LubeRDMA: A Fail-safe Mechanism of RDMA

Shengkai Lin
jefflin@sjtu.edu.cn

Qinwei Yang
yinwai@sjtu.edu.cn

Zengyin Yang
zengyiny@163.com

Yuchuan Wang
yuchuan.wang@gmail.com

Shizhen Zhao
shizhenzhao@sjtu.edu.cn

ABSTRACT

Recent years have witnessed a wide adoption of Remote Direct Memory Access (RDMA) to accelerate distributed systems. As the scale of distributed applications keeps increasing, network failures become more prominent. Although some link/switch failures can be circumvented by in-network rerouting, failures like NIC failure are still fatal in RDMA networks and may cause the entire system to fail.

To address this issue, we propose a fail-safe mechanism of RDMA called LubeRDMA. The core idea is to leverage multiple RDMA NICs on a server and treat them as backups for each other. We introduce a vRDMA model that abstracts a failure-resilient RDMA network to the application. With this model, we achieve RDMA fault tolerance and recovery. In our evaluation, we demonstrate that LubeRDMA efficiently handles RDMA failures while having a minimal impact on RDMA performance.

CCS CONCEPTS

• **Computer systems organization** → **Redundancy**; *Fault-tolerant network topologies*; • **Networks** → *Data center networks.*

KEYWORDS

RDMA, fault-tolerant, backup RNIC, virtual RDMA

ACM Reference Format:

Shengkai Lin, Qinwei Yang, Zengyin Yang, Yuchuan Wang, and Shizhen Zhao. 2024. LubeRDMA: A Fail-safe Mechanism of RDMA. In *The 8th Asia-Pacific Workshop on Networking (APNet 2024), August 03–04, 2024, Sydney, Australia.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3663408.3663411

1 INTRODUCTION

RDMA enables efficient data transfer between local and remote memory without involving CPUs, and has been widely adopted to accelerate large-scale distributed applications in data centers and high-performance computing (HPC) clusters. RDMA offers high throughput, low latency, and low CPU overhead to network applications.

However, as the application scale increases, RDMA failures become more and more detrimental to the application-level performance. Especially for gang scheduled tasks, e.g., distributed AI training, even a single broken RDMA connection could fail the entire task and the failure rate increases with the application size. Although one could adopt a checkpoint mechanism to restart the failed applications, such an application-layer failure recovery method incurs significant overhead. One may improve the RDMA failure-resiliency by rerouting the failed network links/switches, however, such a rerouting strategy cannot handle failures like NIC failure.

We propose LubeRDMA, a fail-safe mechanism of RDMA. The intuition behind LubeRDMA is to utilize multiple physical RDMA NICs (RNICs) on a server and treat them as backups for each other. In the event of a failure on the default path, LubeRDMA seamlessly switches to the backup RNIC (also the backup path) until the default path recovers without impacting the applications. We require LubeRDMA to be application-transparent, meaning that the RDMA applications are not required to change. We implement LubeRDMA in the RDMA userspace library [16].

To realize this intuition, three major challenges need to be addressed. Firstly, considering that RDMA resources and operations are designed to be managed by applications, *how* to manage LubeRDMA resources and operations in an application-transparent manner. Secondly, as the performance and overhead is still the core concerns of RDMA applications, *how* to minimize the overhead and impact on performance while providing fault tolerance capability. Lastly, applications expect that the execution order and the issue order of RDMA data transfer are the same (described in §2.1), *how* to ensure the execution order when switching traffic between RNICs.

LubeRDMA addresses all three of these challenges. We propose the *shadow control verbs* to manage both the default and backup RDMA resources without complicating applications (§3.1). We then propose the *vRDMA model*, which abstracts a failure resilient RDMA network to applications while eliminating the default path overhead (§3.2). Finally, we realize RDMA fault tolerance and recovery in the vRDMA model and minimize the path switching overhead (§3.3 and §3.4).

The paper is organized as below: We first provide the overview of RDMA, the motivation, intuition and challenges of LubeRDMA in §2. Then we introduce the design of LubeRDMA in §3. The evaluation of LubeRDMA is presented in §4. Finally, we discuss the common concerns in §5.

This work does not raise any ethical issues.

APNet 2024, August 03–04, 2024, Sydney, Australia

ACM ISBN 979-8-4007-1758-1/24/08
https://doi.org/10.1145/3663408.3663411

2 BACKGROUND

2.1 RDMA Overview

RDMA allows applications to transfer data between local and remote memory without the participation of CPU. The RDMA workflow can be summarized as follows: the application initiates a work request (WR) for data transfer to the RNIC; the RNIC then processes the WR and provides the results (work completion, WC) to the completion queue (CQ); finally, the application queries the CQ for the transfer results. In this paper, our focus is on the Reliable Connection (RC) transport mode of RDMA, which is the most commonly used mode in real-world applications.

To utilize RDMA, the application need to allocate RDMA resource on both the sender and receiver sides using control verbs. It should open a device (i.e., RNIC) which will be used for communication and obtain a context. Within this context, the application allocates various RDMA resources. First, it allocates a Protection Domain (PD) to isolate resources. Then, it registers a Memory Region (MR) that can be read/write by the RNIC, and gets a local key (lkey) as well as a remote key (rkey). The keys are used to access the MR later by local and remote RNIC. Finally, a Completion Queue (CQ), and a Queue Pair (QP) are created. When creating the QP, the user should specify the local group ID (GID) index, which is associated with the source IP and protocol version (RoCE v1 or RoCE v2). Note all allocated resources are unique to their respective contexts and is isolated between RNICs.

Once the resources have been allocated, the application proceeds to exchange important information. This includes the QP information (QPinfo, including the QP ID, QP number), as well as the MR information (MRinfo, consisting of memory address and rkey). This exchange typically occurs through a TCP connection. Following this, the application modifies the QP to apply the local configuration and the received remote QP information. After the modification, the QP transitions into ready-to-send (RTS) state and is ready to transfer data.

Once a QP is ready, the application can utilize it for RDMA operations using several data verbs. For SEND/RECEIVE operation, known as two-sided operation, the application must first post a receive WR to the receive queue on the receiver and then posts a send WR to the send queue on the sender. In the case of a READ/WRITE operation, which is a one-sided operation, the receiver's participation is not required. The application simply needs to post a send WR (with READ/WRITE operation) to the send queue.

The send WR, with operation like SEND, WRITE, causes actual data transfer. These WRs are completed when the local RNIC receives an ACK from the remote RNIC. The ACK indicates that the remote RNIC has received the data. On the other hand, the receive WR, with operation like RECEIVE, simply waits for incoming data transfer. These WRs are completed when the received data has been written to the memory.

The completion of a WR (i.e., work completion, WC) is reported to the CQ associated with the QP. The application can then retrieve the WC by polling the CQ. In RC transport mode, the RNIC executes the WRs one by one, and the completion order is the same as the execution order as well as the application posting order.

If a WR fails, the current QP immediately enters the error state. As a result, all the WRs that have been posted to the QP but have not been executed go to fail, and their corresponding WCs are reported to the CQ. While the QP is in an error state, any attempt to post WRs will result in an error. The QP must be modified back to RTS state before any further WRs can be processed.

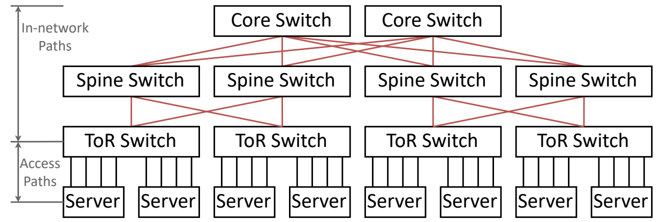

Figure 1: Example of a data center network [1]. One red line stands for 4 black lines.

2.2 Single Point of Failure on Access Paths

In recent years, the significance of large-scale distributed applications, particularly deep learning applications [19, 25], has been increasingly recognized. These applications commonly rely on RDMA networks to achieve high-performance and low-overhead communication.

However, as these larger applications require larger-scale networks, the RDMA network failures are inevitably more frequent. To mitigate the impact of network failures on applications, it is typically desirable for the network to be able to withstand non-fatal failures, which are failures that can be circumvented. An effective approach to achieve this is to utilize multiple paths and consider them as backups for one another. Fortunately, in modern data center networks, it is quite common to have multiple paths between servers [1, 8, 9]. This allows for the possibility of tolerating the network failures. The paths of end-to-end traffic can be divided into two parts, as shown in Figure 1: in-network paths, which refer to the paths between access switches, and access paths, which refer to the paths between servers and access switches.

For in-network paths, failures can be circumvented through the use of dynamic routing mechanisms. If a failure occurs on the default path, the switch can route subsequent packets to an alternate path with the same destination.

However, for access paths, existing RDMA solutions are unable to leverage the multiple paths (i.e., the multple RNICs). Consequently, access links become *single point of failure* (SPOF), whose failures directly impact RDMA functionality and disrupt applications.

Current distributed applications typically handle RDMA failures in application layer. They employ two methods, namely *fault recovery* and *fault tolerance*, neither of which are optimal solutions.

Application fault recovery employs mechanisms such as checkpointing [2, 3, 12–14, 17, 21] or logging [20, 24] to recover failed applications and minimize overhead. However, these solutions introduce CPU and storage overhead due to checkpointing/logging and the subsequent recovery process. Moreover, they inevitably result in wasted machine time, thereby increasing costs [18].

Application fault tolerance aims to prevent the failure of a worker from causing the entire application to fail. This is achieved by dynamically evicting the failed worker and replacing it with a new one from the cluster [4, 10, 11, 18, 23, 26, 27]. Compared to fault recovery solutions, fault tolerance methods help reduce overhead and costs. However, when it comes to handling network failures, these fault tolerance methods still incur unnecessary overhead when it comes to removing and restarting workers, instead of leveraging multiple paths and bypassing the failures.

2.3 Intuition and Challenges

Based on the analysis above, the intuition of LubeRDMA is clear: it utilizes multiple RNICs on a physical host to tolerate failures on the access paths. In addition to the RNIC specified by the application (called default RNIC), LubeRDMA uses other RNICs as backups. In the event of network failures on the default RNIC, LubeRDMA seamlessly switches to the backup RNICs without impacting the application. It first reposts the failed WR to the backup RNIC and then uses the backup RNIC for subsequent WRs until the default path recovers. LubeRDMA does not aim to replace application fault recovery and tolerance but rather to reduce their frequency and the associated overhead on RDMA failures.

However, despite the simplicity of LubeRDMA's intuition, there are three major challenges that need to be addressed:

Challenge #1 As all the RDMA resources and operations are designed to be managed by the application, the challenge is to manage LubeRDMA resources and operations in an application-transparent manner.

Challenge #2 To enable RDMA fault tolerance, some extra operations, like buffering the outstanding WRs, switching between RNICs, are needed. However, it is crucial for LubeRDMA to prioritize performance and minimize both the overhead and impact on performance, regardless of whether network failures occur or not.

Challenge #3 As the WR execution order should be the same as the application posting order, it is necessary to keep the execution order when switching between RNICs.

3 DESIGN

We describe the design of LubeRDMA, a fail-safe mechanism of RDMA, in this section. The overall architecture of LubeRDMA is shown in Figure 2.

We implemented the core logic of LubeRDMA within the RDMA userspace library [16], which we refer to as LubeLib. This involves integrating the LubeRDMA logic into control verbs and data verbs, while ensuring that the verbs remain unchanged from the application's perspective. LubeRDMA can be used by simply changing the RDMA library called by the application (e.g., update the `LD_LIBRARY_PATH` in Linux). As shown in Figure 2, LubeRDMA only addes logic on the control flow, and the data flow (i.e., memory-RNIC-memory) remains unchanged.

To exchange the LubeRDMA-managed resource information among hosts, we employ a LubeDaemon on each host. This LubeDaemon functions as a distributed key-value storage system (e.g., etcd [5] in our implementation), serving a similar purpose as the TCP connection established by the application. LubeLib, on the other hand, communicates with the LubeDaemon via a Unix Domain Socket.

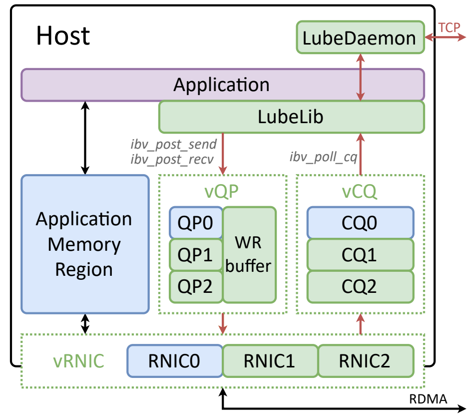

Figure 2: Basic architecture of LubeRDMA. The blue boxes and the green boxes denote the resources managed by the application and LubeLib, respectively. The black arrows and the red arrows denote the data flows and the control flows, respectively.

We then introduce the *shadow control verbs* to utilize the backup RDMA resource in §3.1; the *vRDMA model* to enabling fault tolerance and recovery in §3.2; the RDMA fault tolerance in §3.3; and the fault recovery in §3.4.

3.1 Utilizing the Backup RNIC

To utilize backup RNICs in LubeRDMA, each group of default RDMA resources (including MR, QP, etc.) is equipped with its own backup RDMA resources. While the RNIC may be shared among processes, the RDMA resources remain independent to prevent control interference.

To manage backup RDMA resources for backup RNICs without complicating the applications, we propose the *shadow control verbs*. When an application initializes the default RDMA resources, LubeRDMA implicitly performs the same operations on the backup RNICs, which we refer to as shadow control verbs. The pointers to the backup RDMA resources are stored within the data structure of the default resources, thus they can be used later by LubeRDMA. This mechanism allows LubeLib to create and manage the backup RDMA resources (green boxes in Figure 2) in an application-transparent manner. The detailed process is as below.

First, the application calling `ibv_open_device` to open the RNIC specified by the application (known as the default RNIC). Simultaneously, LubeLib opens backup RNICs, which are specified using environment variables. The order of the backup RNICs also determines their priority as backups.

Next, the application allocates PD and MR, creates CQ and QP. For each of these operations, LubeLib performs the same actions, such as allocating the same addresses as MR for the backup RNICs. To exchange the backup MRinfo and QPinfo with peers, LubeLib sends the mapping of <*default MRinfo* $\longrightarrow$ *backup MRinfo*> and <*default QPinfo* $\longrightarrow$ *backup QPinfo*> to the LubeDaemon after creating the resources. The LubeDaemon, functioning as a distributed key-value storage, stores this information.

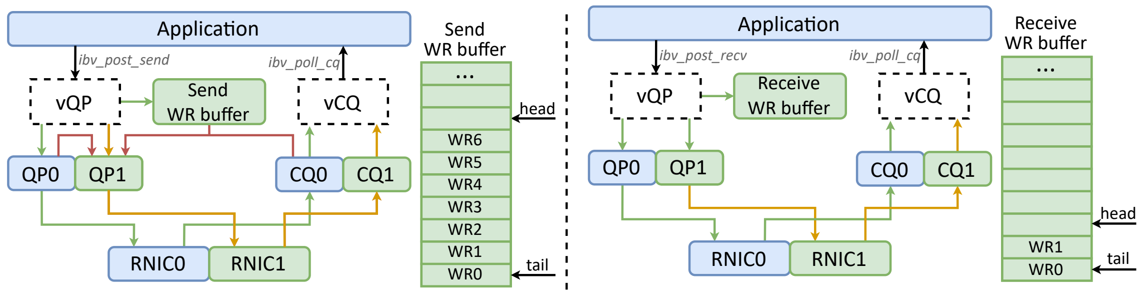

Figure 3: The control flow of send WRs (left graph) and receive WRs (right graph) in LubeRDMA. The green arrows denote the control flow without network failures, while the red and orange arrows denote the control flow with network failures.

Subsequently, the application calls `ibv_modify_qp` to configure the QP and modify it to RTS state. LubeLib performs the same configuration on the backup QPs. As the configuration depends on the peer QPinfo, LubeLib queries the LubeDaemon to obtain this information.

Finally, all the QPs enter RTS state and are ready to perform RDMA operations. LubeLib snapshots the attributes of all the QPs for potential fault recovery (as described in §3.4).

3.2 vRDMA Model

To realize application-transparent RDMA fault tolerance, we then propose the *vRDMA model*.

In the vRDMA model, the RDMA resources used by the application, such as QP and CQ, are virtual resources provided by LubeRDMA. These virtual resources are abstractions of multiple actual resources and do not actually exist, as depicted in Figure 2. This design enables LubeRDMA to proactively handle RDMA failures and seamlessly switch between the different RNICs, ensuring that the application remains unaffected by any RDMA failures.

The WR buffer is the core component of the vRDMA model, as illustrated in Figure 3. It functions to keep track of the outstanding WRs. By utilizing this buffer, LubeLib can access the WRs and repost them if necessary.

In order to minimize the overhead associated with enqueuing and dequeuing WRs in the WR buffer, as well as to maintain the order of the WRs, we implements the WR buffer as a preallocated ring buffer. The newly posted WRs are enqueued at the head pointer, while the completed WRs are dequeued from the tail pointer. This design is effective because the WR completion order is consistent with the order in which they were posted, as explained in §2.1.

When no network failure occurs, the control flow of LubeRDMA is depicted by the green arrows in Figure 3 and described as below.

When the application posts send WRs (with operation such as SEND, WRITE) to the vQP, LubeLib enqueues them into the send WR buffer and posts them to one of the actual QPs. When there are no network failures, the WR is posted to the default QP.

When the application posts receive WRs (with operations such as RECEIVE) to the vQP, LubeLib enqueues them into the receive WR buffer and then posts them to *all* of the actual QPs. This is done to prepare for any potential incoming data transfer. LubeRDMA implicitly aligns the tail pointer of the receive queue of the actual QPs each time the application attempts to post receive WRs. This process dequeues any unnecessary WRs in the backup QPs, preventing the receive queue of the backup QPs from overflowing.

When the application polls for WC, LubeLib polls all the CQs in order. If no network failure occurs, the successful WCs are polled from the default CQ. The WC is then reported to the application, and the corresponding WR is dequeued from the WR buffer.

3.3 RDMA Fault Tolerance

LubeRDMA is designed to tolerate RDMA network failures by seamlessly switching traffic to the backup RNIC. Considering that send WRs are responsible for transmitting data to the remote RNIC, it is enough for LubeRDMA to primarily handling the failures of send WRs. Send WRs can fail at two stages: when posting the WR to the QP, and when the RNIC fetches and executes the WR.

While the naive idea to handle these failures is to simply repost the fail send WR to the backup QP, it can introduce unnecessary overhead. This is because that a fail WR implies that all subsequent WRs posted to the same QP will also fail. If we only address the current failed WR without considering the future failed WRs, it can lead to a continuous triggering of the fault tolerance logic, resulting in additional overhead.

To reduce the overhead, LubeRDMA employs a *repost-and-reset* mechanism, which handles not only the current fail WR, but also all the posted WRs at once. When failure happens in either stage discussed above, the detailed processes are as below (red arrows in Figure 3).

First, LubeLib *reposts* all the outstanding send WRs, which are stored in the WR buffer, to the backup QP. The WR buffer ensures that the order of the reposted WRs remains consistent with the order in which the application posted them. These reposted WRs are then executed by the backup RNIC. LubeLib marks the backup QP as in-use, and subsequent send WRs from the application will directly use the backup QP, depicted as the orange arrows in Figure 3.

Second, LubeLib *resets* the QP from the error state. The reset operation clears all the WRs and WCs in the QP and prevents subsequent failed WCs (which have already been reposted) from continuously disrupting LubeLib.

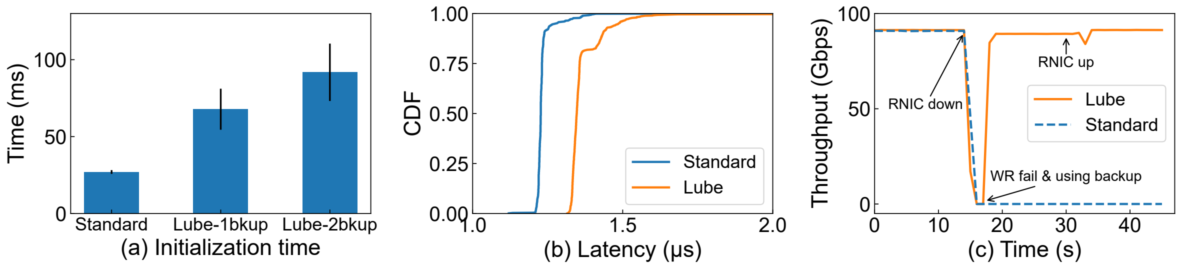

Figure 4: (a) The initialization time (with its standard deviation) of standard RDMA, LubeRDMA with 1 and 2 backup RNICs. (b) The latency CDF for standard RDMA and LubeRDMA. (c) The throughput per second of standard RDMA and LubeRDMA. In the throughput test, we intentionally turn down and up the RNIC to show the fault tolerance and recovery of LubeRDMA. Lube is short for LubeRDMA, and standard is short for standard RDMA.

It is worth noting that although the above description primarily focuses on handling failures on the default path, the same mechanism applies if both the default path and the first backup path fail. LubeLib attempts the backup paths in the priority order (as described in §3.1). If all the paths fail, the failure will be reported to the application.

3.4 RDMA Fault Recovery

When the network failure on the default path is resolved, LubeRDMA switches back to using the default QP, which is called fault recovery. This is necessary because, considering the system architecture, the application-specified default RNIC is typically the optimal choice for communication.

Fault recovery is necessary on both the sender and receiver sides, as the QPs on both sides may become unavailable during network failures. On the receiver side, an RNIC failure can result in a change in the GID index, rendering the QP with an incorrect GID index unable to send or reply any data. On the sender side, in addition to the GID index issue, a failed send WR can also cause the QP to enter the error state. In such cases, the QP is unable to transmit or receive data until it is reset and recovered.

LubeRDMA adopts a *wait-and-retry* mechanism for fault recovery. When a failure occurs or a retry is attempted, LubeLib records the timestamp and *waits* for a user-defined time interval (e.g., 3 seconds) before the next retry. This helps prevent excessively frequent retry attempts.

LubeLib *retries* the default QP when it polls a send or receive WC from the backup CQ (indicating possible failures on the default path) and has waited for at least the time interval. If the default QP is not in RTS state, the retry process first modifies the default QP back to RTS state by the QP attributes snapshot taken at starting (as described in §3.1), along with the new GID index. If the QP successfully transitions back to RTS state, LubeLib reposts the receive WRs to the default QP for potential incoming data transfer. Then, LubeLib switches the in-use QP back to the default QP, and the subsequent WRs will be posted to the default QP. If the subsequent WRs encounter failures again, they will be handled again by the failure tolerance described in §3.3.

Discussion. There are still shortcomings in the recovery mechanism: Firstly, it only works for two-sided operations such as SEND and RECV. This is because only two-sided operation invokes the data verbs on both sides, allowing for the detection of default path recovery. Secondly, the retry-based mechanism still has nonnegligible overhead for high-speed communication. Lastly, when LubeLib switches communication back to the default RNIC, there is a possibility that newly posted WRs may be executed before the former WRs, which are still queuing in the backup RNIC, thus disrupt the WR execution order. We would like to leave the improvement in future work.

4 EVALUATION

In this section, we present the evaluation results of LubeRDMA. Although the evaluation is still ongoing, the current results provide an overview of the performance of LubeRDMA.

Our testbed consists of two servers equipped with Intel Xeon Silver 4110 CPU (32 cores, 2.10GHz) and 128 GB memory. Each server ia equipped with a dual-port Mellanox Connect-X5 100 Gb NIC and a dual-port BlueField-2 100Gb NIC, located on different NUMA (Non-Uniform Memory Access) nodes. The Connect-X5 NICs are directly connected to each other, as are the BlueField-2 NICs, using 100 Gb copper cables. The RNICs are configured to operate in RoCEv2 mode. We install MLNX_OFED v5.8 on both servers. The two RNICs serve as backup of each other.

We have developed LubeLib based on rdma-core v48 [16], and LubeDaemon based on etcd v3.5.11 [5].

4.1 Control-plane Overhead

We conduct experiments to evaluate the initialization time of LubeRDMA. We time the typical initialization operations, including opening the device, allocating a PD, registering a MR, creating a CQ, creating a QP, and modifying the QP to RTS state, for LubeRDMA and standard RDMA. Specifically, we compare LubeRDMA with 1 backup RNIC and 2 backup RNICs, against standard RDMA.

Figure 4 (a) shows the results. It can be observed that allocating additional RDMA resources does increase the initialization time. With a standard RDMA setup, the initialization time is 27 ms. When utilizing one backup RNIC, the initialization time increases to 67.8 ms. With two backup RNICs, the time further increases to 91.8 ms.

Discussion. Currently, the RDMA initialization overhead is caused by allocating backup RDMA resources. The overhead can be minimized by employing multiple threads for resource initialization. We leave this implementation in future work.

4.2 Data-plane Performance

In this section, we evaluated the latency and throughput of LubeRDMA and demonstrated its fault tolerance and recovery capabilities. To measure the performance, we utilized the `ib_write_lat` and `ib_write_bw` tools from the perftest package [15].

RDMA Latency. Figure 4 (b) shows the CDF graph of RDMA latency for LubeRDMA and standard RDMA. The average latency of LubeRDMA is measured to be 1.38 μs, while the standard RDMA latency averaged at 1.23 μs. These results indicate that the vRDMA model has a slight impact on RDMA latency.

Discussion. The data-plane overhead is primarily caused by the enqueuing and dequeuing operations of the WR buffer. This overhead can be reduced by utilizing multi-threads to perform these operations. We leave this to future work.

RDMA Throughput. We then evaluate the throughput of LubeRDMA and demonstrate its fault tolerance and recovery abilities, as shown in Figure 4 (c). Prior to the RNIC down, the thoughput of LubeRDMA is the same as that of standard RDMA, reaching 91 Gbps. At the 15th second, we intentionally turned down the sender RNIC, causing the throughput to drop to 0.[1] After about 2 seconds, the RNIC eventually reports a fail WC. It is important to note that the 0 throughput observed during this period was caused by the packet retransmission conducted by the RNIC.

With standard RDMA, the failure is reported to the application, finally leading to the termination of the test. In contrast, LubeRDMA switches to the backup RNIC upon receiving the WR failure. The throughput restores to 89 Gbs, albeit slightly lower due to the cross-NUMA overhead. Once the default RNIC recovers, the traffic is switched back to it, and the throughput returns to its previous level.

5 DISCUSSION

QP overhead. The creation of additional QPs in LubeRDMA for backup purposes does not negatively impact RDMA performance as reported in previous research studies [22]. This is because the backup QPs in LubeRDMA are typically inactive, resulting in no cache misses on the RNICs and, therefore, no impact on performance.

GPUDirect RDMA (GDR) [7]. GDR allows direct access of GPU memory by the RNIC, effectively reducing the RDMA overhead associated with GPUs. The usage of GDR does not alter the application's usage of RDMA as described in §2.1, thus LubeRDMA is compatible with GDR.

Large-scale key-value storage. Currently, LubeRDMA relies on distributed key-value storage for initialization. However, as reported by etcd [6], larger cluster sizes can impact the storage's performance and potentially introduce overhead during LubeRDMA initialization. We plan to evaluate and address this issue in future work.

[1]The throughput seems to decrease slowly because each data point is the average throughput of last 1 second.

6 CONCLUSION

In this paper, we introduce LubeRDMA, a fail-safe mechanism for RDMA. LubeRDMA utilizes multiple RNICs on a server and employs them as backups for one another. LubeRDMA offers fault tolerance and recovery capabilities for RDMA while minimizing impact on RDMA performance.

ACKNOWLEDGMENTS

We sincerely thank anonymous reviewers for their helpful comments. This work was supported by the NSF China (No. 62272292).

REFERENCES

[1] Wei Bai, Shanim Sainul Abdeen, Ankit Agrawal, Krishan Kumar Attre, Paramvir Bahl, Ameya Bhagat, Gowri Bhaskara, Tanya Brokhman, Lei Cao, Ahmad Cheema, Rebecca Chow, Jeff Cohen, Mahmoud Elhaddad, Vivek Ette, Igal Figlin, Daniel Firestone, Mathew George, Ilya German, Lakhmeet Ghai, Eric Green, Albert Greenberg, Manish Gupta, Randy Haagens, Matthew Hendel, Ridwan Howlader, Neetha John, Julia Johnstone, Tom Jolly, Greg Kramer, David Kruse, Ankit Kumar, Erica Lan, Ivan Lee, Avi Levy, Marina Lipshteyn, Xin Liu, Chen Liu, Guohan Lu, Yuemin Lu, Xiakun Lu, Vadim Makhervaks, Ulad Malashanka, David A. Maltz, Ilias Marinos, Rohan Mehta, Sharda Murthi, Anup Namdhari, Aaron Ogus, Jitendra Padhye, Madhav Pandya, Douglas Phillips, Adrian Power, Suraj Puri, Shachar Raindel, Jordan Rhee, Anthony Russo, Maneesh Sah, Ali Sheriff, Chris Sparacino, Ashutosh Srivastava, Weixiang Sun, Nick Swanson, Fuhou Tian, Lukasz Tomczyk, Vamsi Vadlamuri, Alec Wolman, Ying Xie, Joyce Yom, Lihua Yuan, Yanzhao Zhang, and Brian Zill. 2023. Empowering Azure Storage with RDMA. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. USENIX Association, Boston, MA, 49–67. https://www.usenix.org/conference/nsdi23/presentation/bai

[2] Paris Carbone, Stephan Ewen, Gyula Fóra, Seif Haridi, Stefan Richter, and Kostas Tzoumas. 2017. State management in Apache Flink®: consistent stateful distributed stream processing. *Proc. VLDB Endow.* 10, 12 (Aug. 2017), 1718–1729. https://doi.org/10.14778/3137765.3137777

[3] Assaf Eisenman, Kiran Kumar Matam, Steven Ingram, Dheevatsa Mudigere, Raghuraman Krishnamoorthi, Krishnakumar Nair, Misha Smelyanskiy, and Murali Annavaram. 2022. Check-N-Run: a Checkpointing System for Training Deep Learning Recommendation Models. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. USENIX Association, Renton, WA, 929–943. https://www.usenix.org/conference/nsdi22/presentation/eisenman

[4] Elastic Horovod 2024. Elastic Horovod. https://horovod.readthedocs.io/en/latest/elastic_include.html.

[5] etcd 2024. etcd. https://etcd.io/.

[6] etcdscale 2024. What is maximum cluster size? https://etcd.io/docs/v3.5/faq/#what-is-maximum-cluster-size.

[7] gdr 2024. GPUDirect RDMA. https://docs.nvidia.com/cuda/gpudirect-rdma/index.html.

[8] Phillipa Gill, Navendu Jain, and Nachiappan Nagappan. 2011. Understanding network failures in data centers: measurement, analysis, and implications. *SIGCOMM Comput. Commun. Rev.* 41, 4 (Aug. 2011), 350–361. https://doi.org/10.1145/2043164.2018477

[9] Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitu Padhye, and Marina Lipshteyn. 2016. RDMA over Commodity Ethernet at Scale. In *Proceedings of the 2016 ACM SIGCOMM Conference* (Florianopolis, Brazil) *(SIGCOMM '16)*. Association for Computing Machinery, New York, NY, USA, 202–215. https://doi.org/10.1145/2934872.2934908

[10] Changho Hwang, Taehyun Kim, Sunghyun Kim, Jinwoo Shin, and KyoungSoo Park. 2021. Elastic Resource Sharing for Distributed Deep Learning. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. USENIX Association, 721–739. https://www.usenix.org/conference/nsdi21/presentation/hwang

[11] Insu Jang, Zhenning Yang, Zhen Zhang, Xin Jin, and Mosharaf Chowdhury. 2023. Oobleck: Resilient Distributed Training of Large Models Using Pipeline Templates. In *Proceedings of the 29th Symposium on Operating Systems Principles* (, Koblenz, Germany,) *(SOSP '23)*. Association for Computing Machinery, New York, NY, USA, 382–395. https://doi.org/10.1145/3600006.3613152

[12] Jayashree Mohan, Amar Phanishayee, and Vijay Chidambaram. 2021. CheckFreq: Frequent, Fine-Grained DNN Checkpointing. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*. USENIX Association, 203–216. https://www.usenix.org/conference/fast21/presentation/mohan

[13] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. 2013. Naiad: a timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (Farminton,

Pennsylvania) *(SOSP '13)*. Association for Computing Machinery, New York, NY, USA, 439–455. https://doi.org/10.1145/2517349.2522738
- [14] Bogdan Nicolae, Jiali Li, Justin M. Wozniak, George Bosilca, Matthieu Dorier, and Franck Cappello. 2020. DeepFreeze: Towards Scalable Asynchronous Checkpointing of Deep Learning Models. In *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*. 172–181. https://doi.org/10.1109/CCGrid49817.2020.00-76
- [15] perftest 2024. linux-rdma/perftest. https://github.com/linux-rdma/perftest.
- [16] rdma-core 2024. rdma-core. https://github.com/linux-rdma/rdma-core.
- [17] G.A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D.I. August. 2005. SWIFT: software implemented fault tolerance. In *International Symposium on Code Generation and Optimization*. 243–254. https://doi.org/10.1109/CGO.2005.34
- [18] John Thorpe, Pengzhan Zhao, Jonathan Eyolfson, Yifan Qiao, Zhihao Jia, Minjia Zhang, Ravi Netravali, and Guoqing Harry Xu. 2023. Bamboo: Making Preemptible Instances Resilient for Affordable Training of Large DNNs. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. USENIX Association, Boston, MA, 497–513. https://www.usenix.org/conference/nsdi23/presentation/thorpe
- [19] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurelien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. 2023. Llama 2: Open Foundation and Fine-Tuned Chat Models. arXiv:2307.09288 [cs.CL]
- [20] Stephanie Wang, John Liagouris, Robert Nishihara, Philipp Moritz, Ujval Misra, Alexey Tumanov, and Ion Stoica. 2019. Lineage stash: fault tolerance off the critical path. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 338–352. https://doi.org/10.1145/3341301.3359653
- [21] Zhuang Wang, Zhen Jia, Shuai Zheng, Zhen Zhang, Xinwei Fu, T. S. Eugene Ng, and Yida Wang. 2023. GEMINI: Fast Failure Recovery in Distributed Training with In-Memory Checkpoints. In *Proceedings of the 29th Symposium on Operating Systems Principles* (, Koblenz, Germany,) *(SOSP '23)*. Association for Computing Machinery, New York, NY, USA, 364–381. https://doi.org/10.1145/3600006.3613145
- [22] Zilong Wang, Layong Luo, Qingsong Ning, Chaoliang Zeng, Wenxue Li, Xinchen Wan, Peng Xie, Tao Feng, Ke Cheng, Xiongfei Geng, Tianhao Wang, Weicheng Ling, Kejia Huo, Pingbo An, Kui Ji, Shideng Zhang, Bin Xu, Ruiqing Feng, Tao Ding, Kai Chen, and Chuanxiong Guo. 2023. SRNIC: A Scalable Architecture for RDMA NICs. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. USENIX Association, Boston, MA, 1–14. https://www.usenix.org/conference/nsdi23/presentation/wang-zilong
- [23] Lei Xie, Jidong Zhai, Baodong Wu, Yuanbo Wang, Xingcheng Zhang, Peng Sun, and Shengen Yan. 2020. Elan: Towards Generic and Efficient Elastic Training for Deep Learning. In *2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS)*. 78–88. https://doi.org/10.1109/ICDCS47774.2020.00018
- [24] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. 2013. Discretized streams: fault-tolerant streaming computation at scale. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (Farminton, Pennsylvania) *(SOSP '13)*. Association for Computing Machinery, New York, NY, USA, 423–438. https://doi.org/10.1145/2517349.2522737
- [25] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, Todor Mihaylov, Myle Ott, Sam Shleifer, Kurt Shuster, Daniel Simig, Punit Singh Koura, Anjali Sridhar, Tianlu Wang, and Luke Zettlemoyer. 2022. OPT: Open Pre-trained Transformer Language Models. arXiv:2205.01068 [cs.CL]
- [26] Jun Zhou, Ke Zhang, Feng Zhu, Qitao Shi, Wenjing Fang, Lin Wang, and Yi Wang. 2023. ElasticDL: A Kubernetes-native Deep Learning Framework with Fault-tolerance and Elastic Scheduling. In *Proceedings of the Sixteenth ACM International Conference on Web Search and Data Mining* (, Singapore, Singapore,) *(WSDM '23)*. Association for Computing Machinery, New York, NY, USA, 1148–1151. https://doi.org/10.1145/3539597.3573037
- [27] Siyuan Zhuang, Zhuohan Li, Danyang Zhuo, Stephanie Wang, Eric Liang, Robert Nishihara, Philipp Moritz, and Ion Stoica. 2021. Hoplite: efficient and fault-tolerant collective communication for task-based distributed systems. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference* (Virtual Event, USA) *(SIGCOMM '21)*. Association for Computing Machinery, New York, NY, USA, 641–656. https://doi.org/10.1145/3452296.3472897