SHIFT: Exploring the Boundary of RDMA Network Fault Tolerance

Paper # XXX, XXX pages

ABSTRACT

GPT *Abstract claim-bounding: the paper later adopts best-effort WQE-level retransmission (e.g., atomics-in-flight and write-once/flag-style protocols may still surface an application-visible error). Consider one sentence that scopes SHIFT's "continuity until next checkpoint" guarantee to the supported traffic/protocol subset to avoid over-claiming.* With gang scheduling in large-scale distributed Large Language Model training, a single network anomaly can propagate and cause complete task failure. The frequency of such anomalies increases with network scale. However, existing fault-tolerance mechanisms, such as checkpointing and runtime resilience methods, primarily operate at the application layer and inevitably cause disruptions in training progress.

We propose to address this challenge by introducing fault tolerance at the Remote Direct Memory Access (RDMA) layer and integrating it with existing application-layer techniques. We present SHIFT, a fault-resilient layer over RDMA that enables seamless redirection of RDMA traffic across different intra-host NICs. By allowing applications to continue execution in the presence of network anomalies until the next checkpoint, SHIFT effectively minimizes training progress loss. SHIFT is designed to be application-agnostic, transparent to applications, and low-overhead.

Through a carefully designed failure state machine and control flow, unmodified applications such as PyTorch with NCCL can run with RDMA-level fault tolerance. Experimental results demonstrate that SHIFT introduces minimal data path overhead while ensuring application continuity under network failures.

1 INTRODUCTION

Large language models (LLMs) [3, 6, 43] continue to scale rapidly. The latest models require tens of thousands of GPUs interconnected via high-speed fabrics such as RDMA networks to enable high-throughput training [5, 17, 49]. With the scaling of training tasks and the increasing adoption of rail-optimized networks, which require more high-failure-rate optical connections, the frequency of network anomalies increases commensurately. Production data from Alibaba indicates that network-related issues account for 15.8% of all production failures [5]. In the context of gang scheduling for distributed training tasks, a single network anomaly can cause the entire task to fail, resulting in wasted computational resources and lost training progress [13, 34].

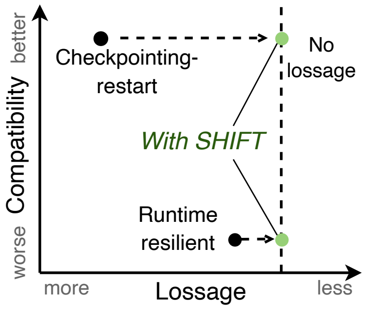


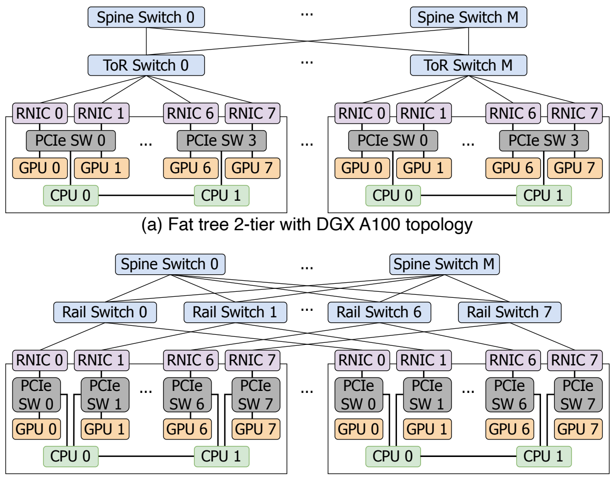
Figure 1: *Comparison of failure-handling technologies for handling network anomalies.* SL *Update*

To mitigate the impact of task failures, prior work has proposed application-layer mechanisms, including checkpointing-restart [7, 13, 17, 26, 45, 46] and runtime resilience strategies [9, 16, 19, 42] (Figure 1). However, relying solely on these mechanisms to handle network anomalies is inefficient. Fundamentally, they serve as *imperfect backup solutions*: checkpointing incurs significant rollback-induced progress loss, while runtime resilience strategies often suffer from limited compatibility and high resource redundancy.

Given the limitations of application-level solutions, network-layer fault tolerance remains a critical direction. The prevailing principle in this domain is "fallback and bypass": fallback to an alternative path and bypass the failure. Failures within the switching fabric are typically handled by in-network switch-level rerouting, while failures at the access layer (e.g., access links and ToR switches) are tolerated through dual-port NICs or dual-ToR designs [25, 34].

This status quo prompts a fundamental question: given that most AI workloads require the standard RDMA Reliable Connection (RC) interface (which guarantees reliable, in-order delivery), commodity RDMA hardware, and zero-copy data paths to remain unchanged, *have we achieved the boundary of RDMA network fault tolerance?* Is it possible to extend the "fallback and bypass" principle to hide even more severe anomalies—potentially even those occurring within the NIC itself—from the application?

We argue that the answer is *partially positive.* In this paper, we extend the principle of "fallback and bypass" to the cross-NIC level. Our objective is to enable a system where, upon the detection of an anomaly on one NIC, ongoing traffic is retransmitted and seamlessly switched to another intra-host NIC. Such a mechanism would tolerate NIC-to-NIC failures



(a) Fat tree 2-tier with DGX A100 topology

(b) Rail-optimized topology with DGX H100 topology

Figure 2: *Illustration of GPU cluster networks with different inter-host and intra-host topologies. Fig (a) depicts a traditional fat-tree network with a DGX A100 architecture [1, 5, 10], while Fig (b) shows a rail-optimized [28] network with a DGX H100 architecture [12, 50]. While specific examples are shown, these inter-host and intra-host topologies can be combined in various ways. For simplicity, Fig (b) omits that a DGX H100 connects four PCIe switches to a single CPU. Even in topologies such as H100, intra-host RNICs remain accessible via the CPUs, although with reduced performance.*

and shield the application from the physical fragility of the endpoints.

However, we identify a fundamental constraint: the commonly used RDMA RC protocol provides a delivery guarantee that is arguably *too strong*. We prove that with current commodity RNICs and the zero-copy data path, seamless cross-NIC fault tolerance is impossible without violating RC's ordering guarantees. Such constraint blocks the support for workloads that rely on strict ordering semantics, such as atomic operations.

Fortunately, we observe that the dominant communication patterns in training workloads often exhibit idempotency or relaxed ordering requirements at the application level, rendering them naturally compatible with cross-NIC fault tolerance. This insight provides the necessary flexibility to circumvent the rigidity of the RC protocol, making it feasible to tolerate NIC-to-NIC RDMA failures for training workloads.

Motivated by this insight, we propose **SHIFT**. SHIFT exploits intra-host NICs as mutual backups. When a network anomaly is detected, SHIFT seamlessly reroutes compatible traffic through a backup NIC on the same host, effectively bypassing the failure. This ensures that transient glitches are hidden from the application and permanent failures can be handled without immediate crash-stop behaviors, minimizing training progress loss. SHIFT remains orthogonal to application-layer solutions and can be integrated to provide a comprehensive defense-in-depth strategy.

Our contributions are summarized as follows:

(1) Through an analysis of common communication libraries, including NCCL [27], NVSHMEM [30], and MSCCL++ [39], we demonstrate that despite inherent limitations, cross-NIC fault tolerance via WR-level retransmission remains highly effective for distributed training workloads. (§3.1)
(2) To fully exploit WR-level retransmission, we propose SHIFT, which introduces three key mechanisms: (1) Completion Queue (CQ)-guided Work Queue (WQ) synchronization; (2) WR execution fences; and (3) CQ event-based 3-way handshake. We implement SHIFT within the RDMA userspace library [36]. (§3.2)
(3) We evaluate SHIFT's bandwidth, latency, and overhead using microbenchmarks and assess its impact on real-world distributed training. Our results confirm that SHIFT delivers robust RDMA resiliency with negligible performance overhead. (§5)

This work does not raise any ethical issues.

2 BACKGROUND AND MOTIVATION

This section provides an overview of large-scale AI training and its failure landscape (§2.1), reviews existing network-layer fault tolerance mechanisms (§2.2), and introduces the basics of RDMA (§2.3).

2.1 Failures in Large-Scale AI Training

Large language models (LLMs) have scaled to unprecedented sizes. For instance, Llama 3 [6, 43], GPT-3 [3], and GPT-4 contain hundreds of billions of parameters. Training these models requires massive clusters with tens of thousands of GPUs [34, 50], typically interconnected by high-performance RDMA networks.

However, the synchronous nature of distributed training renders it inherently fragile. A standard training iteration involves collective communication (e.g., AllReduce) where all workers must synchronize. Consequently, a failure in any single component (be it a GPU, NIC, or cable) stalls the entire training task. This phenomenon is often referred to as the "straggler problem" or results in a complete task crash [13, 34].

To mitigate failure impact, modern training frameworks propose two types of primary mechanisms: **Checkpointing-restart** [7, 26, 29, 45, 46] serves as the standard approach by periodically saving model states to persistent storage. Upon failure, training resumes from the last stored checkpoint.

Runtime resilience techniques [9, 16, 19, 42] offer a more agile alternative. They dynamically replace failed workers or reconfigure the pipeline to exclude faulty nodes, thereby reducing downtime compared to full restarts.

Despite these efforts, these application-layer solutions remain suboptimal. Checkpointing inherently incurs "progress loss": all computation performed since the last checkpoint is discarded upon failure. While increasing checkpoint frequency reduces this loss, it introduces significant I/O and network overhead. Besides, runtime resilience mechanisms often require specific parallelism strategies (e.g., pipeline parallelism) or redundant resources (e.g., hot standbys), which limits their universality and compatibility.

***Takeaway* #1** Distributed training is highly sensitive to hardware anomalies. Existing application-layer fault tolerance mechanisms alleviate the impact but suffer from progress loss, resource overhead, or limited universality.

2.2 Network Anomalies and Fault Tolerance

In modern AI training clusters, network anomalies have emerged as a significant source of hardware failures due to two primary factors:

(1) **Scale:** As cluster sizes expand, the absolute frequency of network anomalies increases commensurately, even with constant per-device failure rates.
(2) **Optical Interconnects:** The adoption of rail-optimized architectures (Figure 2(b)) [28, 34, 41] necessitates optical fibers for NIC-to-switch connections. Optical transceivers exhibit significantly higher failure rates than copper cables.

Production data corroborates this trend: Alibaba [5] attributes **15.8%** of training failures to network issues (e.g., 9.1% NICs, 6.7% optics), while Azure [48] reports that **8.3%** of failures in InfiniBand clusters are network-related. Similarly, Tencent [25] reports that NIC errors account for **15%** of anomalies and other network-related anomalies (switches and fiber) account for **30%** of all anomalies identified in their network. These anomalies manifest as either fatal failures (e.g., device breakdown) or intermittent failures (e.g., interface flapping [17]), both of which stall or disrupt training progress.

Given the limitations of application-layer fault tolerance solutions, enhancing fault tolerance at the network layer is critical. The objective is to *mask network failures* from the application, adhering to the "fallback and bypass" paradigm.

In-network Rerouting. Failures within the network fabric (e.g., switch-to-switch links) are typically handled via in-network rerouting. Data center networks typically provide multiple paths (e.g., ECMP), allowing traffic to seamlessly bypass faulty devices without application intervention.

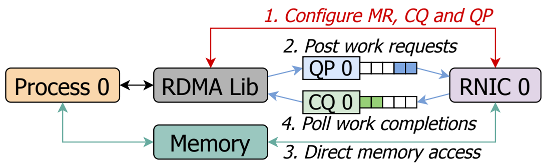


Figure 3: Overview of the RDMA workflow. Red and blue arrows denote control flow, while green arrows represent data flow. Queue Pairs (QPs) and Completion Queues (CQs) are metadata structures residing in host/GPU memory.

Access-Layer Redundancy. Failures at the access layer (between the ToR switch and the RNIC) are also addressed through redundancy: RoCE LAG (Link Aggregation) [38] bonds the two ports of the same NIC into a single logical interface to tolerate single cable or port failures. Dual-ToR architectures [34] further connect a dual-port RNIC to two different ToR switches to tolerate ToR switch failures.

However, these mechanisms face two limitations: (1) They fail to tolerate failures of RNIC itself, which constitute a significant portion of production incidents (15% in Tencent's cluster [25]). (2) They impose extra hardware dependencies (e.g., dual-port NICs) and network configurations, restricting deployment flexibility.

***Takeaway* #2** While in-network rerouting handles in-network failures, existing access-layer solutions (LAG, Dual-ToR) leave RNIC failures unhandled and lack flexibility. This prompts the question: Given the common multi-NIC architecture of modern GPU servers, can we leverage it and achieve *cross-NIC fault tolerance*? We investigate this in §**??**.

2.3 RDMA Basics

We briefly review the basics of RDMA relevant to our design. RDMA serves as the standard for high-throughput, low-latency communication in AI clusters. The RDMA userspace library [36] exposes the verbs API, managing connections via Queue Pairs (QPs). Each QP comprises a Send Queue (SQ) and a Receive Queue (RQ), associated with a Completion Queue (CQ). The typical RDMA workflow is shown in Fig 3.

QPs support multiple transport modes, including Reliable Connection (RC), Unreliable Connection (UC), and Unreliable Datagram (UD) [44]. This work focuses on RC, the predominant transport for training workloads, which guarantees reliable delivery.

RDMA operations fall into two categories: one-sided (e.g., *write*) and two-sided (e.g., *send*, *write with immediate*[1]). To initiate an operation, the CPU posts a Work Request (WR) to the SQ via `post_send`. The driver converts this WR into a Work Queue Element (WQE) in registered memory and rings the RNIC's doorbell register to trigger processing. For

[1] *Write with immediate* transfers data within work requests and consumes a receive work request from the receive queue.

signaled WRs, the application retrieves a Work Completion (WC) from the CQ via `poll_cq` upon completion. Unsignaled WRs generate no notification unless an error occurs.

For two-sided operations, the receiver must pre-post a receive WR to the RQ using `post_recv`. Due to this requisite CPU involvement, two-sided operations incur higher overhead and are typically used for control messages.

In RC transport, the receiver sends an acknowledgment (ACK) only after successfully writing the data to memory. The sender RNIC generates a successful WC upon receiving this ACK [35]. If packet loss persists beyond the retransmission limit, the sender RNIC reports an error WC and transitions the QP to an error state.

Refer to Appendix A for further technical details on RDMA.

3 APPROACH AND OVERVIEW

Basic Approach: *Implicitly RNIC-level fault tolerance.* Rather than propagating network anomalies to the application layer, we propose addressing them at the RNIC level to negate impacts on the applications.

The core insight is that modern GPU servers are typically equipped with multiple RNICs that are connected to GPUs via the PCIe bus using various topologies, such as those shown in Figure 2. It is therefore feasible for intra-host RNICs to *serve as backups for one another*. When network anomalies disrupt RDMA, communication can temporarily *fallback to a backup RNIC on both sides*, which is a **passive switching**. By utilizing the backup RNIC pair, RDMA traffic can bypass network failures. In instances of network interface flapping, RDMA traffic is transitioned back to the default RNIC once it recovers, which is a **active switching**.

Although communication through a backup RNIC may be constrained by PCIe bandwidth and may interfere with other traffic, it remains advantageous to tolerate network anomalies and allow training to continue *until checkpointing or network connectivity recovers* gracefully. Applications also have the flexibility to just complete training within the fallback setting. Combined with existing checkpointing or elastic mechanisms, this approach helps minimize training progress loss and reduce computational waste.

This approach intuitively tolerates network anomalies occurring at the RNIC, access layer links, and other in-network devices. SHIFT cannot tolerate network anomalies if the hardware configuration is insufficient, such as a scenario where all intra-host RNICs are connected to a single switch that has failed.

3.1 Insight: Boundary of Cross-NIC Fault Tolerance

SL *Mellanox's slides, why? Experiments demonstrate problem of overwriting. Understanding the behavior of NIC. - [Note: Experiments in §3.2 demonstrate overwriting behavior; NIC semantics analysis drives best-effort boundary]*

To achieve cross-NIC fault tolerance, an ideal design would seamlessly migrate packet-level RDMA transmission states, such as sequence numbers, retry counters, and send window states, between RNICs. This would allow a backup RNIC to resume transmission precisely from the last failed packet. However, this approach is impractical because RDMA offloads these packet-level states directly to the hardware; consequently, these NIC-managed states are difficult to migrate and often become inaccessible during an access-layer failure.

As described in §??, the Work Request (WR) and Work Completion (WC) serve as the interface between software and RNICs, offering a feasible point for traffic management during failures. Nevertheless, WR-level management is inherently coarser than the ideal packet-level granularity. We therefore investigate how WR/WC-level RDMA traffic retransmission affects RDMA semantics and memory consistency under various failure scenarios.

We consider the general case where multiple packets of multiple WRs are in flight and the sender encounters a WR failure. We assume an ideal design that, upon receiving an error WC, retransmits starting from the first failed WR and continues execution in the original submitted order. The first failed WR is identified by the earliest of either the first failed send WR or the first unfinished receive WR. We then identify the following two cases:

1. If all in-flight WRs are one-sided: The WR/WC interface does not reveal how many packets have already been delivered for an in-flight WR. Therefore, retransmission must start from the first failed send WR, which can overwrite data that was already written by partially completed WRs.
2. If in-flight WRs include two-sided operations: Receive WRs provide receiver-side progress. Retransmission starts from the first failed WR (the earlier of the first failed send WR and the first unfinished receive WR). Any data received after this WR can still be overwritten.

Consequently, even with precise WR-level state knowledge, any data following the last successful send or receive WR will be overwritten. To ensure memory consistency, WR-level retransmission requires that **retransmitted WR data is not accessed before the overwriting process completes**.

We investigated common communication libraries, including NCCL [14, 27], NVSHMEM [30], and MSCCL++ [39], as summarized in Table 1. Our findings indicate:

1. NCCL (Simple), *the most prevalent* protocol, is resilient to data overwriting except for reverse clear-to-send (CTS) messages. We found that bytes of reverse CTS

Library & Protocol	RDMA Ops (Data / Notify)	Resilience to Overwriting
NCCL (Simple) [14]	**Data:** `RDMA Write` **Notify:** `RDMA Write_Imm` **Reverse clear-to-send (CTS):** `RDMA Write`	**Mostly Resilient.** The receiver only accesses the data buffer after retrieving the corresponding WC. Only data received following the final successful WR is overwritten. Reverse CTS message is a WRITE that advertises the remote buffer address, rkeys, and tag information. **SL** *Overwriting may lead to data corruption. Based on message size.*
NCCL (LL/LL128) [14]	**Data + Notify:** `RDMA Write` (Packed: 4B Data + 4B Flag or 120B Data + 8B Flag)	**Vulnerable.** Since both data and flags are transferred via pure writes, the receiver might access data immediately after the flag is written. Overwriting these entries may cause data corruption.
NVSHMEM [30]	**Data:** `RDMA Write` **Notify:** `RDMA Atomic`	**Mostly Resilient.** The receiver does not access the target data buffer until polling the signal variable updated by the atomic operation. If the atomic completes but prior data is overwritten, the receiver might read corrupted values.
MSCCL++ [39]	**Data:** `RDMA Write` **Signal:** `RDMA Atomic`	**Mostly Resilient.** The same as NVSHMEM.

Table 1: Comparison of common communication libraries' protocols in training workloads.

messages account for less than 0.02% of total RDMA traffic during training with the same setting as in §5.3.

(2) NVSHMEM and MSCCL++ utilize atomic operations for notification. To prevent data corruption, WR retransmission must be prohibited if an atomic WR is in flight, as its completion status and the accessibility of preceding data cannot be determined. **SL** *Better to have evidance.*

(3) NCCL (LL/LL128) is vulnerable to data overwriting because both data and notification flags are transferred via pure write operations. However, the LL protocol only utilizes 25–50% of peak bandwidth, which conflicts with high-bandwidth training requirements [10, 34]; LL128 requires specialized hardware capabilities (byte-level ordering). Therefore, these protocols are not widely adopted.

Based on these insights, we propose that the inter-NIC fault tolerance design should employ *best-effort WR-level retransmission.* Inter-NIC fault tolerance is adopted only when retransmitted WRs are not accessed before overwriting completes; otherwise, the network failure is still propagated to the application.

We implement this approach through two mechanisms: (1) returning an application error if an atomic WR is in flight during a failure, and (2) utilizing an unused field in WRs as an *optional* hint for applications to mark WRs that are ineligible for retransmission, returning an error if such marked WRs are in flight during a failure. By default (no hint provided), SHIFT permits retransmission for all non-atomic WRs, which covers NCCL Simple out-of-the-box without library modifications. **SL** *todo*

These insights and mechanisms enable SHIFT to handle the majority of training traffic without compromising memory consistency.

3.2 Design Challenges

To realize WQE-level retransmission, SHIFT must address three technical challenges during both passive and active switching.

C#1: HOW to resubmit under passive switching. Achieving seamless RDMA traffic switching under passive switching requires resubmitting WRs that have been submitted but not yet completed to the backup RNIC. To keep this process application-agnostic, a naive approach is to add a WR buffer to each QP and locally store all submitted but uncompleted WRs. Passive switching can then be achieved by resubmitting WRs from this buffer [21]. However, such buffering introduces CPU overhead and data path latency. Furthermore, the buffer's memory footprint grows linearly with the number of QPs.
Solution: Copying inherent WQEs. As mentioned in §**??**, WRs are converted into WQEs and stored in WQs, which reside in host memory. Therefore, WQEs remain valid in memory despite network failures. SHIFT can thus recover WQEs from the default QP's WQ, modify a few attributes, and resubmit them to the backup QP. See details in §4.3.2.

C#2: WHEN and WHAT to resubmit under passive switching. Two-sided operations require that receive WQEs be posted in the receive queue before the matching send WQEs arrive; otherwise, the send WQEs triggers Receive-not-Ready (RNR). However, this error is observed at the sender. Therefore, when should the receiver resubmit receive WQEs to the backup RNIC, and when can the sender resubmit send WQEs to the backup RNIC? Furthermore, as analyzed in §3.1, SHIFT needs receive-side progress to determine which WQEs should be retransmitted. Together, these requirements necessitate a synchronization mechanism between the sender and receiver QPs.

Solution: CQ event-based 2-way handshake. To record send and receive progress, SHIFT tracks the number of completed send and receive WQEs locally. To minimize CPU overhead, SHIFT leverages RDMA CQ events, which trigger user-thread interrupts upon the arrival of a new WC, to initiate two-sided synchronization.

The synchronization consists of a 2-way handshake between the two sides: (1) Once encountering error, side A resubmits its outstanding receive WQEs to the backup QP (noting that QPs are bidirectional) and then sends a *notification* to side B's backup QP, carrying side A's receive WQE counter and indicating that its receive WQEs are ready. (2) Upon receiving this notification, side B resubmits its receive WQEs to its backup QP and sends an *acknowledgment*, also carrying its receive WQE counter, back to side A's backup QP. Side B then resubmits its send WQEs to the backup QP. (3) After receiving this acknowledgment, side A resubmits its send WQEs to the backup QP.

C#3: HOW to maintain execution order under active switching. RDMA RC guarantees that WRs are executed in the order they are submitted. This is not problematic in passive switching because network anomalies act as hard fences to safely switch traffic. However, maintaining execution order during active switching is challenging because RNICs execute WRs asynchronously with respect to software, which makes it difficult for SHIFT to determine the exact progress of WR execution on the hardware.

Solution: WR execution fence. We propose introducing a software fence for active switching. Specifically, SHIFT continues submitting unsignaled WRs to the backup QP until it submits a signaled WR, which serves as a *fence WR*. After submitting the fence WR, SHIFT directs all subsequent WRs to the default QP but withholds the doorbell update, ensuring that these WRs remain unexecuted. Only after SHIFT receives the WC for the fence WR does it update the doorbell on the default QP to begin execution. This mechanism ensures that the execution order surrounding the fence is strictly preserved.

4 SHIFT DESIGN

4.1 Architecture and Execution Flow

Architecture. Figure 4 depicts the SHIFT architecture. For each QP, SHIFT implicitly creates a backup QP on a backup RNIC and, upon default QP failure, seamlessly switches RDMA traffic to that backup QP without requiring application awareness. To tolerate anomalies that may occur anywhere along the path and to remain compatible with rail-aligned topologies (Figure 2(b)), SHIFT establishes backup QPs on the backup RNICs of both sides.

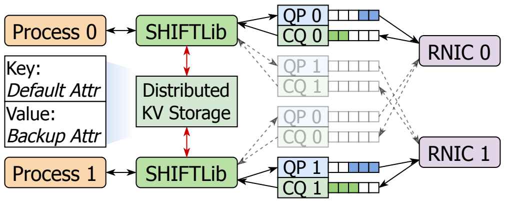


Figure 4: *Architecture of SHIFT. The blue boxes and the green boxes denote the resources managed by the application and SHIFTLib, respectively. The black arrows and the red arrows denote the data flows and the control flows, respectively.*

We implement SHIFT's core functions inside the control and data verbs of the RDMA userspace library [36], called SHIFTLib, while preserving verbs APIs for applications unchanged. Applications can adopt SHIFTLib by switching the RDMA library they use at runtime without rebuilding (e.g., updating `LD_LIBRARY_PATH` on Linux).

As illustrated in Figure 4, SHIFTLib modifies only the implementation of RDMA verbs while preserving the zero-copy data path (i.e., memory-RNIC-RNIC-memory). This design ensures that RDMA retains its low overhead and high performance.

Additionally, SHIFT deploys a distributed KV storage on each host to assist in establishing backup QPs, with communication carried out over the management network.

Execution flow. The logic of SHIFT involves two threads: an application thread that handles RDMA verb calls and a background thread responsible for managing backup RDMA resources and synchronizing two-sided operations. These threads operate across two phases: the RDMA setup phase and the data exchange phase.

RDMA setup phase. During this phase, the application thread initializes RDMA QPs using control verbs and captures these verbs along with their attributes. Simultaneously, the background thread executes the captured verbs to set up backup RDMA resources. (§4.2)

Data exchange phase. Once the RDMA connection is established, the application thread performs RDMA transfers using data verbs as usual, while the background thread waits for CQ events on backup QPs. (§4.3.1)

When a failure is detected on the sender [2], the sender and receiver initiate a 3-way handshake to synchronize and fallback traffic to backup RNIC. (§4.3.2)

[2]In RC mode, the sender ensures data delivery to the receiver RNIC, and network failures manifest as error send work completions.

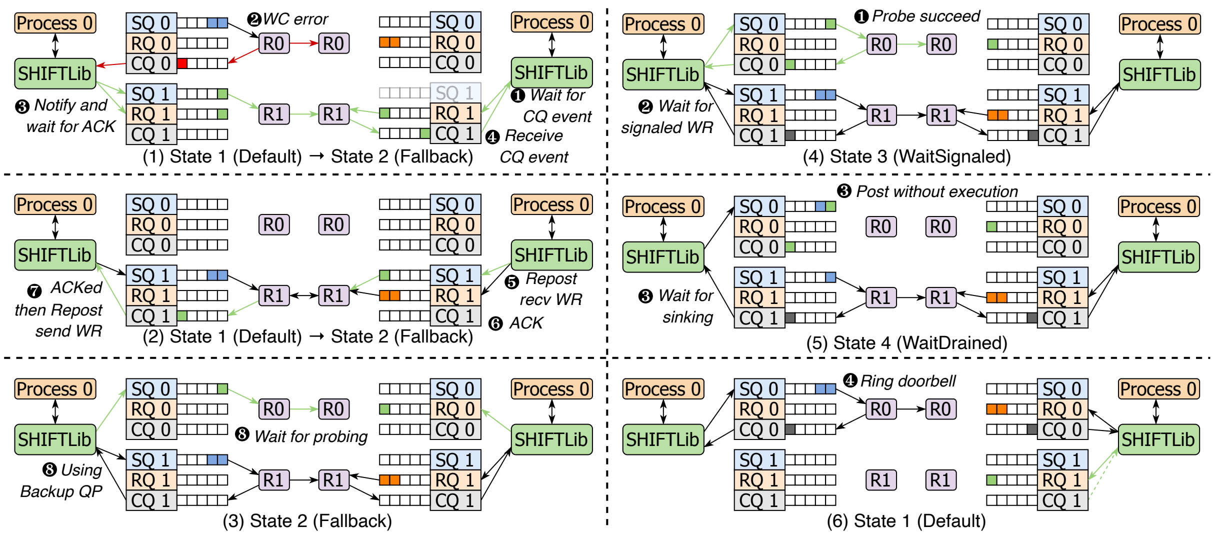


Figure 5: ***Brief overview of the SHIFT state machine. The indices label the steps taken during the state transitions. Green arrows and boxes denote SHIFT control flow and WRs and WCs.***

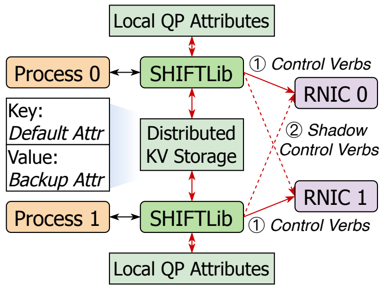


Figure 6: ***The control plane of SHIFT operates as follows: When the application invokes control verbs, SHIFTLib records the verb attributes, storing some locally and others in the KV store. A background thread in SHIFTLib then executes shadow control verbs using these recorded attributes.***

While traffic uses backup RNICs, SHIFTLib periodically posts probe WRs to the default RNIC to assess its recovery status. Once the default RNIC has recovered, SHIFTLib switches the traffic back with the WR execution fence and 3-way handshake as described above. (§4.3.3)

4.2 RDMA Setup Phase

This section describes the RDMA setup phase of SHIFT. SHIFT manages backup RNICs in an application-transparent and low-overhead manner. Using a backup RNIC requires creating the corresponding backup RDMA resources (e.g., CQs and QPs). To avoid interference when RDMA resources are shared across processes, SHIFTLib allocates and tracks these backup resources on a per-process basis.

Shadow control verbs. To manage backup RDMA resources without adding complexity to applications, SHIFT introduces *shadow control verbs*. As illustrated in Figure 6, SHIFTLib implicitly initializes backup RDMA resources by replaying the same control verbs, with the same attributes, that the application used to create default resources. For example, when an application registers a memory region, SHIFTLib also registers the same region on the backup RNIC. Shadow control verbs ensure that backup resources are ready before SHIFT redirects traffic from the default RNIC to the backup RNIC.

To minimize the impact of additional control plane operations on the application, SHIFTLib executes shadow control verbs in background threads. When the application invokes control verbs to set up default RDMA resources, SHIFTLib records these verbs and their attributes. SHIFTLib concurrently runs a background thread that executes the recorded verbs on the backup RNICs. To avoid creating excessive threads, for each process, SHIFTLib assigns one control thread per RNIC to execute the shadow verbs for all QPs associated with that RNIC.

Out-of-band key-value (KV) transfer. An RDMA connection requires an out-of-band channel, typically TCP, to exchange QP route attributes (global ID (GID), QP number (QPN), and local ID (LID)) and memory region attributes (e.g., the remote memory region key (rkey)). Because SHIFT is application-transparent, it cannot assume access to the application's out-of-band channel. SHIFT therefore uses a key-value store over the management network to exchange backup attributes without interfering with the data plane. The KV entries store <*default memory region attributes* → *backup memory region attributes*> and <*default QP route attributes* → *backup QP route attributes*>. Given a peer's default attributes, which is managed by the application, SHIFTLib can query the corresponding backup attributes.

Specifically, after initializing the local backup MR with `ibv_reg_mr` and backup QPs with `ibv_create_qp`, SHIFTLib puts these mappings to the KV store.

Before invoking `ibv_modify_qp` to configure a backup QP, SHIFTLib queries the KV store for the remote backup QP route attributes using the remote default route attributes. SHIFTLib similarly queries remote backup memory region attributes on demand, before their first use, which occurs when SHIFTLib first attempts to send on the backup RNIC. Retrieved attributes are cached locally to avoid redundant queries.

Overhead of backup QPs. Although several previous studies have reported that excessive QPs may cause performance degradation due to on-chip QP context cache misses [18, 40, 47], this issue does not affect SHIFT. This is because the backup QPs are used only during network anomalies and remain idle under normal conditions. As a result, the backup QPs do not compete with the default QPs for on-chip cache and therefore do not introduce additional performance overhead. This is confirmed by the evaluation in §5.2.2.

Moreover, although the backup QPs consume additional memory resources, this overhead is acceptable. When control verbs create a new QP, the QP context requires 3120 Bytes of memory, each SEND WQE requires 256 Bytes, and each RECV WQE requires 16 Bytes. Considering a send queue with 512 entries and a receive queue with 256 entries (the default value of NCCL), the total memory consumption for each backup QP is 138 KB. Similarly, a CQ context requires 592 Bytes, and each CQE requires 64 Bytes. Each backup CQ with 512 entries consumes 33 KB. Therefore, even if SHIFT creates 1k backup QPs on a server, the total memory consumption for backup QPs and CQs is only about 171 MB, which is negligible for modern servers.

Scalability of the key-value store. SHIFT utilizes a cluster-level key-value store to facilitate the initialization of backup RDMA resources. Common key-value stores, such as Redis [37], demonstrate the capacity to scale to thousands of clients and hundreds of thousands of requests per second [8], which far exceeds the requirements for SHIFT. Furthermore, because all interactions with the key-value store occur in a background thread, the latency and scalability of the store do not constitute a bottleneck for SHIFT.

4.3 Data Exchange Phase: State Machine

We design the *SHIFT state machine* to manage RDMA traffic and resources for each default QP and its backup QP. It comprises four send-queue states:

State 1 (Default): The default QP operates normally with no failures currently detected.

State 2 (Fallback): Traffic has switched to the backup QP while SHIFTLib waits for default QP to recover.

State 3 (WaitSignaled): The default QP has recovered and is awaiting the next signaled WR from the application.

State 4 (WaitDrained): SHIFTLib awaits completion of the last signaled WR on the backup QP before switching traffic back to the default QP.

The receive queue has two states, **State 1 (Default)** and **State 2 (Fallback)**, indicating whether the default QP or backup QP handles incoming traffic.

We next describe the data exchange phase by walking through these send-queue states, as illustrated in Figure 5.

4.3.1 Default State.

In State 1 (Default), the system runs normally with no detected failures. The default QP handles all RDMA traffic, while the backup QP remains idle. SHIFTLib maintains counters for the send and receive queues to track the number of completed two-sided send and receive WRs.

SHIFTLib posts a RECV WR to the backup QP and then asynchronously waits for its CQ event in a background thread (used by the handshake described in §3.2).

4.3.2 Fault Tolerance (State 1→State 2).

When network anomalies occur, including fatal failures or interface flapping, RDMA traffic times out and the RNIC places an error WC in the CQ. After polling this WC, SHIFT initiates the RDMA fallback process. A complete fallback process handles traffic in both directions.

Retransmission-safe check. Before resubmitting send WQEs, SHIFTLib first verifies that all outstanding WQEs are safe for retransmission, following the best-effort boundary defined in §3.1. Specifically, SHIFTLib scans outstanding WQEs to detect atomic operations and WQEs with optional ineligibility hints. If any are detected, SHIFT propagates the error to the application instead of attempting fallback. This check ensures that SHIFT only retransmits WQEs that tolerate overwriting.

Work queue resubmitting. SHIFT resubmits WQEs by copying the WQE records in the send and receive work queues.

For send WQEs, SHIFTLib also compares the send and receive WQE counters to infer receive-side progress. If the receive-side progress is ahead, which indicates that the data arrived but the corresponding acknowledgment was lost, SHIFTLib treats the corresponding send WQEs as completed and excludes them from the outstanding set to avoid retransmissions.

SHIFTLib then copies these outstanding WQEs to the backup QP after updating their MR keys, whose mappings between default and backup QPs are stored in the KV store during MR registration, as well as their WQE indices and QP number to match the backup QP configuration. For send WQEs, SHIFTLib rings the doorbell of the backup QP to notify the RNIC of the new entries. For receive WQEs, SHIFTLib updates the receive doorbell record, which maintains the head index of the receive queue.

Overall RDMA fallback procedure. As described in §3.2, SHIFT employs a 2-way handshake to synchronize the sender and receiver. The overall fallback procedure is as follows (also illustrated in Figure 5):

(1) Side A resubmits its outstanding receive WQEs to the backup QP and then sends a `WRITE_WITH_IMM` to side B's backup QP, using side A's receive counter as the immediate value. The receive state on side A transitions to **State 2 (Fallback)**. Side A then waits for the acknowledgment.

(2) Upon receiving this notification, side B receives a CQ event, which triggers the background thread. The background thread resubmits side B's receive WQEs to its backup QP and sends a `WRITE_WITH_IMM`, also carrying side B's receive counter, back to side A's backup QP as the ack. Side B then resubmits its send WQEs to the backup QP. The send and receive states on side B transition to **State 2 (Fallback)**.

(3) After receiving this acknowledgment, side A resubmits its send WQEs to the backup QP. The send state on side A transitions to **State 2 (Fallback)**.

Thereafter, RDMA traffic is handled transparently by the backup QP, maintaining application continuity.

To prepare for potential network recovery, SHIFTLib also resets the default QP. This reset is required to make the error-state QP, set by the RNIC when a WC error occurs, reusable. The QP reset procedure mirrors the initialization performed by the application during setup. SHIFTLib then posts a receive WR to the default QP for the same purpose as in §4.3.1.

4.3.3 Recovery (State 2→State 3→State 4→State 1). Under network interface flapping, the default path may recover intermittently. SHIFT therefore continuously tests for recovery and, when possible, switches traffic back without violating WR ordering.

Probing for default path recovery. To test whether the default path has recovered, SHIFT uses a ping-like probe. SHIFTLib posts a WR that sends a empty write packet through the default path. The probe succeeds only if the WR completes, which implies that both the request and its acknowledgment were delivered. If the probe fails, SHIFTLib resets the default QP and continues probing periodically.

Seamless recovery with ordering. If the probe succeeds, SHIFTLib restores traffic to the default QP. As discussed in §3.2, SHIFT uses a WR execution fence to ensure seamless recovery while preserving the order of WR execution. SHIFT executes this recovery procedure independently in each direction. Specifically, SHIFT switches traffic using the following steps:

(1) Once the probe succeeds, the SHIFT state machine transitions from State 2 (Fallback) to **State 3 (WaitSignaled)**. SHIFT continues posting WRs to, and polling the CQ from, the backup QP until the application posts the next signaled WR. This step is necessary because SHIFT can only rely on WCs, which are reported only for signaled WRs, to determine when all prior WRs have completed. We refer to this next signaled WR as the *fence WR*.

(2) After the fence WR is posted to the backup QP, the state machine transitions to **State 4 (WaitDrained)**. SHIFTLib first posts a notification WR (i.e., a `WRITE_WITH_IMM` WR) to the default QP and enqueues all subsequent application WRs to the default QP. Importantly, SHIFT *withholds the doorbell* of the default QP, which means these WRs will not be executed by the RNIC. This step ensures that the notification is processed before any later WRs after the switch.

(3) After all outstanding WRs on the backup QP have completed (as indicated by polling the corresponding WC of the fence WR), SHIFTLib first rings the doorbell only for the notification WR (the `WRITE_WITH_IMM` WR) on the default QP and waits for its completion. When the receiver observes this `WRITE_WITH_IMM`, a CQ event triggers its background thread, which runs the receive-side procedure described in §4.3.2 to repost outstanding receive WRs. **SL** *Ack omitted here.* SHIFTLib then rings the doorbell for all WRs that were posted but not executed on the default QP. After that, the SHIFT state machine transitions back to **State 1 (Default)**, and subsequent WRs are posted to the default QP directly.

We also include some implementation details in Appendix B.

4.4 Discussion

Relieving straggler issues. In some cases, even without explicit failures, performance can degrade due to grey failures or network contention, creating straggler workers that slow overall training progress [17]. For example, grey failures can cause intermittent, silent packet drops, often triggered by firmware bugs, driver bugs, or temperature-related effects [23]. Because RDMA uses Go-Back-N (GBN) retransmission, each lost packet invalidates subsequent packets, which can substantially reduce throughput.

Although SHIFT is not designed to detect stragglers, it can interoperate with existing straggler detection mechanisms. Once a straggler is detected, and the performance impact outweighs the cost of switching to the backup RNIC, SHIFT can actively switch traffic to the backup RNIC using the recovery mechanism in §4.3.3. This setting is analogous to recovery because the traffic switching occurs while WRs may still be outstanding.

RNICs with different attributes. As described in §4.2, shadow control verbs initialize and manage backup RDMA resources by replaying the same verbs with the same attributes. Consequently, shadow control verbs require the backup RNIC to support the same verbs and attributes as the default RNIC. To the best of our knowledge, RNICs within a server are typically compatible at the verbs level for ease of maintenance. We leave support for heterogeneous RNICs with different attributes to future work.

Hardware constraints of SHIFT. If all RNICs on a server are connected to a single ToR switch, that ToR is a Single Point of Failure (SPOF) and its failure cannot be bypassed by SHIFT. SHIFT cannot overcome such hardware constraints. In contrast, rail-optimized networks [11, 22, 28, 41] connect different RNICs on a server to different ToR switches. This configuration allows SHIFT's RNIC fallback mechanism to bypass ToR failures, improving overall resilience.

KV store unavailability. SL *Need considering.* If the KV store is unreachable during fallback attribute lookup, SHIFT propagates an error to the application rather than risking inconsistent state (fail-closed semantics). This design ensures that SHIFT does not silently degrade to undefined behavior under infrastructure failures.

Multi-tenant scenario. In a multi-tenant environment, SHIFT may require additional security policies, including: (1) all backup RNICs must belong to the same tenant to ensure isolation, and (2) the KV store must be accessible only to that tenant to prevent data leakage. The design for multi-tenancy is beyond the scope of this paper and is left for future work.

Optimization with checkpointing-restart. To mitigate the performance impact of SHIFT fallback, checkpointing-restart mechanisms can be made aware of network anomalies. For example, after fallback completes, the application can checkpoint promptly instead of waiting for the next scheduled checkpoint. This approach reduces future progress loss and limits the time spent under degraded throughput.

SL *1/29 Polished.*

5 EVALUATION

GPT *Please clarify the statement "Due to hardware limitations, the GDR throughput is lower than without GDR." This is counter-intuitive to many readers; specify the exact limitation (PCIe topology? registration/pinning overhead? GPU/driver generation? message sizes?) and whether it affects your main conclusions.*

In this section, we present the evaluation results of SHIFT. We assess SHIFT with microbenchmark tools and a typical AI training application, PyTorch. The evaluation demonstrates the advantages of utilizing SHIFT for handling failures, as well as its low-overhead characteristics.

We have developed SHIFTLib based on rdma-core v48 [36] and implemented the key-value store based on Redis [37] v8.0.5. This key-value store is accessible via the control-plane network.

Bytes	Standard	SHIFT	Standard w/ 500 QP
1 B	2.68 ± 0.39	2.69 ± 0.38	2.69 ± 0.41
2 B	2.70 ± 0.44	2.70 ± 0.38	2.70 ± 0.39
4 B	2.70 ± 0.35	2.69 ± 0.37	2.71 ± 0.41
8 B	2.69 ± 0.40	2.71 ± 0.35	2.70 ± 0.43
16 B	2.69 ± 0.39	2.68 ± 0.38	2.70 ± 0.40

Table 2: ***Average latency (μs) ± standard deviation of ib_write_lat across different message sizes and settings.***

5.1 Failure Injection Methods

To simulate network anomalies within our experiments, we employ the following methods:

(1) **Inject NIC failures**: To emulate RNIC faults, we manipulate the RNIC status through the operating system. Standard Linux network utilities (e.g., `ip link set dev down/up`) are utilized to toggle the RDMA interface state during our experimental evaluations.
(2) **Inject switch/link failures**: To emulate physical link or switch failures, we manipulate switch ports state on the Top-of-Rack (ToR) switch through its management API.
(3) **Inject WQE errors**: While the previous techniques require administrative access, our real-world application experiments (§5.3) are executed on production GPU clusters where such permissions are restricted; consequently, we emulate network anomalies by posting erroneous WRs within these experiments. Specifically, when awaiting a WQE failure injection, SHIFTLib monitors a variable in shared memory; once this variable is set, SHIFTLib manipulates one field of the subsequent WR to an invalid value, which causes both the WR and the entire QP to fail to emulate a network anomaly. Note that this WR is subsequently resubmitted correctly to the backup QP to ensure that failures are not injected into the backup QP.

GPT *Failure-model clarity: the WQE-error method induces a local verbs/QP error (by construction) rather than a "natural" network-side fault (e.g., link down, ToR port down, congestion/packet loss). Please justify why this is a faithful proxy for the specific anomalies you target, and explicitly discuss what behaviors it does* not *capture (e.g., ACK loss, reordering, partial delivery, or receiver-side failure during DMA).*

5.2 Microbenchmarks

We evaluate SHIFT's failure tolerance ability and its overhead by microbenchmarks in this section.

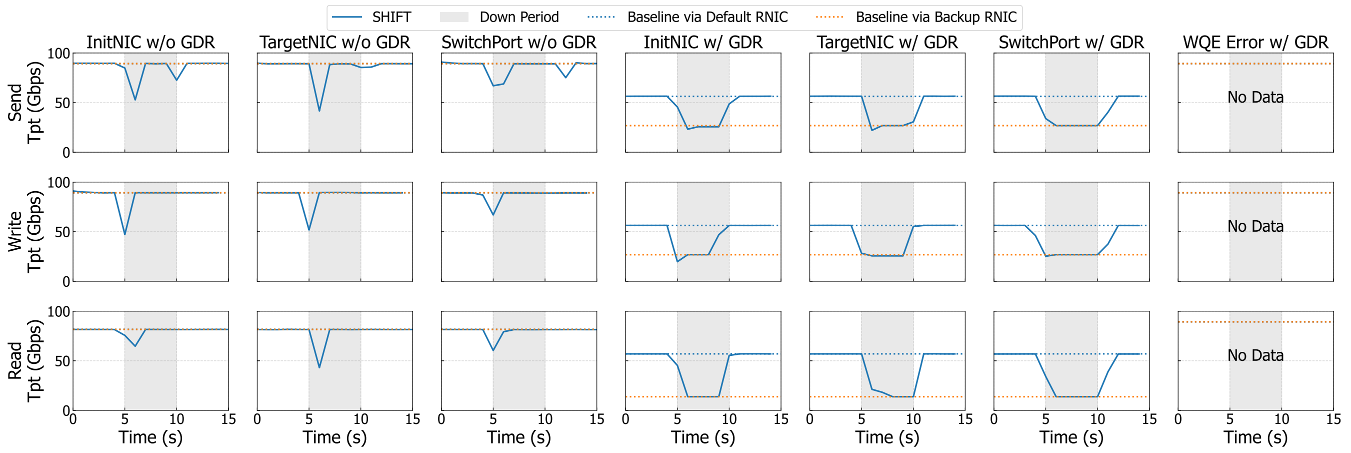


Figure 7: ***Perftest results for ib_send/write/read_bw, with throughput sampled at one-second intervals [33]. The columns correspond to different failure injection methods as described in §5.1, evaluated both with and without GPUDirect RDMA (GDR). For each experimental setup, failures are injected at the 5th second and recovered at the 10th second. Due to hardware limitations, the GDR throughput is lower than without GDR.***

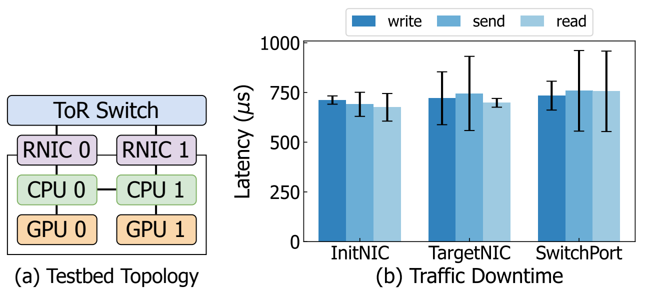


Figure 8: ***(a) Microbenchmark testbed topology. (b) Fallback latency of SHIFT under different failure scenarios.***

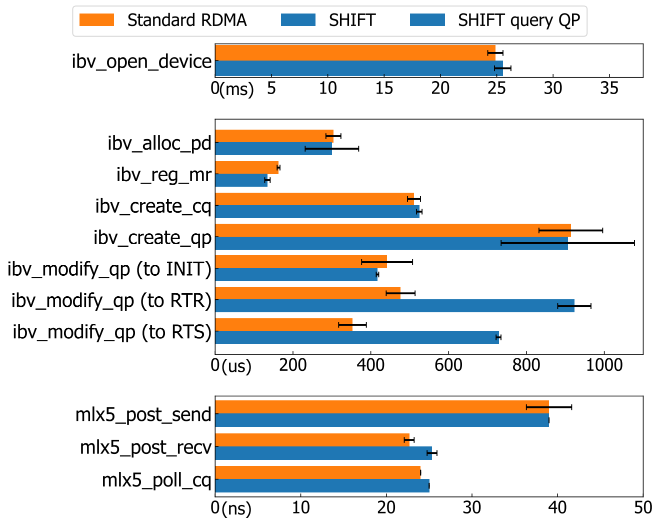


Figure 9: ***Execution time of control verbs and data verbs in standard RDMA and SHIFT. The light orange bars in ibv_modify_qp represent the overhead of ibv_query_qp.***

Our testbed consists of two servers, each equipped with an Intel Xeon Silver 4110 CPU (32 cores, 2.10 GHz) and 128 GB of memory. Each server has two dual-port ConnectX-5 100 Gb RNICs, with port 0 of each RNIC connected to a 100 GbE Ethernet switch using 100 Gb copper cables. The RNICs are configured to operate in RoCEv2 mode. We install MLNX_OFED [31] v5.8 on both servers, and each server is also equipped with two NVIDIA P100 GPUs [32]. The testbed topology is shown in Figure 8(a).

5.2.1 Handling Network Failures. To evaluate the effectiveness of SHIFT in tolerating failures, we conduct microbenchmarks both with and without SHIFT while manually inducing hardware device failures. We use `ib_send_bw`, `ib_write_bw`, and `ib_read_bw` as microbenchmark tools, corresponding to two-sided, one-sided RDMA write, and one-sided RDMA read operations, respectively, which represent the typical RDMA traffic in common AI training.

We inject various failures as detailed in Section 5.1. Figure 7 (a), (b), and (c) illustrate the perftest throughput logs alongside the measured traffic downtime, which spans from the initial polled failed WC to the first successful WC following the RNIC fallback. Because perftest records throughput only per second, the fluctuations in the logged data appear significantly more extensive than the actual downtime.

The results show that: (1) SHIFT does not affect the throughput of one-sided write, read, or two-sided send operations when no anomalies occur; (2) although throughput fluctuates when the default RNIC is disabled, SHIFT sustains RDMA communication by switching traffic to the backup RNIC,

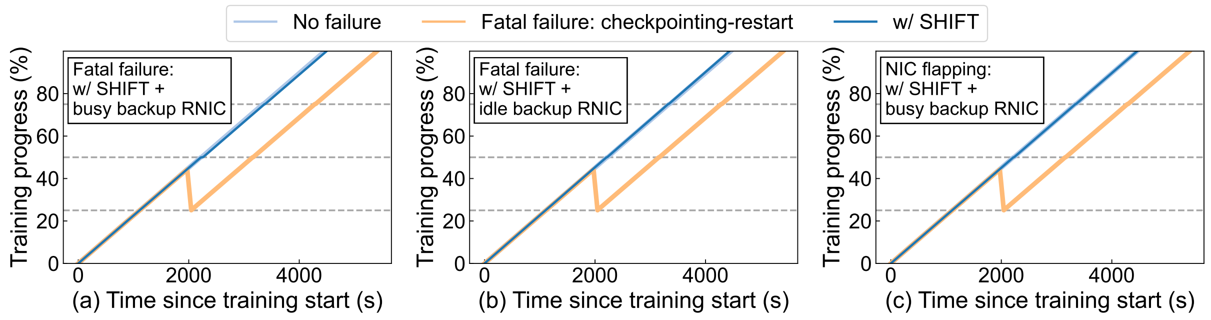


Figure 10: ***The training progress of PyTorch with and without SHIFT under various network conditions. We use the no-failure case as the upper bound and checkpointing-restart without SHIFT as the baseline. The three subfigures illustrate how SHIFT mitigates training progress loss under different network conditions. The dashed lines indicate the training progress of checkpoints.***

whereas standard RDMA just terminates. (3) SHIFT effectively tolerates RNIC failures on either side and restores traffic once the devices are recovered.

The additional throughput fluctuations observed in Figure 7 (b) are likely due to system noise, since send operations rely more on the CPU.

5.2.2 Overhead of RDMA Verbs. To evaluate the performance overhead of SHIFT when *no anomalies occur*, we measure the execution time of RDMA verbs and conduct an end-to-end latency microbenchmark using `ib_write_lat`.

Execution Time of RDMA Verbs. We measure the execution time of control verbs and data verbs when using the microbenchmark tool `ib_send_bw`. For control verbs, we evaluate the execution time of each verb in a single execution. For data verbs, we calculate the average execution time over 1,000,000 iterations for `post_send` and `post_recv`, and over 100,000,000 iterations for `poll_cq`. The execution times of RDMA verbs are shown in Figure 9.

The results show that SHIFT introduces minimal overhead on the execution time of most control verbs, including `ibv_open_device`, `ibv_alloc_pd`, `ibv_reg_mr`, `ibv_create_cq`, `ibv_create_qp`, and `ibv_modify_qp` (to INIT). This indicates that storing control verbs and their attributes as shadow control verbs causes little increase in their execution time.

Except for aforementioned control verbs, two control verbs experience greater overhead: the execution times for `ibv_modify_qp` (to RTR) and `ibv_modify_qp` (to RTS) are slower by 93.65% and 106.34%, respectively, because SHIFTLib queries the attributes of default QPs at these stages, which is essential for resetting default QPs after fallback to backup QPs (recall §4.3.2). The execution time of `ibv_query_qp` is marked in shadow in Figure 9. The overhead of these control verbs is a one-time cost and thus remains acceptable for applications.

The execution time for shadow control verbs is about 35 ms, primarily due to `ibv_open_device` and KV transfer (recall §4.2). This overhead is acceptable because these verbs execute asynchronously without blocking the application and only during initialization.

The results for data verbs indicate that SHIFT introduces minimal overhead for `post_send`, `post_recv`, and `poll_cq` operations, as little additional processing is required when no anomalies occur.

Write Latency. We employ `ib_write_lat`, a microbenchmark tool from the perftest package [33], to measure the end-to-end write latency for message sizes of 1, 2, 4, 8, and 16 bytes in both standard RDMA and SHIFT. The comparison of end-to-end write latency between SHIFT and standard RDMA is shown in Table 2.

The results indicate that SHIFT introduces near zero overhead on data path compared to standard RDMA in end-to-end write latency, with negligible impact on both average latency and standard deviation (Std).

These findings demonstrate that SHIFT incurs almost no overhead in terms of end-to-end RDMA latency when no anomalies occur.

Overhead of Additional QP. As discussed in §4.2, the additional backup QPs do not impact RDMA performance because they do not take up RNIC cache when there is no anomaly. We demonstrate this with a simple experiment: 500 QPs are first created on an RNIC without being used, after which we run the same `ib_write_lat` benchmark as above to measure end-to-end latency. As shown in Table 2, the results show that both the average latency and standard deviation remain nearly identical to those of standard RDMA without additional QPs.

5.3 Real-World Application: PyTorch

In this section, we evaluate SHIFT on PyTorch (v2.0.1) [2], a widely used framework for AI training, with NCCL (v2.19) [27].

In our real-world application experiments, we employ two production GPU servers, each equipped with eight Hopper GPUs and eight ConnectX-7 200 Gb RNICs [4]; Figure 2(b) illustrates the corresponding intra-host topology. We train a GPT-3 13B model [3] using Fully Sharded Data Parallel (FSDP) [51]. The model is trained with a global batch size of 64. Given the limited GPU capacity, we restrict the training process to four epochs and configure the task to generate a checkpoint at every epoch. To emulate network anomalies, we manually inject WQE failures (administrative permissions are restricted within the production cluster) on one server after 2000 seconds of training, which corresponds to approximately 1.78 epochs.

To demonstrate the benefits of SHIFT under network anomalies, we conduct experiments using the same dataset, model, hyperparameters, and failure injection but under different network conditions and fault-tolerance mechanisms:

(1) *No failure.* The training process runs without any network anomalies, and the observed training speed is used as the upper bound.
(2) *Fatal failure: checkpointing-restart.* The training runs without SHIFT. After termination, we restore the RNIC and rerun the training from the last checkpoint, simulating either anomaly resolved or task migrated to another server in production. This case serves as the baseline.
(3) *Fatal failure: w/ SHIFT & busy backup RNIC.* SHIFT is enabled while the backup RNIC is occupied with other traffic (`ib_write_bw` in the experiment). After the next checkpoint completes, the training terminates and resumes from the latest checkpoint.
(4) *Fatal failure: w/ SHIFT & idle backup RNIC.* The backup RNIC is idle, and after fallback to it, the training continues without performance interference.
(5) *NIC flapping: w/ SHIFT & busy backup RNIC.* The same busy backup RNIC as above is used, but the default RNIC is restored after 2 seconds.

The training progress over time under different experiments is shown in Figure 10. In the baseline case, when network anomalies occur, the checkpoint-restart mechanism prevents the training from restarting entirely but still results in substantial progress loss.

Figure 10 (a) shows that with SHIFT, even when the backup RNIC is busy, training continues until the next checkpoint, thereby eliminating progress loss. The training speed exhibits almost no degradation, likely because network bandwidth is not the bottleneck of the training job. After restarting and resuming from the latest checkpoint (with a slight stall observed after the completion of the second epoch in Figure 10 (a)), the training proceeds normally.

Figure 10 (b) demonstrates that if the backup RNIC is idle, training progresses without performance degradation after fallback. Figure 10 (c) illustrates that under interface flapping, SHIFT switches RDMA traffic back to the default RNIC, leaving the end-to-end training process nearly unaffected. SHIFT avoids training to terminate in both cases.

These experiments demonstrate that SHIFT effectively mitigates training progress loss caused by network anomalies, while introducing negligible overhead in their absence.

6 RELATED WORKS

SL *RDMA Migration.* [20]

Previous works on utilizing multiple RNICs. Mellanox [24] introduced the idea of employing multiple RNICs for failure tolerance but did not provide a detailed system design or evaluation. Moreover, their proposal does not support failure recovery for interface flapping.

LubeRDMA [21] was the first to propose a detailed system design for fallback and recovery RDMA traffic across multiple RNICs. However, as a workshop paper, its design has several shortcomings, including significant overhead from control verbs, data verbs, and the WR buffer. In addition, it provides limited support for failure tolerance and recovery, as it does not incorporate CQ event-based notification (§3.2), the state machine (§4.3.3), or background threads (§B).

7 CONCLUSION

GPT *Conclusion over-claims: "thereby eliminating training progress loss" and "requires no application modifications" should be reconciled with the best-effort boundary (§3.2) and the potential need for WR eligibility hints. Please restate as a conditional guarantee (supported protocols/failure model) to avoid credibility loss.* This work presents SHIFT, a fault-resilient layer over RDMA that enables transparent traffic redirection across intra-host NICs and complements application-layer techniques such as checkpointing. SHIFT preserves application continuity under network anomalies until the next checkpoint, thereby eliminating training progress loss. The design requires no application modifications, incurs minimal overhead, and remains agnostic to applications. Experimental results demonstrate that SHIFT sustains high performance while delivering RDMA fault tolerance, making it a practical and effective solution for large-scale distributed LLM training in failure-prone network environments.

REFERENCES

[1] a100 2025. Introduction to the NVIDIA DGX A100 System. https://docs.nvidia.com/dgx/dgxa100-user-guide/introduction-to-dgxa100.html. (2025).

[2] Jason Ansel, Edward Yang, Horace He, Natalia Gimelshein, Animesh Jain, Michael Voznesensky, Bin Bao, Peter Bell, David Berard, Evgeni Burovski, et al. 2024. PyTorch 2: Faster Machine Learning Through Dynamic Python Bytecode Transformation and Graph Compilation. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 929–947. https://doi.org/10.1145/3620665.3640366

[3] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language Models are Few-Shot Learners. (2020). arXiv:cs.CL/2005.14165 https://arxiv.org/abs/2005.14165

[4] cx7 2025. NVIDIA ConnectX-7 Datasheet. https://resources.nvidia.com/en-us-accelerated-networking-resource-library/connectx-7-datasheet. (2025).

[5] Jianbo Dong, Kun Qian, Pengcheng Zhang, Zhilong Zheng, Liang Chen, Fei Feng, Yichi Xu, Yikai Zhu, Gang Lu, Xue Li, Zhihui Ren, Zhicheng Wang, Bin Luo, Peng Zhang, Yang Liu, Yanqing Chen, Yu Guan, Weicheng Wang, Chaojie Yang, Yang Zhang, Man Yuan, Hanyu Zhao, Yong Li, Zihan Zhao, Shan Li, Xianlong Zeng, Zhiping Yao, Binzhang Fu, Ennan Zhai, Wei Lin, Chao Wang, and Dennis Cai. 2025. Evolution of Aegis: Fault Diagnosis for AI Model Training Service in Production. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*. USENIX Association, Philadelphia, PA, 865–881. https://www.usenix.org/conference/nsdi25/presentation/dong

[6] Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. 2024. The Llama 3 Herd of Models. (2024). arXiv:cs.AI/2407.21783 https://arxiv.org/abs/2407.21783

[7] 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

[8] elasticache 2023. Achieve over 500 million requests per second per cluster with Amazon ElastiCache for Redis 7.1. https://aws.amazon.com/blogs/database/achieve-over-500-million-requests-per-second-per-cluster-with-amazon-elasticache-for-redis-7-1/. (2023).

[9] Swapnil Gandhi, Mark Zhao, Athinagoras Skiadopoulos, and Christos Kozyrakis. 2024. ReCycle: Resilient Training of Large DNNs using Pipeline Adaptation. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles (SOSP '24)*. Association for Computing Machinery, New York, NY, USA, 211–228. https://doi.org/10.1145/3694715.3695960

[10] Adithya Gangidi, Rui Miao, Shengbao Zheng, Sai Jayesh Bondu, Guilherme Goes, Hany Morsy, Rohit Puri, Mohammad Riftadi, Ashmitha Jeevaraj Shetty, Jingyi Yang, et al. 2024. Rdma over ethernet for distributed training at meta scale. In *Proceedings of the ACM SIGCOMM 2024 Conference*. 57–70.

[11] Hao Gao and Nikolai Sakharnykh. 2021. Scaling Joins to a Thousand GPUs.. In *ADMS@ VLDB*. 55–64.

[12] h100 2025. Introduction to NVIDIA DGX H100/H200 Systems. https://docs.nvidia.com/dgx/dgxh100-user-guide/introduction-to-dgxh100.html. (2025).

[13] Qinghao Hu, Zhisheng Ye, Zerui Wang, Guoteng Wang, Meng Zhang, Qiaoling Chen, Peng Sun, Dahua Lin, Xiaolin Wang, Yingwei Luo, Yonggang Wen, and Tianwei Zhang. 2024. Characterization of Large Language Model Development in the Datacenter. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. USENIX Association, Santa Clara, CA, 709–729. https://www.usenix.org/conference/nsdi24/presentation/hu

[14] Zhiyi Hu, Siyuan Shen, Tommaso Bonato, Sylvain Jeaugey, Cedell Alexander, Eric Spada, James Dinan, Jeff Hammond, and Torsten Hoefler. 2025. Demystifying NCCL: An in-depth analysis of GPU communication protocols and algorithms. In *2025 IEEE Symposium on High-Performance Interconnects (HOTI)*. IEEE, 48–59.

[15] ibv_post_send 2013. ibv_post_send. https://www.rdmamojo.com/2013/01/26/ibv_post_send/. (2013).

[16] 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 (SOSP '23)*. Association for Computing Machinery, New York, NY, USA, 382–395. https://doi.org/10.1145/3600006.3613152

[17] Ziheng Jiang, Haibin Lin, Yinmin Zhong, Qi Huang, Yangrui Chen, Zhi Zhang, Yanghua Peng, Xiang Li, Cong Xie, Shibiao Nong, Yulu Jia, Sun He, Hongmin Chen, Zhihao Bai, Qi Hou, Shipeng Yan, Ding Zhou, Yiyao Sheng, Zhuo Jiang, Haohan Xu, Haoran Wei, Zhang Zhang, Pengfei Nie, Leqi Zou, Sida Zhao, Liang Xiang, Zherui Liu, Zhe Li, Xiaoying Jia, Jianxi Ye, Xin Jin, and Xin Liu. 2024. MegaScale: Scaling Large Language Model Training to More Than 10,000 GPUs. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. USENIX Association, Santa Clara, CA, 745–760. https://www.usenix.org/conference/nsdi24/presentation/jiang-ziheng

[18] Xinhao Kong, Jingrong Chen, Wei Bai, Yechen Xu, Mahmoud Elhaddad, Shachar Raindel, Jitendra Padhye, Alvin R Lebeck, and Danyang Zhuo. 2023. Understanding {RDMA} microarchitecture resources for performance isolation. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. 31–48.

[19] ChonLam Lao, Minlan Yu, Aditya Akella, Jiamin Cao, Yu Guan, Pengcheng Zhang, Zhilong Zheng, Yichi Xu, Ennan Zhai, Dennis Cai, and Jiaqi Gao. 2025. TrainMover: An Interruption-Resilient and Reliable ML Training Runtime. (2025). arXiv:cs.DC/2412.12636 https://arxiv.org/abs/2412.12636

[20] Xiaoyu Li, Ran Shu, Yongqiang Xiong, and Fengyuan Ren. 2025. Software-based Live Migration for RDMA. In *Proceedings of the ACM SIGCOMM 2025 Conference (SIGCOMM '25)*. Association for Computing Machinery, New York, NY, USA, 99–113. https://doi.org/10.1145/3718958.3750487

[21] Shengkai Lin, Qinwei Yang, Zengyin Yang, Yuchuan Wang, and Shizhen Zhao. 2024. LubeRDMA: A Fail-safe Mechanism of RDMA. In *Proceedings of the 8th Asia-Pacific Workshop on Networking (APNet '24)*. Association for Computing Machinery, New York, NY, USA, 16–22. https://doi.org/10.1145/3663408.3663411

[22] Kefei Liu, Zhuo Jiang, Jiao Zhang, Shixian Guo, Xuan Zhang, Yangyang Bai, Yongbin Dong, Feng Luo, Zhang Zhang, Lei Wang, et al. 2024. R-Pingmesh: A Service-Aware RoCE Network Monitoring and Diagnostic System. In *Proceedings of the ACM SIGCOMM 2024 Conference (ACM SIGCOMM '24)*. Association for Computing Machinery, New York, NY, USA, 554–567. https://doi.org/10.1145/3651890.3672264

[23] Ruiming Lu, Yunchi Lu, Yuxuan Jiang, Guangtao Xue, and Peng Huang. 2025. {One-Size-Fits-None}: Understanding and Enhancing {Slow-Fault} Tolerance in Modern Distributed Systems. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*. 359–378.

[24] mellanoxmprdma 2015. Multi-Path RDMA. https://www.openfabrics.org/downloads/Media/Monterey_2015/Tuesday/tuesday_04.pdf. (2015).

[25] Qingkai Meng, Hao Zheng, Zhenhui Zhang, ChonLam Lao, Chengyuan Huang, Baojia Li, Ziyuan Zhu, Hao Lu, Weizhen Dang, Zitong Lin, Weifeng Zhang, Lingfeng Liu, Yuanyuan Gong, Chunzhi He, Xiaoyuan Hu, Yinben Xia, Xiang Li, Zekun He, Yachen Wang, Xianneng Zou, Kun Yang, Gianni Antichi, Guihai Chen, and Chen Tian. 2025. Astral: A Datacenter Infrastructure for Large Language Model Training at Scale. In *Proceedings of the ACM SIGCOMM 2025 Conference (SIGCOMM '25)*. Association for Computing Machinery, New York, NY, USA, 609–625. https://doi.org/10.1145/3718958.3750521

[26] 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

[27] nccl 2025. NVIDIA Collective Communications Library (NCCL). https://developer.nvidia.com/nccl. (2025).

[28] ncclall2all 2022. Doubling all2all Performance with NVIDIA Collective Communication Library 2.12. https://developer.nvidia.com/blog/doubling-all2all-performance-with-nvidia-collective-communication-library-2-12/. (2022).

[29] 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

[30] nvshmem 2025. NVSHMEM. https://developer.nvidia.com/nvshmem. (2025).

[31] ofed 2024. Linux Drivers. https://network.nvidia.com/products/infiniband-drivers/linux/mlnx_ofed/. (2024).

[32] p100 2016. NVIDIA Tesla P100 GPU Datasheet. https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/tesla-p100/pdf/nvidia-tesla-p100-datasheet.pdf. (2016).

[33] perftest 2024. linux-rdma/perftest. https://github.com/linux-rdma/perftest. (2024).

[34] Kun Qian, Yongqing Xi, Jiamin Cao, Jiaqi Gao, Yichi Xu, Yu Guan, Binzhang Fu, Xuemei Shi, Fangbo Zhu, Rui Miao, Chao Wang, Peng Wang, Pengcheng Zhang, Xianlong Zeng, Eddie Ruan, Zhiping Yao, Ennan Zhai, and Dennis Cai. 2024. Alibaba HPN: A Data Center Network for Large Language Model Training. In *Proceedings of the ACM SIGCOMM 2024 Conference (ACM SIGCOMM '24)*. Association for Computing Machinery, New York, NY, USA, 691–706. https://doi.org/10.1145/3651890.3672265

[35] qptype 2013. Which Queue Pair type to use? https://www.rdmamojo.com/2013/06/01/which-queue-pair-type-to-use/. (2013).

[36] rdma-core 2024. rdma-core. https://github.com/linux-rdma/rdma-core. (2024).

[37] redis 2025. Redis. https://github.com/redis/redis. (2025).

[38] rocelag 2022. How to Configure RoCE over LAG. https://enterprise-support.nvidia.com/s/article/How-to-Configure-RoCE-over-LAG-ConnectX-4-ConnectX-5-ConnectX-6. (2022).

[39] Aashaka Shah, Abhinav Jangda, Binyang Li, Caio Rocha, Changho Hwang, Jithin Jose, Madan Musuvathi, Olli Saarikivi, Peng Cheng, Qinghua Zhou, et al. 2025. MSCCL++: Rethinking GPU Communication Abstractions for Cutting-edge AI Applications. *arXiv preprint arXiv:2504.09014* (2025).

[40] Arjun Singhvi, Aditya Akella, Dan Gibson, Thomas F Wenisch, Monica Wong-Chan, Sean Clark, Milo MK Martin, Moray McLaren, Prashant Chandra, Rob Cauble, et al. 2020. 1rma: Re-envisioning remote memory access for multi-tenant datacenters. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*. 708–721.

[41] superpod 2023. NVIDIA DGX SuperPOD: Next Generation Scalable Infrastructure for AI Leadership. https://docs.nvidia.com/https:/docs.nvidia.com/dgx-superpod-reference-architecture-dgx-h100.pdf. (2023).

[42] 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

[43] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. 2023. Llama 2: Open Foundation and Fine-Tuned Chat Models. (2023). arXiv:cs.CL/2307.09288

[44] transportmodes 2023. Transport Modes. https://docs.nvidia.com/networking/display/rdmaawareprogrammingv17/transport+modes. (2023).

[45] Borui Wan, Mingji Han, Yiyao Sheng, Yanghua Peng, Haibin Lin, Mofan Zhang, Zhichao Lai, Menghan Yu, Junda Zhang, Zuquan Song, Xin Liu, and Chuan Wu. 2025. ByteCheckpoint: A Unified Checkpointing System for Large Foundation Model Development. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*. USENIX Association, Philadelphia, PA, 559–578. https://www.usenix.org/conference/nsdi25/presentation/wan-borui

[46] 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 (SOSP '23)*. Association for Computing Machinery, New York, NY, USA, 364–381. https://doi.org/10.1145/3600006.3613145

[47] Zilong Wang, Layong Luo, Qingsong Ning, Chaoliang Zeng, Wenxue Li, Xinchen Wan, Peng Xie, Tao Feng, Ke Cheng, Xiongfei Geng, et al. 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

[48] Yifan Xiong, Yuting Jiang, Ziyue Yang, Lei Qu, Guoshuai Zhao, Shuguang Liu, Dong Zhong, Boris Pinzur, Jie Zhang, Yang Wang, et al. 2024. Anubis: Towards Reliable Cloud AI Infrastructure via Proactive Validation. *CoRR* (2024).

[49] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, et al. 2022. OPT: Open Pre-trained Transformer Language Models. (2022). arXiv:cs.CL/2205.01068

[50] Chenggang Zhao, Chengqi Deng, Chong Ruan, Damai Dai, Huazuo Gao, Jiashi Li, Liyue Zhang, Panpan Huang, Shangyan Zhou, Shirong Ma, et al. 2025. Insights into deepseek-v3: Scaling challenges and reflections on hardware for ai architectures. In *Proceedings of the 52nd Annual International Symposium on Computer Architecture*. 1731–1745.

[51] Yanli Zhao, Andrew Gu, Rohan Varma, Liang Luo, Chien-Chin Huang, Min Xu, Less Wright, Hamid Shojanazeri, Myle Ott, Sam Shleifer, Alban Desmaison, Can Balioglu, Pritam Damania, Bernard Nguyen, Geeta Chauhan, Yuchen Hao, Ajit Mathews, and Shen Li. 2023. PyTorch FSDP: Experiences on Scaling Fully Sharded Data Parallel. (2023). arXiv:cs.DC/2304.11277 https://arxiv.org/abs/2304.11277

A RDMA INTRODUCTION

RDMA is widely adopted for high-throughput, low-latency communication in large-scale AI training. RDMA enables applications to transfer data between local and remote memory without CPU intervention. As depicted in Figure 3, the RDMA workflow proceeds as follows: the application posts a work request (WR) for data transfer to the RNIC; the RNIC processes the WR and returns the result, known as a work completion (WC), to the completion queue (CQ); finally, the application polls the CQ for transfer results [36].

RDMA typically offers three transport modes: Reliable Connection (RC), Unreliable Connection (UC), and Unreliable Datagram (UD) [44]. Among these, RC resembles TCP, whereas UD is datagram-based like UDP. UC effectively functions as a hybrid, offering unreliable connections similar to UDP but with connected semantics.

To utilize RDMA, the application must allocate resources on both the sender and receiver sides using control verbs. It first opens a device (i.e., RNIC) for communication and obtains a context. Within this context, the application allocates several RDMA resources. First, it creates a Protection Domain (PD) to isolate resources. Then, it registers a Memory Region (MR) accessible by the RNIC, obtaining a local key (lkey) and a remote key (rkey) for subsequent local and remote access. Finally, it creates a Completion Queue (CQ) and a Queue Pair (QP). When creating the QP, the user specifies the local group ID (GID) index, which associates with the source IP and protocol version (e.g., RoCE v1 or v2). Notably, all allocated resources are unique to their contexts and isolated across RNICs.

Once resources are allocated, the application exchanges essential information, including QP details (e.g., QP number) and MR information (e.g., memory address and rkey). This exchange typically occurs via a TCP connection. Subsequently, the application modifies the QP to apply both local configuration and the received remote QP information. Upon modification, the QP transitions to Ready-to-Receive (RTR) or Ready-to-Send (RTS) state, enabling data transfer.

When a QP is ready, the application can initiate RDMA operations using various data verbs. For SEND/RECEIVE (two-sided) operations, the application must first post a receive WR to the receiver's queue and then a send WR to the sender's queue. Conversely, for READ/WRITE (one-sided) operations, receiver participation is unnecessary; the application simply posts a WR specifying the READ/WRITE operation to the send queue.

GPT *Appendix correctness: please be precise about what "ACK" implies and what completion semantics you rely on (receiver RNIC acceptance vs DMA completion vs remote visibility). SHIFT's safety argument depends on these details; vague ACK wording will draw reviewer pushback.* Send WRs, such as SEND or WRITE, trigger actual data transfers. These complete when the local RNIC receives an acknowledgment (ACK) from the remote RNIC confirming data reception [15]. In contrast, receive WRs, such as RECEIVE, wait for incoming data transfers and complete once the data is successfully written to memory.

WR completions (i.e., WCs) are reported to the CQ associated with the QP, where the application retrieves them via polling. In RC mode, the RNIC executes WRs sequentially, ensuring that the completion order strictly matches both the execution and posting orders.

If a WR fails, the QP immediately enters an error state. Consequently, all posted but unexecuted WRs also fail, with their corresponding WCs reported to the CQ. While in this error state, any attempt to post new WRs results in an error. The QP must be transitioned back to the RTS state before processing further WRs.

B IMPLEMENTATION DETAILS

Avoiding deadlock control dependencies. Because SHIFTLib executes all shadow control verbs belonging to the same RNIC within a single background thread, deadlocks may occur when an application uses multiple threads to initialize default QPs. Suppose servers A and B each initialize QP 1 and QP 2 through control verbs in separate threads, and SHIFTLib records the executed verbs in a list as shadow control verbs. The order of these verbs in the list is non-deterministic. Since QP initialization involves creating a QP and then configuring it with the remote QP's attributes, it is evident that configuring a QP depends on the creation of its corresponding remote QP. A potential deadlock may arise during the execution of shadow control verbs in the background thread if server A has only completed creating QP 1 and querying server B QP 1's attributes, while server B has only completed creating QP 2 and querying server A QP 2's attributes. If the shadow control verbs are then executed strictly one by one, cyclic dependencies will emerge.

To break the cyclic dependencies, SHIFTLib executes shadow control verbs in a *best-effort* manner. When executing a shadow control verb, if it depends on a remote attribute that is not ready yet, SHIFTLib skips this verb and tries to execute the next verb in the list. SHIFTLib repeatedly scans the list until all verbs are executed.

WC buffer. When preparing to resubmit outstanding WQEs to the backup QP, SHIFTLib first needs to determine the outstanding WQEs. SHIFTLib polls the default CQ to obtain successful but unpolled WCs, storing them in a local WC buffer [3], and continues until it observes the first error WC. The remaining WQEs in the send and receive queues are then identified as outstanding.

[3]These WCs will later be passed to the application when it polls the CQ

Towards Real-World RDMA Applications

The preceding sections describe the simplest case to illustrate the basic principles of SHIFT. However, real-world RDMA applications often have more complex RDMA usage, such as both sides initializing RDMA or RDMA traffic being initiated intermittently, which complicates the control flow of SHIFT. We now discuss how SHIFT supports these diverse RDMA applications, with the detailed control flow shown in Figure 11.

Most SHIFT logic is designed to run in a background thread rather than the application thread. Executing in a separate thread not only prevents impacting the application thread but also simplifies proper management of the state machine. The background thread is spawned once the backup QP is initialized, as described in §4.3.1.

When the application thread encounters an error WC or fails to post a send, it executes *repost SEND WR*, which also notifies the remote background thread through a CQ event. On the remote side, upon receiving the CQ event, the background thread executes *repost RECV WR*. If both sides are initiating RDMA traffic, they will detect network anomalies independently (since getting an error WC turns the local QP into an error state, causing the remote QP at least gets an RNR error), and both sides then execute the same fallback procedure independently. In contrast, when the send queue is empty, *repost RECV WR* invokes *repost SEND WR*.

After both the send and receive queues of a QP transition to fallback, the background thread continually performs *reset and probe* and *set CQ event* on the default QP at an interval.

When probing succeeds, the application thread switches traffic back according to the state machine described in §4.3.3, proceeding until State 4 (WaitDrained). After SEND WQEs on the backup QP are drained, the application thread executes *repost SEND WR*, which notifies the remote background thread to execute *repost RECV WR*. This procedure mirrors the earlier fallback process but occurs in the opposite direction.

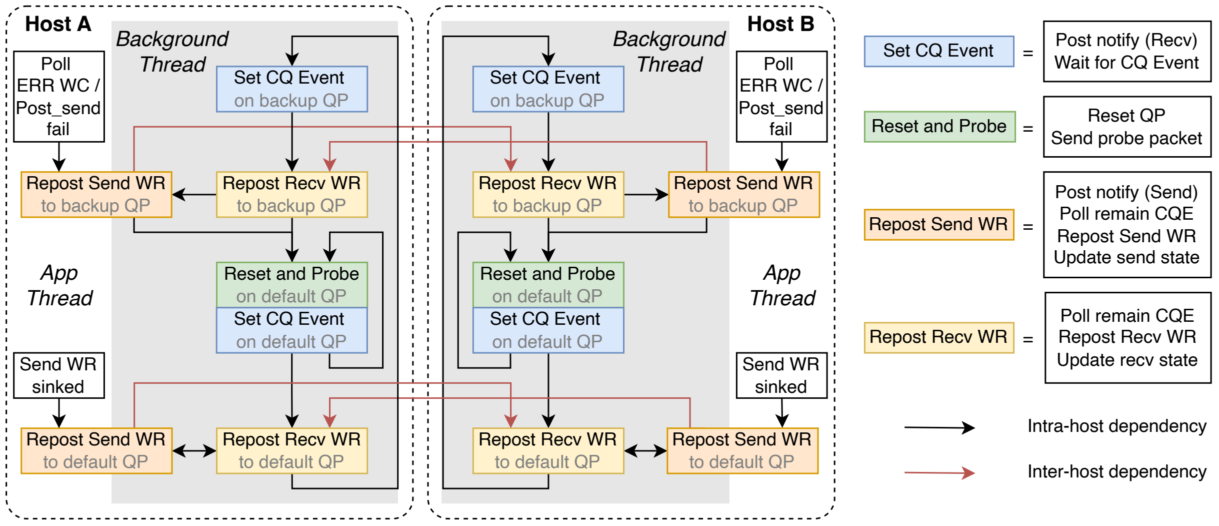

Host A
Background Thread
Poll ERR WC / Post_send fail
Set CQ Event on backup QP
Repost Send WR to backup QP
Repost Recv WR to backup QP
App Thread
Reset and Probe on default QP
Set CQ Event on default QP
Send WR sinked
Repost Send WR to default QP
Repost Recv WR to default QP
Host B
Background Thread
Poll ERR WC / Post_send fail
Set CQ Event on backup QP
Repost Recv WR to backup QP
Repost Send WR to backup QP
App Thread
Reset and Probe on default QP
Set CQ Event on default QP
Send WR sinked
Repost Recv WR to default QP
Repost Send WR to default QP
Set CQ Event = Post notify (Recv) Wait for CQ Event
Reset and Probe = Reset QP Send probe packet
Repost Send WR = Post notify (Send) Poll remain CQE Repost Send WR Update send state
Repost Recv WR = Poll remain CQE Repost Recv WR Update recv state
Intra-host dependency
Inter-host dependency

Figure 11: ***The detailed control flow of SHIFT. To transparently support RDMA traffic fallback and recovery for real-world RDMA applications, SHIFT employs a background thread to execute the state machine for each QP. Black arrows denote intra-host dependencies, whereas red arrows denote inter-host dependencies.***