [30].

Interface	Description
`register`	Register with the local agent before using the data multicast service.
`unregister`	Unregister from the agent. The calling process cannot use the data multicast service after it unregisters.
`add`	Add the multicast data to the agent after the process has written the data to the in-memory file system. The process provides `dataID` of the multicast data.
`send`	Request to send the multicast data to a set of receivers. The application process should `add` the data to the agent before calling `send`. The process provides `dataID` and the list of `serverID`s of the receivers.
`read`	Request to read the multicast data from the agent. Return with file reading instruction when the data is ready to read.
`delete`	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 `dataID`.

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

Interface	Description
`manager .request`	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 `serverID`s of the receivers and a locally generated unique `requestID`.
`agent .response`	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`.
`manager .release`	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.

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

If the sender agent receives a `response` partially accepting the requested data, the sender agent should send another `request` for the remaining data immediately.

A. 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 the sender and the receivers of each single data multicast. The protocol uses a data channel and a control channel (Fig. 2) 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 III) between the sender and the receivers. The control messages are small unicast messages and require low latency and high reliability. Therefore, the control channel uses packet-switched unicast paths for delivering messages in low latency and it adopts TCP for reliability.

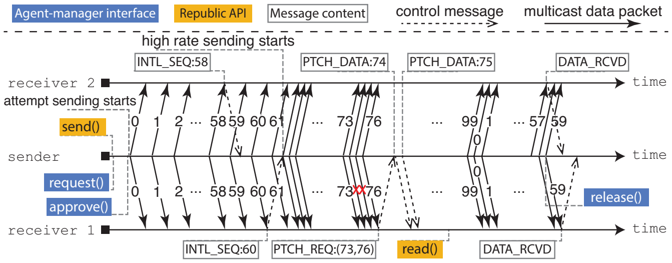


Fig. 3. Timeline of the sender and two receivers in a data multicast example in Republic.

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

TABLE III
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`.

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 (`sn`s 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 session, 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 `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 (Sec. II). 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 Fig. 3's example, the sending start from `sn` 0 to 99. After `sn` 99 is sent, the `sn` goes back to `sn` 0 and continues another round of sequential sending. The receiver tells the sender the initial `sn` it received via `INTL_SEQ` and notifies the sender about receiving completion via `DATA_RCVD`. Only these two control messages are required for each receiver.

1) 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 *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 *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 Fig. 3. 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. Our experience suggests that an attempt sending interval between 1% and 3% of circuit reconfiguration time achieves a good tradeoff (Sec. VI-C). 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 Fig. 3's example, `sender` starts attempt sending from `sn` 0 and `receiver 2` gets initial `sn` 58. However, starting a high-rate transmission (Sec. III-A2) 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_SEQ`s from each of the receivers. In Fig. 3'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 Fig. 3's example, the sender loops back to send `sn` 0 onward and stops after it sends `sn` 59 in the next round.

2) 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 tradeoff 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 [26] 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. Our 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 our 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

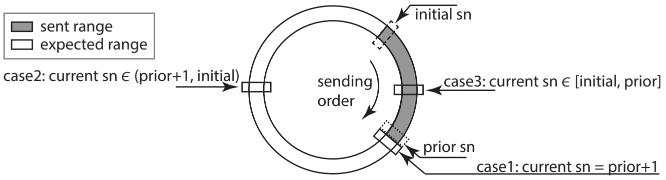

sent range
expected range
initial sn
case2: current sn ∈ (prior+1, initial)
sending order
case3: current sn ∈ [initial, prior]
prior sn
case1: current sn = prior+1

Fig. 4. 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.

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 Fig. 3'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.

3) Recovering Lost Packets Efficiently: According to our deployment experience, three common factors lead to multicast packet losses in hybrid data centers and Republic is designed to avoid and minimize 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 (Sec. III-A1). 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 sns. Republic effectively suppresses type 2 loss by using the kernel-stack-bypass framework to send/receive multicast data packets efficiently (Sec. III-A2). 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 (Sec. III-A2). *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 (Fig. 4). 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 the sns 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 sns 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 sns 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 Fig. 3's example, receiver 1's initial sn is 60. When the current sn is 76 the prior sn is 73. The receiver detects a packet loss since the current sn falls in the expected range but is not the next sn after the prior sn. 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 sn to the ending sn (not including the boundary). Although the receiver doesn't know the number of lost packets if the detected loss range covers sn 0 (sn 0 contains data size), the sender knows exactly the lost packets given the pair of sn 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 Fig. 3's example, the sender retransmits sn 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_DATA`s.

B. 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 [35] 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 paper. 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.

IV. Implementation

Republic agent: Republic agent contains (1) the reliable data multicast protocol (Sec. III-A), (2) the Republic API (Tab. I) and (3) the agent side of the agent-manager interface (Tab. II). 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. We implement the Republic agent program 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) [2] 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 [26] to send/receive UDP packets efficiently. We choose netmap 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.

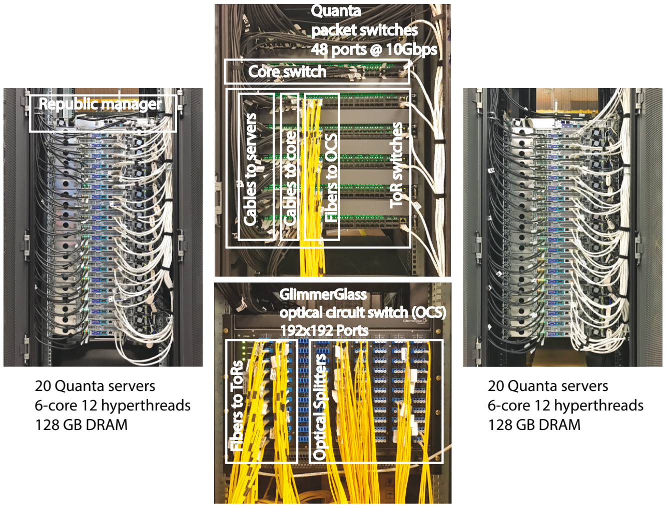


Fig. 5. The deployment of Republic in hybrid data center testbed based on optical circuit switching (OCS) (architecture in Fig. 1(a))

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 [5]. We choose Thrift 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. We implement the Republic manager in 2K lines of Java. We choose Java 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 [7] and manager [8] is open-source.

V. Deployment

We build the OCS-based hybrid data center testbed in Fig. 1(a) and it is shown in Fig. 5. The testbed is built with 40 servers, six 48-port 10 GbE OpenFlow [6] 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 [9] and configures the OCS through a controller talking to the OCS using TL1/telnet commands.

VI. Evaluation

Republic is evaluated in our OCS-based hybrid data center testbed (Sec. V). The evaluation adopts a variety of realistic applications workload (Sec. VI-A). This section shows how and how much Republic reduce end-to-end multicast data transfer time as well as application running time (Sec. VI-B); justifies the design decisions made in Republic agent (Sec. VI-C); finally shows the performance of Republic manager (Sec. VI-D).

A. 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 [3] (Sec. VI-B1 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 [23] implementation in Spark MLlib. The input corpus setting has the same properties as the Wikipedia corpus used in [37]. 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. We use the Spark LDA implementation in [12]. The input corpus is the synthetic 20 Newsgroups dataset [24] 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 [31] is a widely adopted benchmark for database system. The benchmark contains 22 business oriented database queries designed to have broad industry-wide relevance. We run the TPC-H queries on Spark SQL framework [13]. 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.

B. End-to-end Application Level Improvement with Republic

1) Comparison Methodology: The evaluation uses Apache Spark [3] (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. We only replace the broadcast module in Spark 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 Sec. VI-B2, we compare Republic with the state-of-the-art multicast mechanisms adopted in Apache Spark, i.e. *Torrent* (Cornet in [14]) 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.

We set up 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, the packet-switched core bandwidth is 10Gbps and the circuit switch bandwidth

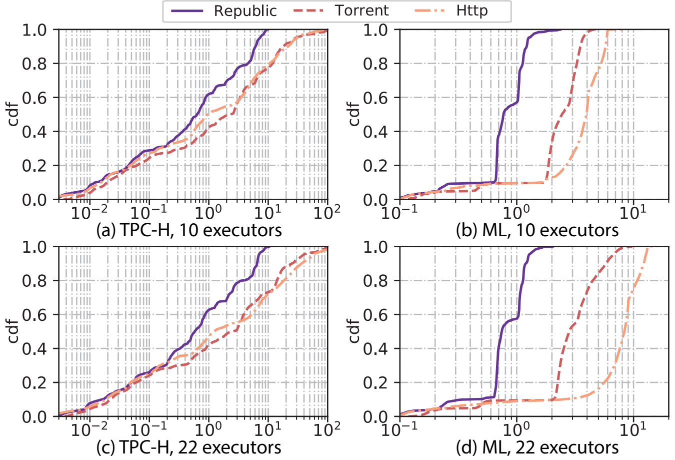


Fig. 6. CDFs of broadcast object reading time in ms

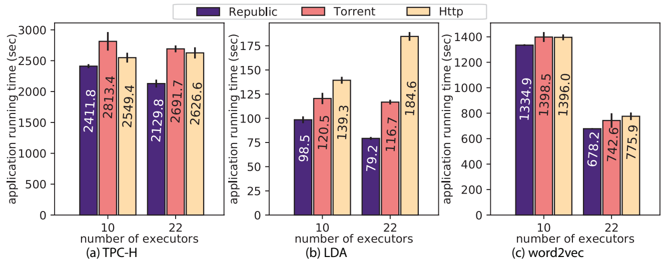


Fig. 7. Application running time with different number of executors

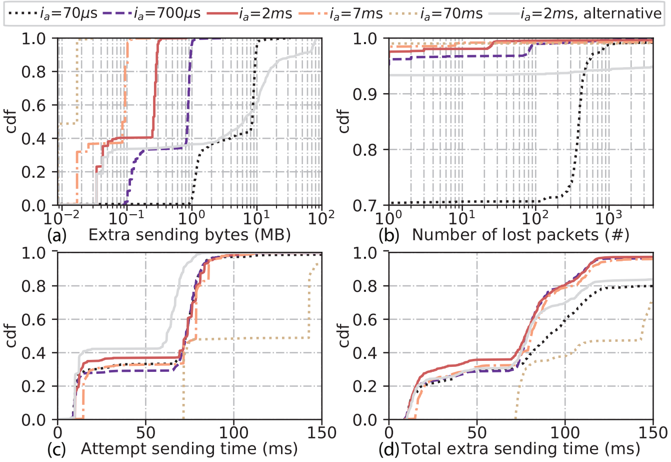


Fig. 8. Efficient data multicast with attempt sending

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.

2) Reduced Broadcast Object Reading Time: Fig. 6 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 (Fig. 6a); in the machine learning applications, Republic improves the reading time by 2.9× at the 40th and 80th persentile comparing with Torrent (Fig. 6b); . 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× and 3.6× improvement at the 40th and the 80th percentile, respectively (Fig. 6d); at 100th percentile in TPC-H, the improvement is 10.7× (Fig. 6c).

The application running time is also improved due to the high-performance data multicast of Republic (Fig. 7). For example, the running time of LDA is improved by 32.1% comparing with Torrent when using 22 executors.

C. 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 (Sec. III-A1). To quantitatively understand the effect of i_a, we examine the cases where i_a varies between 70 μs to 70 ms, which translates to 10% and 0.1% of the 10 Gbps server bandwidth.

For the redundant packets sent to the data channel (Fig. 8a), on our testbed, when i_a is larger than 700 μs, 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 (Fig. 8b), when i_a increases from 70 μs 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 our testbed, when i_a is between between 700 μs and 2 ms (1% and 0.3% of the server bandwidth), the attempt sending time is very close to the attempt sending time that achived when i_a is 70 μs (Fig. 8c). The cases having attempt sending time less than 20 ms reuse the existing circuits (Fig. 8c). In these cases, the circuit reconfiguration delay (about 50-60 ms in our testbed) is eliminated. The delay is caused by rule installation to the packet switches. We use the metric *total extra time* to shows the slow down in transfering 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_RCVD`s 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 (Fig. 8d). However, when i_a is 70 μs, 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, our 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.

Fig. 8 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 (Fig. 8c) due to the large variance on the circuit reconfiguration time. However, starting high-rate sending early cannot reduce the extra sending time (Fig. 8d). 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.

D. High Throughput of Republic Manager

We use *manager response time* to show the achievable throughput of 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 our experiment. Our experiments have about 0.1 request per second, which implies that the Republic manager can handle 7.35K× more concurrent applications that are similar to the applications in our experiments.

VII. 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 [32], a hybrid data center architecture that leverages 3D MEMS-based OCS for fiber optics. Wang *et al.* [33], Samadi *et al.* [27] and Xia *et al.* [35] extend the 3D MEMS-based OCS with passive optical splitters to enable physical-layer multicast. FireFly [18] introduces transparency switchable mirror and galvo (rotating) mirror to direct the free-space optical (FSO) signals between the racks. FlyCast [11] augments FireFly with partial reflection mirror to enable the physical-layer multicast with FSO. Zhou *et al.* [38] proposes a rack-level interconnection solution based on wireless 3D beamforming at 60 GHz. Technologies such as multi-user 3D beamforming [28], [29], [36] 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. We enumerate some of the popular frameworks and the multicast mechanisms they adopt. Spark [3] 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 Sec. VI-B1). In HDFS [1], 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 [10] 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 [4] 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 to improve the data multicast performance as we show in the experience with Spark in the evaluation (Sec. VI-B1).

VIII. Conclusion

In this paper we present Republic, the first fully-fledged solution to data multicast in hybrid data centers. We exploit new physical multicast capabilities in hybrid data centers to design a system that provides reliable and high-performance multicast. Republic is structured as a service, with a simple unified API, making it easily accessible to expert and non-experts alike. We have deployed Republic in our OCS-based hybrid data center testbed and modified Spark to use the service. We observed as much as 4.0× improvement for data multicast. We are currently preparing to open-source the Republic framework so as to provide others with an experimental platform for conducting future research on topics such as multicast scheduling algorithms [30] and new inter-rack network architectures [34].

Acknowledgment

We would like to thank the anonymous reviewers for their thoughtful feedback. This research was sponsored by the NSF under CNS-1422925, CNS-1718980, and CNS-1801884.

References

[1] "Apache hadoop distributed file system (hdfs)," http://hadoop.apache.org/.

[2] "Apache portable runtime (apr)," https://apr.apache.org/.

[3] "Apache spark," http://spark.apache.org/.

[4] "Apache tez," https://tez.apache.org/.

[5] "Apache thrift," https://thrift.apache.org/.

[6] "Openflow," https://www.opennetworking.org/.

[7] "Republic agent," https://github.com/sunxiaoye0116/republic_agent.git.

[8] "Republic manager," https://github.com/sunxiaoye0116/republic_manager.git.

[9] "Ryu openflow controller," https://osrg.github.io/ryu/.

[10] "Tensorflow," https://www.tensorflow.org/.

[11] J. Bao *et al.*, "Flycast: Free-space optics accelerating multicast communications in physical layer," in *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, ser. SIGCOMM '15. New York, NY, USA: ACM, 2015, pp. 97–98.

[12] Z. Cai *et al.*, "A comparison of platforms for implementing and running very large scale machine learning algorithms," in *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '14. New York, NY, USA: ACM, 2014, pp. 1371–1382.

[13] T. Chiba and T. Onodera, "Workload characterization and optimization of tpc-h queries on apache spark," in *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, April 2016, pp. 112–121.

[14] M. Chowdhury *et al.*, "Managing data transfers in computer clusters with orchestra," in *Proceedings of the ACM SIGCOMM 2011 Conference*, ser. SIGCOMM '11. New York, NY, USA: ACM, 2011, pp. 98–109.

[15] N. Farrington *et al.*, "A 10μs hybrid optical-circuit/electrical-packet network for datacenters," in *2013 Optical Fiber Communication Conference and Exposition and the National Fiber Optic Engineers Conference (OFC/NFOEC)*, March 2013, pp. 1–3.

[16] N. Farrington *et al.*, "Helios: A hybrid electrical/optical switch architecture for modular data centers," in *Proceedings of the ACM SIGCOMM 2010 Conference*, ser. SIGCOMM '10. New York, NY, USA: ACM, 2010, pp. 339–350.

[17] M. Ghobadi *et al.*, "Projector: Agile reconfigurable data center interconnect," in *Proceedings of the 2016 Conference on ACM SIGCOMM 2016 Conference*, ser. SIGCOMM '16. New York, NY, USA: ACM, 2016, pp. 216–229.

[18] N. Hamedazimi *et al.*, "Firefly: A reconfigurable wireless data center fabric using free-space optics," in *Proceedings of the 2014 ACM Conference on SIGCOMM*, ser. SIGCOMM '14. New York, NY, USA: ACM, 2014, pp. 319–330.

[19] X. S. Huang, X. S. Sun, and T. S. E. Ng, "Sunflow: Efficient optical circuit scheduling for coflows," in *Proceedings of the 12th International on Conference on Emerging Networking EXperiments and Technologies*, ser. CoNEXT '16. New York, NY, USA: ACM, 2016, pp. 297–311.

[20] H. Liu *et al.*, "Circuit switching under the radar with reactor," in *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'14. Berkeley, CA, USA: USENIX Association, 2014, pp. 1–15.

[21] H. Liu *et al.*, "Scheduling techniques for hybrid circuit/packet networks," in *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies*, ser. CoNEXT '15. New York, NY, USA: ACM, 2015, pp. 41:1–41:13.

[22] W. M. Mellette *et al.*, "Rotornet: A scalable, low-complexity, optical datacenter network," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM '17. New York, NY, USA: ACM, 2017, pp. 267–280.

[23] T. Mikolov *et al.*, "Distributed representations of words and phrases and their compositionality," in *Advances in Neural Information Processing Systems 26*, C. J. C. Burges *et al.*, Eds. Curran Associates, Inc., 2013, pp. 3111–3119.

[24] T. Mitchell, "20 newsgroups," http://kdd.ics.uci.edu/databases/20newsgroups/20newsgroups.html.

[25] G. Porter *et al.*, "Integrating microsecond circuit switching into the data center," in *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, ser. SIGCOMM '13. New York, NY, USA: ACM, 2013, pp. 447–458.

[26] L. Rizzo, "netmap: A novel framework for fast packet i/o," in *21st USENIX Security Symposium (USENIX Security 12)*. Bellevue, WA: USENIX Association, 2012, pp. 101–112.

[27] P. Samadi *et al.*, "Accelerating incast and multicast traffic delivery for data-intensive applications using physical layer optics," in *Proceedings of the 2014 ACM Conference on SIGCOMM*, ser. SIGCOMM '14. New York, NY, USA: ACM, 2014, pp. 373–374.

[28] D. Senaratne and C. Tellambura, "Beamforming for physical layer multicasting," in *2011 IEEE Wireless Communications and Networking Conference*, March 2011, pp. 176–1781.

[29] C. Shepard, A. Javed, and L. Zhong, "Control channel design for many-antenna mu-mimo," in *Proceedings of the 21st Annual International Conference on Mobile Computing and Networking*, ser. MobiCom '15. New York, NY, USA: ACM, 2015, pp. 578–591.

[30] X. S. Sun and T. S. E. Ng, "When creek meets river: Exploiting high-bandwidth circuit switch in scheduling multicast data," in *2017 IEEE 25th International Conference on Network Protocols (ICNP)*, Oct 2017, pp. 1–6.

[31] "Tpc benchmark h," http://www.tpc.org/tpch/, TPC Corp.

[32] G. Wang *et al.*, "c-through: Part-time optics in data centers," in *Proceedings of the ACM SIGCOMM 2010 Conference*, ser. SIGCOMM '10. New York, NY, USA: ACM, 2010, pp. 327–338.

[33] H. Wang *et al.*, "Rethinking the physical layer of data center networks of the next decade: Using optics to enable efficient *-cast connectivity," *SIGCOMM Comput. Commun. Rev.*, vol. 43, pp. 52–58, Jul. 2013.

[34] D. Wu *et al.*, "Hyperoptics: A high throughput and low latency multicast architecture for datacenters," in *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)*. Denver, CO: USENIX Association, 2016.

[35] Y. Xia, T. S. E. Ng, and X. S. Sun, "Blast: Accelerating high-performance data analytics applications by optical multicast," in *2015 IEEE Conference on Computer Communications (INFOCOM)*, April 2015, pp. 1930–1938.

[36] Y. J. Yu *et al.*, "Efficient multicast delivery for wireless data center networks," in *38th Annual IEEE Conference on Local Computer Networks*, Oct 2013, pp. 228–235.

[37] K. Zhai *et al.*, "Mr. lda: A flexible large scale topic modeling package using variational inference in mapreduce," in *Proceedings of the 21st International Conference on World Wide Web*, ser. WWW '12. New York, NY, USA: ACM, 2012, pp. 879–888.

[38] X. Zhou *et al.*, "Mirror mirror on the ceiling: Flexible wireless links for data centers," in *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, ser. SIGCOMM '12. New York, NY, USA: ACM, 2012, pp. 443–454.