RESEARCH-ARTICLE

SDR-RDMA: Software-Defined Reliability Architecture for Planetary Scale RDMA Communication

MIKHAIL KHALILOV, Swiss Federal Institute of Technology, Zurich, Zurich, ZH, Switzerland

SIYUAN SHEN, Swiss Federal Institute of Technology, Zurich, Zurich, ZH, Switzerland

MARCIN CHRAPEK, Swiss Federal Institute of Technology, Zurich, Zurich, ZH, Switzerland

TIANCHENG CHEN, Swiss Federal Institute of Technology, Zurich, Zurich, ZH, Switzerland

KENJI NAKANO, Swiss Federal Institute of Technology, Zurich, Zurich, ZH, Switzerland

NICOLA MAZZOLETTI, Swiss National Supercomputing Centre, Lugano, TI, Switzerland

View all

Open Access Support provided by:

NVIDIA

Microsoft Corporation

Swiss National Supercomputing Centre

Swiss Federal Institute of Technology, Zurich

Published: 16 November 2025

Citation in BibTeX format

SC '25: The International Conference for High Performance Computing, Networking, Storage and Analysis
November 16 - 21, 2025
MO, St. Louis, USA

Conference Sponsors:
SIGHPC

https://doi.org/10.1145/3712285.3759902
ISBN: 9798400714665

SDR-RDMA: Software-Defined Reliability Architecture for Planetary Scale RDMA Communication

Mikhail Khalilov
ETH Zürich
Zurich, Switzerland
mikhail.khalilov@inf.ethz.ch

Siyuan Shen
ETH Zürich
Zurich, Switzerland
siyuan.shen@inf.ethz.ch

Marcin Chrapek
ETH Zürich
Zurich, Switzerland
marcin.chrapek@inf.ethz.ch

Tiancheng Chen
ETH Zürich
Zurich, Switzerland
tiancheng.chen@inf.ethz.ch

Kenji Nakano
ETH Zürich
Zurich, Switzerland
kenji.nakano@inf.ethz.ch

Nicola Mazzoletti
Swiss National Supercomputing Centre (CSCS)
Lugano, Switzerland
nicola.mazzoletti@cscs.ch

Peter-Jan Gootzen
NVIDIA Corporation
Zwolle, Netherlands
pgootzen@nvidia.com

Salvatore Di Girolamo
NVIDIA Corporation
Zurich, Switzerland
sdigirolamo@nvidia.com

Rami Nudelman
NVIDIA Corporation
Santa Clara, USA
ramin@nvidia.com

Gil Bloch
NVIDIA Corporation
Yokne'am Illit, Israel
gil@nvidia.com

Jithin Jose
Microsoft Corporation
Redmond, USA
jijos@microsoft.com

Abdul Kabbani
Microsoft Corporation
Redmond, USA
abdulkabbani@microsoft.com

Sreevatsa Anantharamu
Microsoft Corporation
Redmond, USA
anantharamus@microsoft.com

Jie Zhang
Microsoft Corporation
Redmond, USA
zhanj@microsoft.com

Konstantin Taranov
Microsoft Corporation
Zurich, Switzerland
KoTaranov@microsoft.com

Zhuolong Yu
Microsoft Corporation
Redmond, USA
Zhuolongyu@microsoft.com

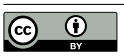
Scott Moe
Microsoft Corporation
Redmond, USA
scottmoe@microsoft.com

Mahmoud Elhaddad
Microsoft Corporation
Redmond, USA
mahmoud.elhaddad@microsoft.com

Torsten Hoefler
ETH Zürich
Zurich, Switzerland
torsten.hoefler@inf.ethz.ch

Abstract

RDMA is vital for efficient distributed training across datacenters, but millisecond-scale latencies complicate the design of its reliability layer. We show that depending on long-haul link characteristics, such as drop rate, distance and bandwidth, the widely used Selective Repeat algorithm can be inefficient, warranting alternatives like Erasure Coding. To enable such alternatives on existing hardware, we propose SDR-RDMA, a software-defined reliability stack for RDMA. Its core is a lightweight SDR SDK that extends standard point-to-point RDMA semantics — fundamental to AI networking stacks — with a receive buffer bitmap. SDR bitmap enables partial message completion to let applications implement custom reliability schemes tailored to specific deployments, while preserving zero-copy RDMA benefits. By offloading the SDR backend to NVIDIA's Data Path Accelerator (DPA), we achieve line-rate performance, enabling efficient inter-datacenter communication and advancing reliability innovation for inter-datacenter training.



SC '25, St Louis, MO, USA

ACM ISBN 979-8-4007-1466-5/25/11
https://doi.org/10.1145/3712285.3759902

CCS Concepts

• **Networks** → **Data center networks; Transport protocols**.

Keywords

long-haul, reliability, RDMA, offloading, inter-datacenter training

ACM Reference Format:
Mikhail Khalilov, Siyuan Shen, Marcin Chrapek, Tiancheng Chen, Kenji Nakano, Nicola Mazzoletti, Peter-Jan Gootzen, Salvatore Di Girolamo, Rami Nudelman, Gil Bloch, Jithin Jose, Abdul Kabbani, Sreevatsa Anantharamu, Jie Zhang, Konstantin Taranov, Zhuolong Yu, Scott Moe, Mahmoud Elhaddad, and Torsten Hoefler. 2025. SDR-RDMA: Software-Defined Reliability Architecture for Planetary Scale RDMA Communication. In *The International Conference for High Performance Computing, Networking, Storage and Analysis (SC '25), November 16–21, 2025, St Louis, MO, USA.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3712285.3759902

1 Motivation

> "We had to scale to more compute, and that compute was not available as part of one cluster. We had to go to multi-cluster training..." — *from OpenAI's "Pre-Training GPT-4.5"* https://www.youtube.com/watch?v=6nJZopACRuQ

The scale of a single AI datacenter is constrained by its power plant supply capacity [10, 30, 40]. As demand for training resources grows, hyperscalers are exploring strategies to utilize the compute capacity of multiple datacenters within a single pre-training job [13, 41, 43]. This necessitates dedicated communication channels between datacenters spanning thousands of kilometers, potentially between continents, such as long-haul black fiber and submarine cables [4]. Over such distances, a GPU networking stack (e.g., xCCL or MPI [15, 35, 45]), tailored for commodity RDMA Network Interface Cards (NICs), must enable efficient, reliable messaging over lossy connections with millisecond-scale round-trip times (RTTs).

Reliability schemes in commodity RDMA NICs (e.g., ConnectX-7/8) are limited to retransmission-based schemes, such as Go-Back-N and Selective Repeat (SR), implemented in the ASIC to support multi-hundred-gigabit bandwidths [17]. In Section 2, we illustrate that constraining reliability to retransmission-based schemes can be inefficient for serving traffic through a high-delay, lossy cross-datacenter channel. Our key observation for SR is that message completion time can accumulate multiple RTTs due to retransmission timeouts. SR extensions, such as negative acknowledgment, can minimize the impact of retransmission timeouts in the average case, but not at the tail [12]. A viable alternative could be to avoid relying on acknowledgments and timeouts, and instead leverage erasure coding of application buffers at the transport layer [50, 53].

However, as long as reliability remains part of the NIC ASIC, realizing alternative algorithms in production deployments would take years to materialize, since datacenter operators would need to wait for next-generation silicon to become available [49]. A possible solution could be prototyping on FPGAs, but as community experience suggests, FPGAs are generally hard to program and are not supported in commodity datacenter NICs. Furthermore, existing FPGA-based prototypes have not been evaluated for upcoming Tbit/s links [29, 46, 53].

We solve these problems with a software-defined solution illustrated in Figure 1. Its bottom software layer — the software-defined reliability (SDR) middleware SDK — decouples the low-level details

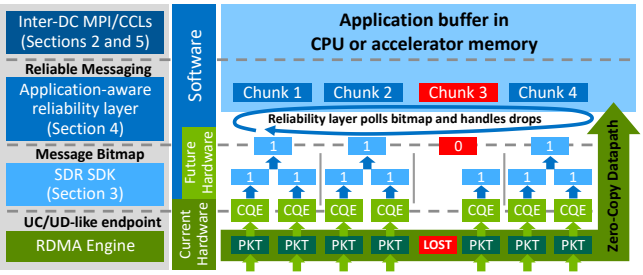


Figure 1: RDMA stack based on the software-defined reliability (SDR) architecture. In all figures throughout the paper, green corresponds to the hardware-related parts of the SDR stack and blue encodes software components.

of the packet progress engine from the upper-layer reliability logic. SDR achieves this through novel *partial message completion* API semantics on the receive side. Partial message completion conveys to the reliability algorithm information about message chunks that were dropped in transit, represented as a bitmap.

The key design challenge for SDR is sustaining packet processing at > 100 Gbit/s with minimal overhead on the CPU and system memory bandwidth. The SDR progress engine uses offloading features in the BlueField-3 SuperNIC to achieve high throughput while exposing a bitmap API to the reliability layer [31]. Namely, SDR utilizes hardware-based unreliable RDMA Write for data movement offloading and offloads the packetization and bitmap-updating logic to the Data Path Accelerator (DPA) [8, 34]. These design decisions allow SDR to sustain packet rates on links of up to 3.2 Tbit/s.

Building on the SDR bitmap, we design various SR- and EC-based reliability schemes. We show that the guided choice and performance tuning of an optimal reliability algorithm can improve average and 99.9th percentile RDMA Write completion time by up to 5× and 12×, respectively. This is especially critical for cross-datacenter AI collectives, where multi-stage execution causes reliability overheads to compound and degrade end-to-end performance.

Our main contributions are:

(1) Analysis of inter-datacenter communication challenges.
(2) SDR-RDMA architecture that decouples application-specific reliability logic from low-level packet processing.
(3) SDR-RDMA data path offloading for full line-rate performance on current and next-generation commodity NICs.
(4) Framework to simulate and analyze the performance of SDR-based reliability algorithms in an inter-datacenter setup.

2 Challenges of inter-DC communication

We show that the NIC reliability algorithm plays a critical role in the performance of point-to-point and collective long-haul communication. We identify two hard requirements for the inter-datacenter networking stack: freedom in the choice of reliability protocol and support for these protocols at line rate in commodity RDMA NICs. In our work, we achieve **both** with a software-defined solution at the endpoint.

2.1 There is *no* ideal approach to reliability

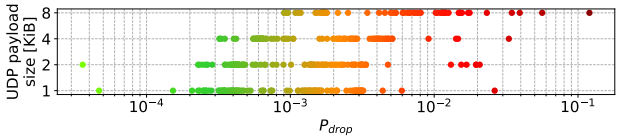


Figure 2: Packet drop rate measured with iperf3 between 16 UDP flows located in Lugano and Lausanne CSCS datacenter sites approximately 350 km apart, connected via a 100 Gbit/s channel. Copper cables are used within both datacenter sites. An optical connection provided by a local ISP is used to connect the two datacenters. For each payload size, drop rates are measured over 200 trials of 15 seconds each, conducted over a 3-day period.

In the inter-datacenter scenario, distances, drop rates, and message sizes are not only 2–3 orders of magnitude different from the intra-datacenter case, but can also vary multiple times across deployments. For example, if we consider cross-datacenter setups from Livermore to Oak Ridge and from Lugano to Kajaani, we expect differences of at least 1000 km in total cable lengths, corresponding to approximately 6.5 ms of added RTT due to geographical features and the proximity of service infrastructure[1].

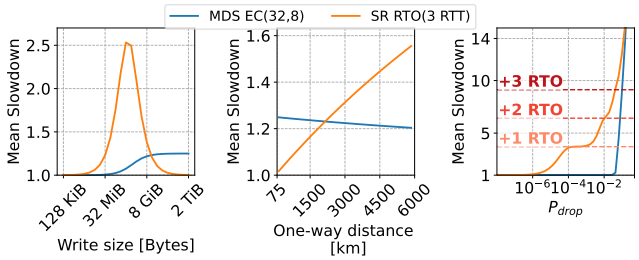


Figure 3: Impact of reliability on message time at 400 Gbit/s.
Left (a): 3750 km = 25 ms RTT, $P_{drop} = 10^{-5}$
Middle (b): 8 GiB message, $P_{drop} = 10^{-5}$
Right (c): 128 MiB message, 3750 km

Bandwidth and drop rates in inter-datacenter links can vary widely due to factors like cable technology, budget, and QoS policies. Figure 2 shows UDP drop rates between two datacenters connected via an optical link from a publicly funded ISP. On both sides, traffic was isolated from other intra-datacenter traffic, and trials with endpoint-side drops — tracked through Linux kernel counters — were excluded. We observe up to three orders of magnitude variation in drop rates across trials for the same packet size, with drop rates increasing for larger packets: from 10^{-4} to 10^{-2} for 1KiB, and from 10^{-3} to over 10^{-1} for 8 KiB packet sizes, respectively. Such variability and its correlation with packet size suggest significant switch buffer congestion on the ISP side. In contrast, data collected across 15 Microsoft datacenters show that pre-Ethernet-FEC drop rates on private optical networks can be as low as 10^{-8} [52].

Figure 17 shows the impact of long-haul channel parameters on reliability performance at 400 Gbit/s. We compare two schemes (see Section 4): a standard Selective Repeat (SR) protocol and an Erasure Coding (EC) scheme. Unlike SR, EC does not rely on retransmissions but instead sends parity data with the original message. While EC is not supported by commodity NICs, recent FPGA-based results at 10 Gbit/s[53] suggest that it could reduce retransmission overhead in high-delay, lossy channels.

We observe that there is no single "best" reliability scheme. In Figure 17a, SR reaches peak slowdown at the point when at least one packet in the message is likely to be dropped, given a drop rate of 10^{-5}. With SR, the retransmission cost for this drop cannot be hidden in the pipeline — each retransmission significantly contributes to the message completion time. In contrast, the EC-based scheme remains close to ideal performance because the receiver can recover from drops in-place using parity data. Above 2^{35} = 32 GiB, the message becomes large enough for packet injection time to dominate over channel delay. For such a "large" message, while SR can efficiently hide the latency of retransmissions behind the time required to inject the message, the EC-based scheme consumes a portion of the channel bandwidth to send parity data.

In Figure 17c, with a "small" 128 MiB message at drop rates above 10^{-4}, we observe an increase in completion time from 3× to 10× due to a single packet requiring multiple retransmission rounds. In Figure17b, as the cable distance between datacenters increases, an 8 GiB message that was previously considered "large" (dominated by the injection time) becomes "small" (dominated by RTT delay). In the "large" interval, SR outperforms EC, whereas in the "small" interval, the trend is reversed.

The above result suggests that supporting various reliability schemes in the NIC is essential to enable efficient communication between two datacenters. Furthermore, when considering a scenario with more than two datacenters, a single endpoint might communicate with remote endpoints at varying distances. Achieving optimal message completion times in this scenario may require per-connection reliability protocol provisioning.

2.2 Need for a software-defined solution

From the bottom-level protocol perspective, current generation NICs (e.g., RoCEv2) can serve inter-datacenter traffic out-of-the-box. Their physical, link, and network layers are compatible with Ethernet and Internet protocols and can operate in lossy mode [48]. The custom RDMA transport logic (e.g., congestion control and reliability) is implemented above the commodity protocols and runs at the endpoints (e.g., QPs).

We begin by analyzing the Reliable Connection (RC) Verbs transport, which provides the reliable Write primitive used by distributed training libraries [15, 35, 45]. To our knowledge, current RDMA NICs from Broadcom, Intel, Microsoft, and NVIDIA support only retransmission-based RC protocols, typically implemented in the NIC's ASIC for performance [17]. Since ASIC development cycles span 3-4 years, adopting and optimizing new reliability protocols, such as erasure coding (EC), would take years to materialize.

[1]We consider paths along the public roads traced by Google Maps between locations of the largest non-private supercomputers in the US (El Capitan, Frontier) and Europe (Alps, Switzerland and LUMI, Finland) from the November 2024 Top 500 list.

Subset	API call	Description
Data path setup	ctx *context_create(char *dev_name, dev_attr *dev_attr);	Allocate HW resources (CQs, DPA threads) shared by QPs.
	qp *qp_create(ctx *ctx, qp_attr *qp_attr);	Create a queue pair within a context.
	int qp_info_get(qp *qp, void *info);	Retrieve QP info for out-of-band exchange.
	int qp_connect(qp *qp, void *remote_qp_info);	Establish connection between queue pairs using QP info.
Memory	mr *mr_reg(ctx *ctx, void *addr, size length, flags flags);	Register memory for send/receive via QPs in the context.
Send	int send_stream_start(qp *qp, start_wr *wr, snd_handle **hdl);	Create streaming send (w. Imm) message context.
	int send_stream_continue(snd_handle *hdl, continue_wr *wr);	Send new chunk(s) into a remote buffer (stream) by an offset.
	int send_stream_end(snd_handle *hdl);	Indicate that no new chunks will be added to the stream.
	int send_post(qp *qp, snd_wr *wr, snd_handle **hdl);	Initiate a one-shot send (w. Imm) message.
	int send_poll(snd_handle *hdl);	Poll for a send message completion.
Receive	int recv_post(qp *qp, rcv_wr *wr, rcv_handle **hdl);	Post a receive message buffer.
	int recv_bitmap_get(rcv_handle *hdl, uint8 **bitmap, size *len);	Get pointer to the bitmap associated with a receive buffer.
	int recv_imm_get(rcv_handle *hdl, uint32 *immediate);	Retrieve immediate data if it is ready.
	int recv_complete(rcv_handle *hdl);	Marks a receive message as complete.

Table 1: SDR API overview. Object metadata (e.g., Write size and offset) is encapsulated into the C structs.

We believe a software-defined approach to RDMA reliability is the right path forward. Just as QUIC enabled rapid innovation in transport protocols on top of UDP [22, 27], a software layer over unreliable RDMA transports can foster fast-paced development, adoption, and optimization of application-aware reliability protocols tailored for datacenter environments.

2.3 Transport design challenges

Existing RDMA implementations offer support for unreliable transports, similar to how the operating system provides UDP service to QUIC. For example, the Verbs API supports Unreliable Datagram (UD) and Unreliable Connected (UC) [19]. Libfabric API endpoints could also support datagram service (FI_EP_DGRAM) with messaging and remote memory access operations [36]. Without loss of generality, we focus our discussion on Verbs:

- *UD* offers a two-sided per-packet service. Emulation of reliable Write semantics on top of UD transport is feasible. However, due to the possibility of out-of-order packets (e.g., because of drops) it comes at the cost of intermediate packet staging in the host CPU or NIC memory on the receive side [21, 23].
- *UC* offers unreliable multi-packet Writes. With UC, out-of-order packets are not problematic, because the sender side determines the target memory (address) of Write. If at least one packet within the UC message is dropped, the whole message will be dropped.

UC is ideal for building zero-copy transport, but its coarse completion semantics are unsuitable for reliability layers. For instance, if one 4 KiB packet is lost in a 1 GiB Write, the NIC considers the entire Write lost-forcing the application to retransmit all 1 GiB, wasting time and bandwidth.

To address this, we introduce a lightweight middleware between UC and the reliability layer that delivers messages in *chunks*, each aligned with the MTU. A *bitmap* tracks received chunks, allowing the reliability layer to process available data. EC can use this to identify and repair losses with parity, while SR uses it to report dropped chunks to the sender.

3 SDR middleware

Our goal is to enable innovation of reliability algorithms in current generation RDMA NICs. We achieve this goal with the software-defined reliability (SDR) SDK, a middleware that extends conventional RDMA completion semantics to support unreliable arbitrary-length messaging with a partial completion bitmap. The bitmap can be used by the reliability layer to locate drops within a message.

3.1 Partial message completion API

Table 1 presents the SDR API. We discuss its novel features.

3.1.1 Partial completion bitmap. A key feature of SDR is a partial completion bitmap enabled on top of standard unreliable RDMA transports (e.g., unreliable Write). The bitmap is a lightweight software abstraction that decouples the reliability protocol logic running above it from the RDMA progress engine running below it. SDR users (e.g., a reliability layer) can post a receive buffer and track chunks that have been received by polling the bits in the associated bitmap, while packet (de)fragmentation progress is offloaded to the NIC (handled by the SDR runtime behind the scenes).

A single bit in the bitmap corresponds to a message **chunk**-a contiguous block of bytes within a receive buffer. Chunk size is a multiple of the network Maximum Transfer Unit (MTU) and is configurable by the user. It can be tailored to the specific needs of the application running on top of SDR. For example, the bitmap resolution can be chosen to mask drop bursts within the same chunk; with a chunk size of 16 packets, dropping 7 packets inside a chunk would appear to the upper layer as a single chunk drop.

3.1.2 Sender-side optimizations. SDR supports two send types:

- *Streaming* send offers fine-grained control: new chunk(s) can target any offset in the remote buffer and are added to the send stream. For reliability, a typical use case is retransmission — for example, resending a chunk after a timeout.
- *One-shot* send prioritizes efficiency when transmitting large contiguous data blocks in a single operation. Once all chunks are injected, the message context is destroyed.

Both APIs are asynchronous to enable overlap between computation and network injection. By providing two distinct primitives we allow the sender to select the appropriate granularity, while the SDR backend can optimize the two send paths independently.

3.1.3 Order-based message matching. SDR is a message-based API: on the receiver side, there is an association between the receive buffer and a bitmap state, while the streaming API assumes that new chunks are added to the same stream queue associated with a fixed remote buffer. In SDR, message matching between sender and receiver is *order-based*: the sender's send messages "land" in the receiver's buffers in the order they were posted.

For example, let's assume that the receiver posts two receive buffers in sequence: Recv1, Recv2. When the sender posts the send messages (e.g., Send1, Send2), the order-based matching ensures that Send1 targets Recv1, and Send2 targets Recv2. Because SDR matching is order-based, explicit buffer metadata (e.g., remote memory key, virtual address, etc.) does not need to be exchanged between receiver and sender, apart from ensuring that the receive is posted before the corresponding send is issued.

3.2 Messaging protocol

We illustrate the internal SDR messaging protocol with the client-server example in Figure 4.

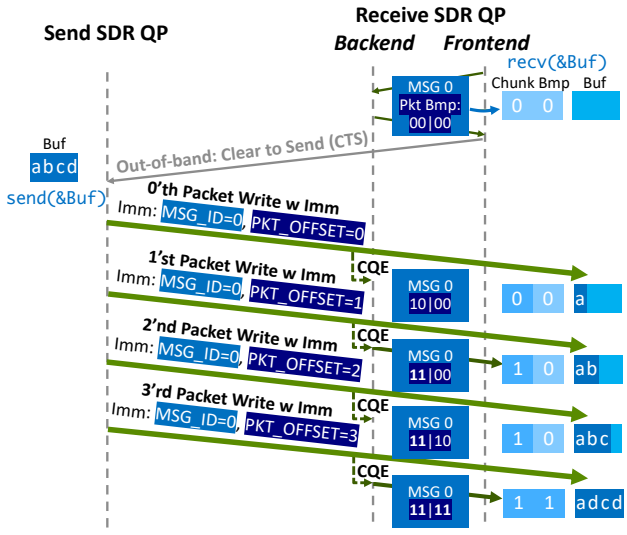


Figure 4: Example of SDR one-shot send data path with four packets, two packets per receive bitmap chunk. The protocol backend can be offloaded to the NIC.

3.2.1 Backend/frontend design. In the backend, SDR relies on UC as a zero-copy engine for chunk delivery (UD can still be used with the limitations outlined in Section 2.3).

While sending each chunk with a single RDMA write-with-immediate over UC would be the simplest solution, this could result in the entire chunks being dropped if packets are reordered by ISPs.

This limitation arises from the behavior of the receiver's UC QP when handling multi-packet messages. Specifically, the UC QP maintains an expected packet sequence number (ePSN) [19] that is incremented with each received packet and resets at the start of every new message. If an incoming packet PSN does not match the ePSN, the entire message is dropped due to PSN mismatch.

To overcome this limitation, we instead issue one RDMA Write-with-immediate operation per packet, making each packet being handled as a single message. While this strategy increases the backend processing load, as it requires tracking each individual packet separately, it enables SDR to handle out-of-order packet delivery.

To efficiently handle per-packet tracking and transport-level queue management without increasing CPU load, we design the internal SDR messaging protocol with offloading in mind. The backend maintains a per-packet bitmap for each message, which is *coalesced* into a frontend chunk bitmap. A chunk is only signaled when all its packets arrive. Through this decoupling, we enable backend offloading to the Data Path Accelerator (DPA), a programmable engine in the BlueField-3 SuperNIC designed for parallel traffic processing [8, 31, 34].

3.2.2 QP creation and connection establishment. During QP setup, send and receive backends populate message tables and associate them with each other via an indirect memory key exchanged out-of-band (Figure 5). The runtime allocates internal buffers for per-packet (backend) and chunk (frontend) bitmaps, based on the user-defined maximum message size and bitmap chunk size (4-packet messages with 2-packet chunks in Figure 4).

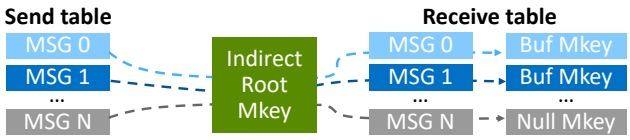


Figure 5: Sender and receiver SDR QPs share a zero-based root memory key, enabling offset-based addressing of receive buffers. For a QP with maximum message size M, message i targets the offset range $[i \cdot M, i \cdot M + M)$.

3.2.3 Posting buffers. During the post-receive call, the backend allocates a context slot in the message table, updates the indirect root memory key table with the user buffer's key, and sets up per-packet and chunk bitmaps. After posting the receive buffer, the receiver sends an out-of-band clear-to-send (CTS) signal. The sender can then begin writing new data into the buffer.

3.2.4 Data path. The SDR send operation results in a sequence of single-packet unreliable Writes — four Writes in our example target the receiver's root indirect memory key, where the offsets 0, MTU, $2MTU$, and $3MTU$ are backed by the receive buffer.

The NIC's RDMA engine writes the UC packet payload directly into user buffer. The receive backend then obtains a completion (CQE or cookie) for the packet. Each CQE includes 32-bit *transport immediate data*, divided into three fields:

(1) 10 bits for the message ID (light blue in *Imm* of Figure 4), allowing up to 1024 in-flight message descriptors per QP.
(2) 18 bits for the packet offset (dark blue in *Imm* of Figure 4), supporting up to 1 GiB message size with a 4 KiB MTU.

(3) 4 bits for user immediate reconstruction (not shown); for messages with user immediate, the sender backend samples fragments of it into this field.

This 10 + 18 + 4 bit split reflects our use-case needs. Alternative splits, such as 8 + 22 + 2, can be used to support larger messages.

Using the immediate fields, the receive backend locates the message descriptor and computes the bitmap offset for the packet. In Figure 4, a packet with offset zero sets the first bit in the packet bitmap; the second packet sets the second bit. Once all bits for the first buffer chunk are set, the backend updates the corresponding bit in the frontend's chunk bitmap. The same procedure is applied to the next two packets.

3.3 Late packet arrival protection

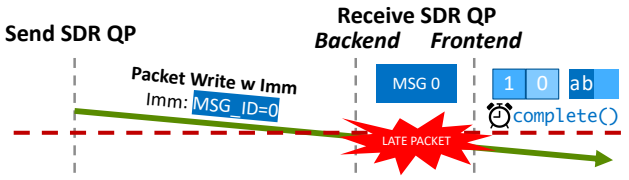


Figure 6: Late packet arrival problem.

3.3.1 Early receive completion. In the example in Figure 4, the application may complete a posted receive even if some chunks have not yet arrived. For instance, a receiver-side timeout (Figure 6) can trigger early completion while packets are still in flight. Because the data path is one-sided, the sender remains unaware and may continue injecting packets for a message that no longer exists. In both cases, receive context resources (buffer and bitmaps) must be protected from late arrivals.

3.3.2 Message ID space wraparound. Encoding the message ID in the transport immediate field is limited to 10 bits, causing wraparound every 1024 messages. If this occurs too fast, late packets may corrupt the buffer and bitmaps of a newly posted message reusing the same ID. At 800 Gbit/s and 16 MiB messages, wraparound occurs about in 100 ms — safe for RTTs below 100 ms. However, switch buffering, faster links, or smaller messages reduce this safety margin.

We solve both problems with a two-stage protection:

(1) When a message is completed, the corresponding entry in the root indirect memory key table is updated to point to a special NULL memory key that discards packet payloads (via `ibv_alloc_null_mr()` in Verbs). Writes to the NULL memory key generate completion entries for late packets, which are filtered at the second stage.

(2) To prevent bitmap corruption by late packet completion entries, we introduce the concept of *message generations*. Upon its creation, an SDR QP allocates multiple internal QPs, each associated with its own generation-for example, 4 internal QPs to support 4 message ID generations. The backend tracks the current generation for each message ID slot. For each completion entry, the current message slot generation is checked against the generation of the QP that delivered the entry. If they do not match, the completion entry is discarded.

The generation mechanism increases tolerance for late packets. Although it requires extra QPs [46], their sequential use is enabled by traffic's temporal locality and the rarity of late packet events.

3.4 Backend acceleration

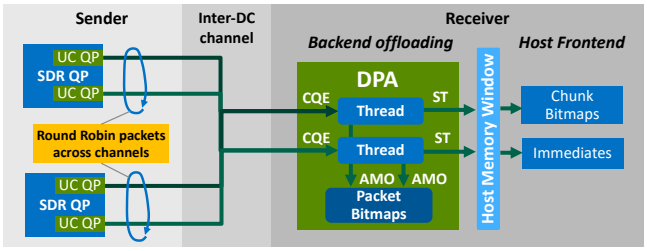


Figure 7: Multi-channel SDR offloading architecture.

Packet processing at line rates above 100 Gbit/s requires endpoint parallelism[5, 16, 21]. To this end, the SDR backend offloads packetization and bitmap processing to the Data Path Accelerator (DPA), a programmable component of the BlueField-3 and ConnectX-8 SuperNICs [8, 31, 33, 34]. The DPA, with its 256 energy-efficient hardware threads, is well-suited for parallel processing of SDR transport Write completions. We implement send- and receive-side offloading using the user-space Flex IO SDK API, part of the DOCA SDK. We focus on receive-side offloading, which handles the major of SDR's data path logic.

3.4.1 Multi-channel design. We extract traffic parallelism using the multi-channel design shown in Figure 7. For each message generation, the backend allocates multiple transport QPs, which serve traffic in parallel and act as independent channels. The sender distributes packets of a message across channels, allowing packets from different channels to be processed concurrently on the DPA. This is achieved by mapping different channels to separate completion queues, each polled by a different receive DPA worker thread. The multi-channel design enables linear scaling of protocol bandwidth with the number of DPA worker threads. Further, by spreading traffic across channel QPs, SDR could leverage intra-datacenter multi-pathing (e.g., ECMP [14, 18]) and multi-plane networks [3, 32].

3.4.2 Receive DPA worker. For each packet completion, the DPA worker thread validates the packet generation and initiates bitmap processing. The worker uses the packet offset from the transport immediate to locate and atomically update the corresponding chunk in the per-packet bitmap stored in DPA memory. The worker thread that receives the final missing packet within a chunk also updates the host-side chunk bitmap over PCIe.

4 Example reliability layers

We discuss how the SDR messaging protocol with bitmap support can be used as a base primitive to express reliable RDMA Write. We compare two example strategies for end-to-end error correction on top of SDR: Selective Repeat (SR) from the Automatic Repeat reQuest (ARQ) family of protocols, and Erasure Coding (EC) from the Forward Error Correction (FEC) family. We choose SR as an

example ARQ scheme since it can be proven theoretically that SR efficiency is at least as good as Go-back-N's [7].

In ARQ, chunk error correction requires at least one RTT to trigger retransmission, while FEC uses extra bandwidth to speculatively send parity chunks. We develop a theoretical framework for message completion time and use it to analyze how the core bandwidth-latency trade-off emerges in RDMA Write over a lossy, delayed channel.

4.1 Protocols

We illustrate both protocols in Figure 8. We assume that the client and server have established two uni-directional connections:

- *Data-path SDR QP:* for zero-copy data transfer.
- *Control-path UC (or UD) QP:* to exchange protocol acknowledgment packets (ACKs) with low overhead.

Notice that the two-connection design is not a hard requirement, and we choose it solely for the purposes of our analysis. The SDR middleware API leaves the control path wireup logic to the application implementing reliability, thereby enabling application-aware optimizations such as the optimized rendezvous protocol [42].

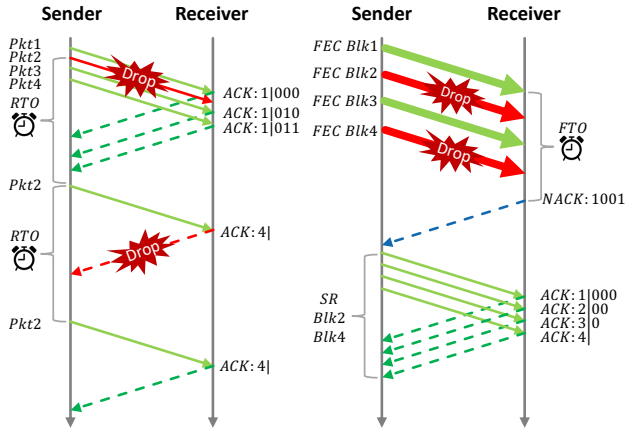


Figure 8: SDR-based SR and EC reliability protocols. Left: a 4-packet message delivered with SR. Right: an 8-packet message delivered using $EC(2,1)$.

4.1.1 SR-based reliability. We illustrate a timeout-based scheme inspired by TCP selective acknowledgment [28], treating it as the most general design. More advanced schemes, such as negative acknowledgment (NACK) [25] and retransmission timeout (RTO) tuning [39], can also be supported. We theoretically analyze how NACK could improve SR performance in Section 5.2.

SR sender uses streaming SDR send to inject message chunks into the network. Each chunk is assigned a timeout ($RTO = RTT + \alpha \cdot RTT$), where α reflects switch buffering along the sender-receiver path. When the RTO expires, the sender retransmits the chunk. Upon receiving an ACK, the sender removes all chunks within the acknowledged range from the retransmission queue.

SR receiver periodically polls the message chunk bitmap and sends ACKs to the sender. Each ACK compactly encodes the receiver's bitmap in two parts:

- *Cumulative ACK:* the highest chunk sequence number for which all previous chunks have been received.
- *Selective ACK:* a portion of the bitmap (as much as fits in the ACK payload), starting from the cumulative ACK.

4.1.2 EC-based reliability. With the EC-based scheme, costly chunk retransmissions can be avoided by computing (EC encoding) additional parity chunks for the data chunks. Parity chunks are sent speculatively alongside the data chunks-in case any data chunks are missing and enough parity chunks are received, the missing data chunks can be recovered (EC decoding).

EC sender splits the message into L data submessages of k chunks each, where $L = M/k$. Each submessage is erasure coded with m parity chunks to form a corresponding parity submessage, resulting in $2L$ one-shot SDR sends. Encoding can proceed asynchronously (e.g., on spare CPU cores [9]) during data transmission, leveraging SDR's non-blocking send semantics. After injecting all submessages and receiving a positive acknowledgment, the sender releases the message buffer.

EC receiver polls the bitmap and sends a positive ACK once enough chunks are available to recover all data submessages. If any submessage bitmap indicates missing chunks, the receiver supplies the received data and parity chunks to the EC decoder.

Fallback scheme: We must consider an edge case where the receiver cannot recover a data submessage. For example, in Maximum Distance Separable (MDS) codes, when the total number of dropped chunks across the data and corresponding parity submessage exceeds m. Our fallback strategy is to switch to Selective Repeat for the failed data submessages. When the receiver sees the first chunk of a message (the first bit in the bitmap is observed), it sets a fallback timeout ($FTO = M/BW_{channel} + 0.5 \cdot \alpha \cdot RTT$). We halve the α coefficient from the SR scheme, as only half of the buffering along the RTT path needs to be accounted for. Upon FTO expiration, the receiver sends a negative ACK (NACK) listing the failed data submessages. A global timeout is also set at message posting to prevent deadlock, though it is assumed to be rarely triggered.

Higher parity-to-data ratios in EC algorithms improve tolerance to chunk drops but increase channel bandwidth usage. We build mathematical intuition for this trade-off in the next subsection and evaluate recovery behavior across EC configurations in Section 5.2.

4.2 Message completion time model

We demonstrated SR- and EC-based reliability schemes built on SDR and now introduce a statistical framework to evaluate their performance. Our model captures key inter-datacenter parameters: drop rate, delay, bandwidth, and application message size. It is released as an open-source Python library, enabling system architects to design and tune the reliability layer to specific RDMA deployments.

4.2.1 Mathematical notation. Our model notation is as follows:

- M is the message size in chunks of receive-side bitmap.
- $T_{protocol}(M)$ is the Write completion time at the sender side when the reliablity *protocol* is used (i.e., the time interval between the injection of the first chunk and the ACK reception for the last unacknowledged chunk).

- T_{INJ} is the time to inject a chunk into the network (the inverse of chunk size divided by link bandwidth [1]).
- P_{drop} is the probability of a chunk drop on the sender-receiver path. We assume that P_{drop} is i.i.d for each chunk.

4.2.2 Selective Repeat. Message completion time for SR must account for all possible positions of a chunk drop.

For the i-th chunk (with $i = 1, \ldots, M$), we define its time:

$$X_i = t_{\text{start}}(i) + O\left(Y_i - 1\right),$$

where:

- $t_{\text{start}}(i) = i \cdot T_{INJ}$ is the start time for the i-th chunk,
- $O = RTO + T_{INJ} > 0$ is the overhead incurred for each drop,
- Y_i is a geometric random variable with success probability $1 - P_{drop}$. Y_i gives a lower bound on the number of transmissions needed for successful delivery of the i-th chunk.

We define the overall completion time of a message as the *maximum* over all individual chunk times:

$$T_{SR}(M) \geq \max_{1 \leq i \leq M} X_i + RTT.$$

Two scenarios must be distinguished in this formulation:

(1) $t_{\text{start}}(M) \leq RTO$: In this case, the initial offset $t_{\text{start}}(i)$ ensures that all retransmitted chunks are reinjected into the network at different times from the initial T_0. For example, if chunks i and $i+1$ are dropped on their first transmissions, their retransmission timeouts will differ by T_{INJ}, and they will be reinjected at times $T_0 + RTO_i$ and $T_0 + RTO_i + T_{INJ}$, respectively.
(2) $t_{\text{start}}(M) > RTO$: In this "large" message case, our derivation for $E[T_{SR}]$ becomes a *lower bound* on the expected message completion time. Consider a scenario where the first chunk is dropped. Its retransmission timestamp occurs *before* the last chunk is injected into the network for the first time at $t_{\text{start}}(M)$, thereby violating serialization.

Two methods are employed to evaluate $T_{SR}(M)$: a stochastic simulation and analytical solution for its expected value. We provide the derivation of the analytical solution in Appendix A (see supplemental materials for the paper).

4.2.3 EC-based reliability. Recall that for an erasure code with k and m chunks in data and parity submessages, respectively, the total number of independently erasure-coded data submessages is $L = M/k$. Let's denote the probability of successful recovery of a data submessage (which is a function of P_{drop}) as $P_{EC(k,m)}$.

The expected number of failed data submessages is

$$E[\text{failures}] = L \cdot \left(1 - P_{EC(k,m)}\right),$$

and the probability of at least one data submessage failing (i.e., the probability of fallback to Selective Repeat) is

$$P^{EC}_{fallback} = 1 - \left(P_{EC(k,m)}\right)^L.$$

Let the parity ratio be $R = k/m$ for $EC(k, m)$. In our protocol, the receiver sets an SR fallback timeout (FTO) as soon as any chunk of the message is received (the first bit is observed in the bitmap of any submessage):

$$FTO = (M + \lceil M/R \rceil) T_{INJ} + \beta \cdot RTT.$$

Assuming no network congestion or resource contention, two scenarios may occur on the receiver after FTO is set:

(1) By the time $FTO - \beta \cdot RTT$, the receiver has received enough chunks to recover all data submessages and sends back an ACK.
(2) FTO expires, and the receiver sends an EC NACK to the sender, requesting selective repeat of the failed data submessages (each consisting of k data chunks).

Assuming full overlap of data injection and parity computation, the lower bound on $E[T_{EC}(M)]$, is a sum of the following terms:

(1) Base time to initially send data and parity chunks:

$$(M + \lceil M/R \rceil) \cdot T_{INJ},$$

(2) *Plus* the expected time spent in timeout and EC NACK delivery:

$$P^{EC}_{fallback} \cdot (RTT + \beta \cdot RTT),$$

(3) *Plus* the expected time to retransmit the failed submessages:

$$E\left[T_{SR}(E[\text{failures}] \cdot k)\right].$$

5 Evaluation

We examine the following research questions:

(1) How do the discussed SDR-based reliability algorithms perform in various long-haul scenarios?
(2) How does the choice of reliability scheme impact performance of inter-datacenter AI training traffic?
(3) Can the SDR middleware enable partial message completion service at line rate on the current and next-generation NICs?

5.1 Experimental setup

5.1.1 SDR simulation. We implement, using Python 3, a stochastic simulation for the Write completion time with the SR and EC SDR protocols presented in Section 4. We validate simulation results against the analytical expectation for message completion time. The mean of 1000 samples from the stochastic model matches the analytical solution within 5% accuracy.

We study two scenarios for the SR-based algorithm. In the first, SR RTO, we set the SR chunk timeout to 3 network RTTs. In the second scenario, SR NACK, we reduce the RTO to 1 network RTT as a best-case approximation of the negative acknowledgment (NACK) optimization. In SR NACK, the receiver sends a negative signal to the sender indicating the specific location of the dropped chunk; therefore, the sender can initiate retransmission in 1 RTT.

For EC-based reliability we compare two erasure codes:

(1) A simple XOR-based code, in which the i'th parity block (out of m) is computed as the XOR of all k data blocks whose indices satisfy $j \bmod m = i$. This code tolerates the loss of up to one block per each modulo group [37].
(2) An MDS (Maximum Distance Separable) code (e.g., Reed-Solomon), which can recover the data submessage from any m missing blocks among the total $m + k$ blocks [47].

We study the performance trade-off between these erasure codes. XOR is simpler and easier to optimize for hardware but offers weaker chunk loss tolerance. MDS coding provides stronger protection but is more complex to implement and optimize. In Appendix B

(see supplemental materials for the paper), we derive the success probabilities to decode a data submessage with these schemes. We implement a parallel XOR-based scheme using OpenMP and AVX-512 in ≈ 100 lines of C++ and compare it against Intel's ISA-L v2.31.1 library [20], which provides an MDS code optimized for Intel CPUs.

5.1.2 End-to-end SDR testbed. We validate the ability of SDR SDK to provide its service to the upper reliability layer, a zero-copy message delivery with partial completion semantics, at line rate. We evaluate SDR offloading on nodes of the Israel-1 TOP500 production supercomputer interconnected with 400 Gbit/s RoCEv2 using NVIDIA Spectrum-X [32]. On each node, we utilize a NVIDIA BlueField-3 SuperNIC connected to a Xeon Platinum 8480 CPU [31]. SDR middleware was compiled against DOCA SDK 2.9.0.

5.2 Deploying SDR at cross-continent scale

In our first case study, we simulate SDR performance with a hypothetical 400 Gbit/s, 3750 km cross-continent link between two datacenters (e.g., within the US or Europe). Such a deployment requires the trade-off between cost and channel QoS (i.e., lower drop rates). We examine this trade-off in Figure 9 and provide a detailed analysis of the 128 MiB row and 10^{-5} column in Figure 10.

5.2.1 Use-case of EC. In red areas of Figure 9, EC outperforms SR for messages from 128 KiB to 1 GiB within the 10^{-6} to 10^{-2} drop rate range. For a fixed drop rate in Figure 10a, a critical message size $1/P$, marks the point beyond which chunk drops become likely. When the Write size nears this point but stays below or comparable to the bandwidth-delay product, SR retransmissions cannot be masked with chunk injection time, leading to slowdowns of up to 6.5× on average and 12.2× at the 99.9th percentile.

The NACK optimization reduces drop detection to 1 RTT, improving SR performance by up to 4× for both average and tail latencies. However, it cannot address the fundamental issue of SR shown in Figure 10c: RTT-scale penalty per chunk drop.

EC avoids this issue by recovering losses in-place at the receiver. Figure 10d evaluates various MDS data-parity splits. Lower data-to-parity ratios offer stronger protection at high drop rates with greater bandwidth overhead. We select the $(32, 8)$ configuration as the most balanced — it tolerates drop rates above 10^{-2} with no more than 20% bandwidth inflation.

Efficient EC implementations must hide the encoding overhead. In Figure 11 we assess the compute cost of achieving this by comparing MDS and XOR codes with a $(32, 8)$ split. At 400 Gbit/s, XOR encoding can be hidden using 4 CPU cores; MDS needs twice as many. However, XOR trades CPU efficiency for resilience: with a 128 MiB buffer, XOR falls back to SR at ≈ 10^{-3} drop rate, while MDS remains robust beyond 10^{-2}.

5.2.2 When to deploy SR?. SR performs best at drop rates below 10^{-6} and message sizes above 1 GiB. An 8 GiB message, ≈ 8× smaller than BDP, is bottlenecked by the injection time, with the final RTT adding little to the total completion time. NACK- and RTO-based SRs hide retransmissions within the injection pipeline, whereas EC introduces a 20% parity overhead.

Notice that if we were to consider a deployment with a higher RTT or more bandwidth as we do in Figure 12, EC would eventually surpass SR at message size 8 GiB, as retransmissions become more exposed due to increasing BDP.

For small messages in the bottom rows of Figure 9, SR and EC result in similar completion times. However, due to the compute footprint of EC for path encoding (and decoding in case of drops), SR is preferable. At very high drop rates above 0.1%, EC is ineffective, as it fails to recover data. As shown in Figure 10b, MDS coding wastes bandwidth sending parity, ultimately falling back to SR.

5.3 Inter-datacenter AI collectives

The point-to-point RDMA Write networking primitive, studied in the previous subsection, serves as a building block for collective algorithms. These algorithms are widely used in large-scale parallelized training (e.g., data parallelism) [6, 15, 24, 35, 51]. Primitives like Allreduce are used for synchronizing model updates across geographically distributed datacenters.

Traditional models, such as LogGP [1], assume lossless links between participants. Although valid for intra-datacenter setups (e.g., InfiniBand [19], Slingshot [11]), this assumption breaks under lossy, high-delay channels, where the choice of reliability scheme becomes paramount, as shown in Figure 13.

In Figure 13, we simulate the performance of the ring Allreduce algorithm across N datacenters with an SDR-based reliability algorithm [44]. Tail completion time is strongly affected by the ring schedule, which introduces $2N - 2$ interdependent point-to-point stages. With 4-8 datacenters, messages remain large enough that latency does not dominate, allowing slowdowns from inefficient reliability schemes to accumulate. As a result, the EC scheme, which outperforms SR for drop rates between 10^{-6} and 10^{-2}, also yields gains in multi-stage Allreduce. Across both plots, EC's speedup over SR grows with the drop rate from 3× to more than 6×.

In the Appendix C (see supplemental materials for the paper), we analytically show that the expected reliability cost in a ring Allreduce is lower bounded by the per-stage cost times the number of stages, explaining the amplified impact on reliability layer efficiency. Our analysis generalizes to other stage-based collective algorithms with schedule dependencies, such as tree algorithms [38]. The SDR framework enables performance engineers to tailor RDMA transport reliability to minimize this cumulative effect in multi-stage protocols.

5.4 End-to-end performance with BlueField-3

The analysis in previous case studies highlight the importance of flexibility in reliability protocol choice, implementation, and configuration — a capability that motivated the design of our SDR SDK. In this case study, we subject our offloaded SDR API to a 400 Gbit/s stress test on real hardware.

5.4.1 SDR client-server performance. We implemented a benchmarking loop on top of SDR middleware API. The benchmark resembles the standard client-server ib_write_bw test from the RDMA perftest suite [2]. For each message, the server emulates a reliability layer by busy polling the completion bitmap. Upon reception of all chunks, server sends an ACK to the client, which runs a timing loop. For each data point, we report average measurements collected from at least 1000 repetitions of the benchmark.

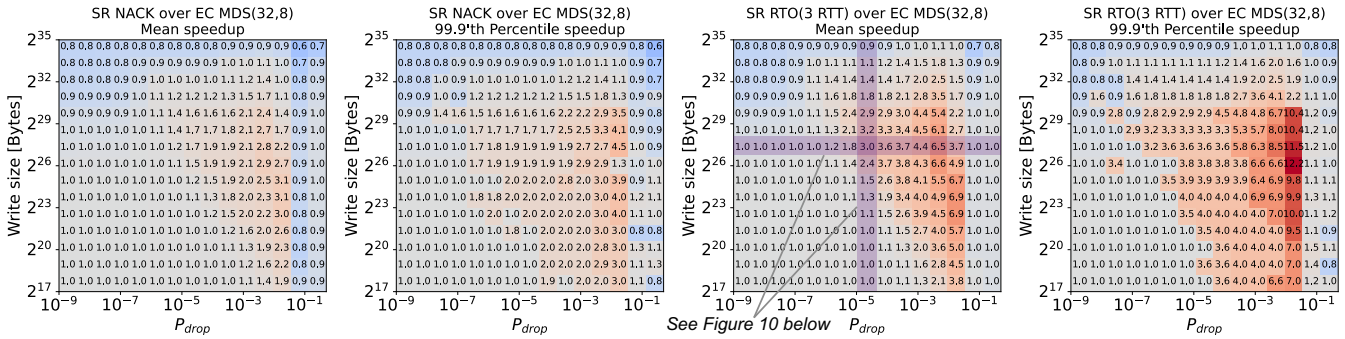


Figure 9: Erasure Coding (EC) improvements (speedup) over Selective Repeat (SR) simulated at 400 Gbit/s and 25 ms RTT.

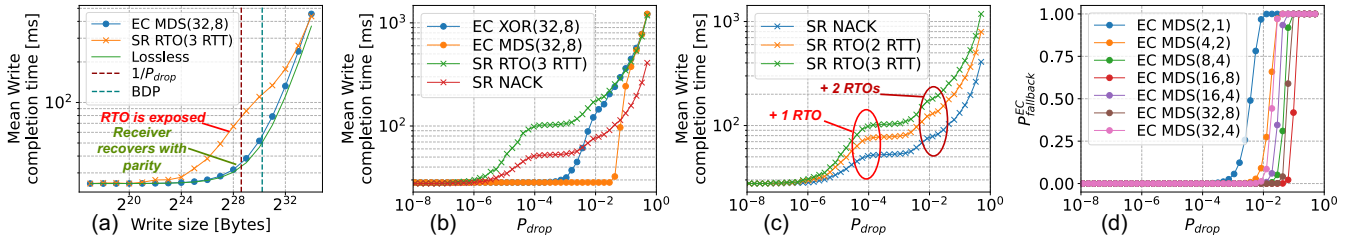


Figure 10: Subplots left to right: (a) variable sized Writes at $P_{drop} = 10^{-5}$; (b), (c), (d) 128 MiB Write at various drop rates.

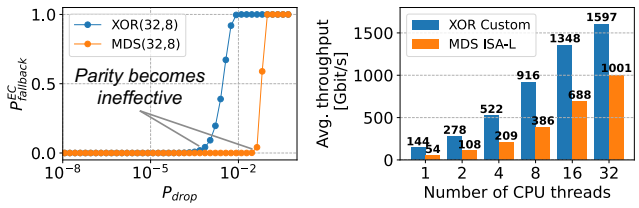


Figure 11: Comparison of MDS EC versus XOR EC. Encoding performance is evaluated on 4 Ghz Intel Xeon Platinum 8580 with 128 MiB buffer, 64 KiB chunk size, $k = 32$, $m = 8$.

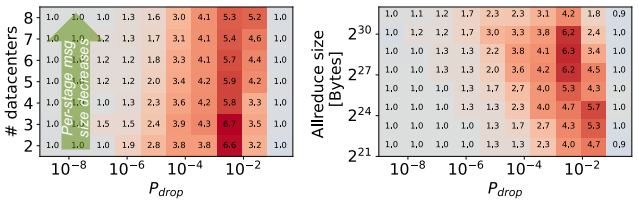


Figure 13: 99.9th percentile completion time speedup for inter-datacenter ring Allreduce with MDS EC over SR RTO reliability. Left: 128 MiB buffer size. Right: 4 datacenters.

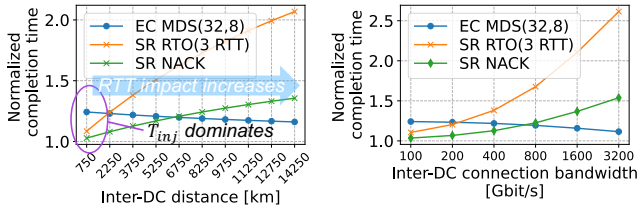


Figure 12: Impact of inter-DC distance and bandwidth on 128 MiB Write completion time. Algorithm times are normalized by a time to perform Write assuming lossless channel.

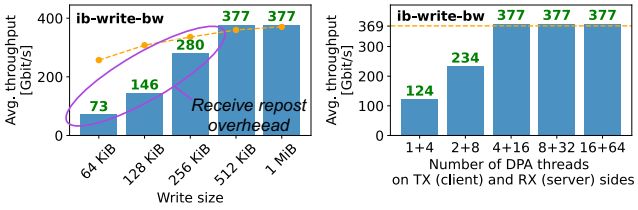


Figure 14: SDR throughput with 16 in-flight Writes and 64 KiB bitmap chunk. Left: throughput scaling at 16 DPA receive threads. Right: Thread scaling for 16 MiB messages.

We evaluate SDR's throughput scaling in Figure 14. In typical distributed training workloads, message sizes are often hundreds of megabytes [24, 51]. SDR can saturate the link's line rate with much smaller sizes, at the 512 KiB message size SDR needs just 1/16 of available DPA thread budget to sustain reception at 400G. However, for messages smaller than 512 KiB, SDR throughput is behind RC Writes due to software overhead from reposting receive buffers.

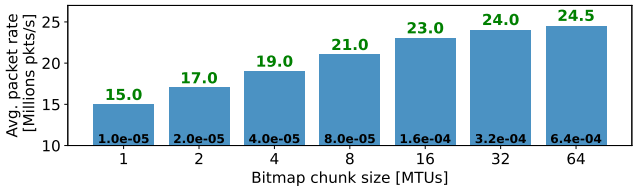


Figure 15: Impact of SDR bitmap chunk size on throughput (shown above bars) and theoretical chunk drop probability P_{drop}^{chunk} (shown at the bottom of bars) assuming packet (MTU) drop probability $P_{drop} = 1e-5$.

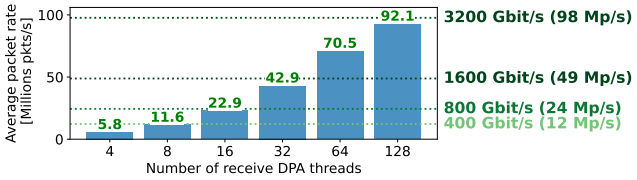


Figure 16: SDR packet rate scaling versus the number of DPA threads used for receive side offloading.

Each new receive requires message slot reallocation, including memory key table update and bitmap cleanup.

5.4.2 Impact of the SDR bitmap chunk size. The SDR bitmap's variable chunk size lets the reliability layer control how finely it detects network drops. Larger chunks increase the chance of a chunk drop being observed-since a single packet loss causes the entire chunk to be marked as lost: $P_{\text{drop}}^{\text{chunk}} = 1-(1-P_{\text{drop}})^N$, where N is the number of packets in the chunk in Figure 15. At the same time, larger chunks also reduce PCIe traffic to the host, as DPA workers update the bitmap only once every N packets.

Interestingly, we observed that 16 receive threads from the best configuration in Figure 14 are enough to deliver line rate both at minimum 1-packet chunk of 4096 KiB and maximum 64-packet chunk of 256 KiB. In Figure 15 we show further investigation of this phenomena with transport Write size reduced to 64 bytes and bitmap chunk size scaled correspondingly. Reducing chunk size allows us to saturate the link with more packets, while keeping the DPA worker's per-packet load constant, because DPA workers process packet Write completions (not payload!) and their cycle footprint is independent of packet size. We observe that in the worst-case configuration of 1-packet chunks, 16 DPA threads can sustain up to 15 million packets per second, while the theoretical packet rate of 400 Gbit/s link at 4 KiB MTU is 11.6 million. This experiment shows that SDR API offloading enables control of the drop rate at the upper reliability layer without compromising performance.

5.4.3 SDR performance with Tbit/s links. Finally, in Figure 16 we examine SDR's ability to serve traffic at packet rates expected with next-generation Tbit/s link bandwidths at 4 KiB MTUs and 64 KiB chunks. To maximize packet rate load at the receiver, we use the methodology from the previous example and use 4 CPU threads to generate 64 byte packets on the client side. DPA-based offloading scales nearly linearly across 4 to 32 threads. At 32 threads (1/8 of DPA capacity), SDR reaches packet rates near 1.6 Tbit/s. Scaling to 128 threads brings throughput close to 3.2 Tbit/s. This demonstrates that SDR's multi-channel backend, combined with DPA offloading, decouples per-packet progress from upper-layer reliability and ensures scalability for next-generation links.

6 Related Work

IRN [29] and SRNIC [46] address design limitations of commodity RDMA NICs. Like SDR, they use per-connection bitmaps for retransmission-based reliability; however, these bitmaps are internal to the FPGA and hidden from users, limiting experimentation with alternative schemes and compatibility with commodity NICs. Their wire protocols are also incompatible with commodity NICs.

Flor [23] enhances Go-back-N reliability in ConnectX-4/5 RC transport by supporting selective retransmission an top of UC. Unlike SDR's partial message completion, it lacks a unified abstraction for general transport-layer reliability and does not support offloading. Khalilov et al. [21] use a software bitmap for multicast-based collectives. Their offload-oriented design also supports UC but does not generalize to arbitrary traffic, unlike SDR.

Two decades ago, Lundqvist and Karlsson [26] showed that end-to-end FEC can significantly boost TCP Reno, SACK, and Tahoe throughput for Internet traffic. Maelstrom [4] introduces a proxy-based design that applies FEC to UDP traffic at the datacenter edge.

More recent works, Cloudburst [50] and LoWAR [53] apply FEC to the datacenter transport layer. Cloudburst distributes coded packets across parallel paths for early recovery, while LoWAR targets long-haul links. Both outperform retransmission protocols but lack scalability analysis for next-generation links, limit users to FEC-based reliability and work on top of custom transport.

7 Conclusion

We presented SDR, a novel software-defined RDMA stack that enables custom reliability algorithms for long-haul RDMA across datacenters. SDR introduces partial message completion via a bitmap API, empowering developers to tailor reliability strategies, such as Selective Repeat and Erasure Coding, to specific network conditions without sacrificing RDMA's zero-copy performance. By offloading packet-processing logic to the NIC, SDR achieves full line rate performance on current generation hardware and supports packet rates of next-generation Tbit/s links. SDR offers immediate, deployable improvements over existing NIC solutions, unlocking optimized inter-datacenter GPU training communication.

Acknowledgments

We thank CSCS and Jérôme Tissières for providing the infrastructure used to perform some of the experiments. This work is supported by the following grant agreements: SwissTwins (funded by the swiss State Secreteriat for Education, Research and Innnovation), ERC PSAP (grant agreement No 101002047), WeatherGenerator (grant agreement No 101187947). The authors used ChatGPT and Perplexity for minor editing and proofreading during the preparation of the manuscript. All ideas and content are exclusively the original work of the authors.

References

[1] Albert Alexandrov, Mihai F Ionescu, Klaus E Schauser, and Chris Scheiman. 1995. LogGP: Incorporating long messages into the LogP model—one step closer towards a realistic model for parallel computation. In *Proceedings of the seventh annual ACM symposium on Parallel algorithms and architectures*. 95–105.

[2] OpenFabrics Alliance. 2025. Open Fabrics Enterprise Distribution (OFED) Performance Tests. https://github.com/linux-rdma/perftest.

[3] Wei An, Xiao Bi, Guanting Chen, Shanhuang Chen, Chengqi Deng, Honghui Ding, Kai Dong, Qiushi Du, Wenjun Gao, Kang Guan, et al. 2024. Fire-Flyer AI-HPC: A Cost-Effective Software-Hardware Co-Design for Deep Learning. In *SC24: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–23.

[4] Mahesh Balakrishnan, Tudor Marian, Ken Birman, Hakim Weatherspoon, and Einar Vollset. 2008. Maelstrom: Transparent Error Correction for Lambda Networks.. In *NSDI*. 263–278.

[5] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. 2014. IX: a protected dataplane operating system for high throughput and low latency. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. 49–65.

[6] Tal Ben-Nun and Torsten Hoefler. 2019. Demystifying parallel and distributed deep learning: An in-depth concurrency analysis. *ACM Computing Surveys (CSUR)* 52, 4 (2019), 1–43.

[7] Dimitri Bertsekas and Robert Gallager. 2021. *Data networks*. Athena Scientific.

[8] Xuzheng Chen, Jie Zhang, Ting Fu, Yifan Shen, Shu Ma, Kun Qian, Lingjun Zhu, Chao Shi, Yin Zhang, Ming Liu, et al. 2024. Demystifying datapath accelerator enhanced off-path smartnic. In *2024 IEEE 32nd International Conference on Network Protocols (ICNP)*. IEEE, 1–12.

[9] Marcin Chrapek, Mikhail Khalilov, and Torsten Hoefler. 2023. HEAR: Homomorphically Encrypted Allreduce. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–17.

[10] João da Silva. 2024. Google turns to nuclear to power AI data centres. *BBC News* (15 October 2024). https://www.bbc.com/news/articles/c748gn94k95o Business.

[11] Daniele De Sensi, Salvatore Di Girolamo, Kim H McMahon, Duncan Roweth, and Torsten Hoefler. 2020. An in-depth analysis of the slingshot interconnect. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–14.

[12] Jeffrey Dean and Luiz André Barroso. 2013. The tail at scale. *Commun. ACM* 56, 2 (2013), 74–80.

[13] Haotian Dong, Jingyan Jiang, Rongwei Lu, Jiajun Luo, Jiajun Song, Bowen Li, Ying Shen, and Zhi Wang. 2025. Beyond A Single AI Cluster: A Survey of Decentralized LLM Training. *arXiv preprint arXiv:2503.11023* (2025).

[14] 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.

[15] Manjunath GorentlaVenkata, Valentine Petrov, Sergey Lebedev, Devendar Bureddy, Ferrol Aderholdt, Joshua Ladd, Gil Bloch, Mike Dubman, and Gilad Shainer. 2025. Unified Collective Communication (UCC): A Unified Library for CPU, GPU, and DPU Collectives. *IEEE Micro* (2025).

[16] Torsten Hoefler, Salvatore Di Girolamo, Konstantin Taranov, Ryan E Grant, and Ron Brightwell. 2017. sPIN: High-performance streaming Processing in the Network. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–16.

[17] Torsten Hoefler, Duncan Roweth, Keith Underwood, Robert Alverson, Mark Griswold, Vahid Tabatabaee, Mohan Kalkunte, Surendra Anubolu, Siyuan Shen, Moray McLaren, et al. 2023. Data center ethernet and remote direct memory access: Issues at hyperscale. *Computer* 56, 7 (2023), 67–77.

[18] C. Hopps. 2009. Analysis of an Equal-Cost Multi-Path Algorithm. RFC 2992. https://www.ietf.org/rfc/rfc2992.txt

[19] InfiniBand Trade Association. 2024. InfiniBand Specification. https://www.infinibandta.org.

[20] Intel. 2025. Intel Intelligent Storage Acceleration Library. https://www.intel.com/content/www/us/en/developer/tools/isa-l/overview.html.

[21] Mikhail Khalilov, Salvatore Di Girolamo, Marcin Chrapek, Rami Nudelman, Gil Bloch, and Torsten Hoefler. 2024. Network-Offloaded Bandwidth-Optimal Broadcast and Allgather for Distributed AI. In *SC24: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–17.

[22] Adam Langley, Alistair Riddoch, Alyssa Wilk, Antonio Vicente, Charles Krasic, Dan Zhang, Fan Yang, Fedor Kouranov, Ian Swett, Janardhan Iyengar, et al. 2017. The quic transport protocol: Design and internet-scale deployment. In *Proceedings of the conference of the ACM special interest group on data communication*. 183–196.

[23] Qiang Li, Yixiao Gao, Xiaoliang Wang, Haonan Qiu, Yanfang Le, Derui Liu, Qiao Xiang, Fei Feng, Peng Zhang, Bo Li, et al. 2023. Flor: An open high performance RDMA framework over heterogeneous NICs. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. 931–948.

[24] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, et al. 2020. Pytorch distributed: Experiences on accelerating data parallel training. *arXiv preprint arXiv:2006.15704* (2020).

[25] Yuanwei Lu, Guo Chen, Bojie Li, Kun Tan, Yongqiang Xiong, Peng Cheng, Jiansong Zhang, Enhong Chen, and Thomas Moscibroda. 2018. Multi-Path transport for RDMA in datacenters. In *15th USENIX symposium on networked systems design and implementation (NSDI 18)*. 357–371.

[26] Henrik Lundqvist and Gunnar Karlsson. 2004. TCP with end-to-end FEC. In *International Zurich Seminar on Communications, 2004*. IEEE, 152–155.

[27] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Michael Dalton, Nandita Dukkipati, William C Evans, Steve Gribble, et al. 2019. Snap: A microkernel approach to host networking. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 399–413.

[28] Matt Mathis, Jamshid Mahdavi, Sally Floyd, and Allyn Romanow. 1996. RFC2018: TCP selective acknowledgement options.

[29] Radhika Mittal, Alexander Shpiner, Aurojit Panda, Eitan Zahavi, Arvind Krishnamurthy, Sylvia Ratnasamy, and Scott Shenker. 2018. Revisiting network support for RDMA. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*. 313–326.

[30] Andrew Moseman. 2024. Amazon Vies for Nuclear-Powered Data Center: The deal has become a flash point over energy fairness. *IEEE Spectrum* (12 August 2024). https://spectrum.ieee.org/amazon-data-center-nuclear-power

[31] NVIDIA. 2023. NVIDIA BlueField-3 Datasheet. https://resources.nvidia.com/en-us-accelerated-networking-resource-library/datasheet-nvidia-bluefield.

[32] NVIDIA. 2023. Spectrum-X Datasheet. https://resources.nvidia.com/en-us-networking-ai/networking-ethernet-1.

[33] NVIDIA. 2025. ConnectX-8 SuperNIC Datasheet. https://resources.nvidia.com/en-us-accelerated-networking-resource-library/connectx-datasheet-c.

[34] NVIDIA. 2025. DPA Subsystem. https://docs.nvidia.com/doca/sdk/DPA+Subsystem/index.html.

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

[36] OpenFabrics Alliance. 2025. Libfabric OpenFabrics. https://ofiwg.github.io/libfabric/.

[37] David A Patterson, Garth Gibson, and Randy H Katz. 1988. A case for redundant arrays of inexpensive disks (RAID). In *Proceedings of the 1988 ACM SIGMOD international conference on Management of data*. 109–116.

[38] Peter Sanders, Jochen Speck, and Jesper Larsson Träff. 2009. Two-tree algorithms for full bandwidth broadcast, reduction and scan. *Parallel Comput.* 35, 12 (2009), 581–594.

[39] Pasi Sarolahti, Markku Kojo, and Kimmo Raatikainen. 2003. F-RTO: an enhanced recovery algorithm for TCP retransmission timeouts. *ACM SIGCOMM Computer Communication Review* 33, 2 (2003), 51–63.

[40] SemiAnalysis. 2024. Multi-Datacenter Training: OpenAI's Ambitious Plan To Beat Google's Infrastructure. https://semianalysis.com/2024/09/04/multi-datacenter-training-openais/.

[41] Foteini Strati, Paul Elvinger, Tolga Kerimoglu, and Ana Klimovic. 2024. ML training with Cloud GPU shortages: Is cross-region the answer?. In *Proceedings of the 4th Workshop on Machine Learning and Systems*. 107–116.

[42] Sayantan Sur, Hyun-Wook Jin, Lei Chai, and Dhabaleswar K Panda. 2006. RDMA read based rendezvous protocol for MPI over InfiniBand: design alternatives and benefits. In *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*. 32–39.

[43] Gemini Team, Rohan Anil, Sebastian Borgeaud, Jean-Baptiste Alayrac, Jiahui Yu, Radu Soricut, Johan Schalkwyk, Andrew M Dai, Anja Hauth, Katie Millican, et al. 2023. Gemini: a family of highly capable multimodal models. *arXiv preprint arXiv:2312.11805* (2023).

[44] Rajeev Thakur and William D Gropp. 2003. Improving the performance of collective operations in MPICH. In *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*. Springer, 257–267.

[45] Hao Wang, Sreeram Potluri, Devendar Bureddy, Carlos Rosales, and Dhabaleswar K Panda. 2013. GPU-aware MPI on RDMA-enabled clusters: Design, implementation and evaluation. *IEEE Transactions on Parallel and Distributed Systems* 25, 10 (2013), 2595–2605.

[46] 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)*. 1–14.

[47] Stephen B Wicker and Vijay K Bhargava. 1999. *Reed-Solomon codes and their applications*. John Wiley & Sons.

[48] Linda Winkler. 2015. SCinet: 25 years of extreme networking. In *Proceedings of the Second Workshop on Innovating the Network for Data-Intensive Science*. 1–9.

[49] Behrooz Zahiri. 2003. Structured ASICs: opportunities and challenges. In *Proceedings 21st International Conference on Computer Design*. IEEE, 404–409.

[50] Gaoxiong Zeng, Li Chen, Bairen Yi, and Kai Chen. 2022. Cutting tail latency in commodity datacenters with cloudburst. In *IEEE INFOCOM 2022-IEEE Conference on Computer Communications*. IEEE, 600–609.

[51] Yanli Zhao, Andrew Gu, Rohan Varma, Liang Luo, Chien-Chin Huang, Min Xu, Less Wright, Hamid Shojanazeri, Myle Ott, Sam Shleifer, et al. 2023. Pytorch fsdp: experiences on scaling fully sharded data parallel. *arXiv preprint arXiv:2304.11277* (2023).

[52] Danyang Zhuo, Monia Ghobadi, Ratul Mahajan, Klaus-Tycho Förster, Arvind Krishnamurthy, and Thomas Anderson. 2017. Understanding and mitigating packet corruption in data center networks. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. 362–375.

[53] Tianyu Zuo, Tao Sun, Shuyong Zhu, Wenxiao Li, Lu Lu, Zongpeng Du, and Yujun Zhang. 2024. LoWAR: Enhancing RDMA over Lossy WANs with Transparent Error Correction. In *2024 IEEE/ACM 32nd International Symposium on Quality of Service (IWQoS)*. IEEE, 1–10.

Appendix: Artifact Description/Artifact Evaluation

Artifact Description (AD)

A Overview of Contributions and Artifacts

A.1 Paper's Main Contributions

C_1 Analysis of inter-datacenter communication challenges.
C_2 SDR-RDMA architecture that decouples application-specific reliability logic from low-level packet processing.
C_3 SDR-RDMA data path offloading for full line-rate performance on current and next-generation commodity NICs.
C_4 Framework to simulate and analyze the performance of SDR-based reliability algorithms in an inter-datacenter setup.

A.2 Computational Artifacts

Artifact DOI: https://doi.org/10.5281/zenodo.16760233
Artifact Git Repo: https://github.com/spcl/sdr-rdma-artifact

Artifact ID	Contributions Supported	Related Paper Elements
A_1	C_1, C_4	Figures 3, 9-12
A_2	C_2, C_3	Table 1 Figures 1, 4-7, 8, 14-16
A_3	C_1, C_4	Figures 2, 3, 9-12
A_4	C_3	Figure 11

B Artifact Identification

B.1 Computational Artifact A_1

Relation To Contributions

This artifact (folder "sdr-model") is a simulation framework for SDR-based reliability algorithms. Simulation core is implemented as a Python 3 module. It also includes a Python 3 notebook that allows parameterizing the simulation and generating plots.

Expected Results

It provides simulated evidence of scenarios where SR outperforms EC and vice versa. The default number of iterations is reduced in comparison to experiments in the paper to ensure fast artifact reproducibility. This doesn't affect key experimental takeaways.

Expected Reproduction Time (in Minutes)

The expected time to run the simulation artifact on a system with an Mac M1 ARM CPU is 10 min.

Artifact Setup (incl. Inputs)

Hardware. Personal computer.

Software. Required software to run simulation:

- Python v3.13.5
- Jupyter Notebook (Conda v24.9.2).
- Matplotlib v3.10.0
- tqdm v4.67.1
- numpy 2.2.2

Datasets / Inputs.

- "sdr-model/fct_model.ipynb" notebook contains simulations related to point-to-point results.
- "sdr-model/ar_model.ipynb" notebook contains simulations related to the Allreduce.

Installation and Deployment. To run the simulation model, Python dependencies listed above must be installed on the system (one can use "sdr-model/requirements.txt" to automate dependencies installation).

Artifact Execution

All cells of both notebooks must be executed.

B.2 Computational Artifact A_2

Relation To Contributions

This artifact ("sdr-api" folder) is an open-sourced SDR SDK API header file with an SDR perftest benchmark implemented on top of it. The SDR perftest benchmark reports client-server throughput for a given message size, bitmap chunk size, and number of threads in the SDR backend. SDR perftest can be compiled and executed against a library that implements the SDR API.

Expected Results

Assuming that the SDR API implementation leverages DPA offloading, similarly to our implementation, a 400 Gbit/s line rate can be reached with SDR DPA-based offloading using 4 TX and 16 RX threads. 128 DPA threads achieve over 90 million packets per second.

Expected Reproduction Time (in Minutes)

The expected computational time to run the benchmark artifact on a two-node system with DPA offloading is 30 min.

Artifact Setup (incl. Inputs)

Hardware. Experiments involving the DPA acceleration are performed using two servers connected to each other through SpectrumX switch with BlueField-3 400 Gbit/s DPUs installed on both servers.

Software. DOCA SDK v2.9.0.

Inputs. We open-source the SDR perftest benchmark code used to collect performance of proprietary closed-source SDR SDK across different message sizes, thread counts, etc.

Installation and Deployment. The benchmark code and SDR API header files can be found in the artifact repo: https://github.com/spcl/sdr-rdma-artifact.

Artifact Execution

Assuming that the SDR API header was implemented and benchmark was compiled against it, to run the benchmark, a server should start the benchmark in listening mode, and a client should run the client-side benchmark by specifying the server IP address.

B.3 Computational Artifact A_3

Relation To Contributions

This artifact ("iperf" folder) is a set of wrapper scripts around iperf3 to collect the UDP drop rate.

Expected Results

Depending on the network under consideration, the benchmark will report the UDP drop rate. For example, on a non-isolated network under switch buffer load, the observed drop rate could be high (e.g., above 10^{-4}).

Expected Reproduction Time (in Minutes)

The expected computational time to run a single sample is ≈ 15 seconds.

Artifact Setup (incl. Inputs)

Hardware. Two nodes connected through a network that supports UDP protocol communication.

Software.

- iperf-3.19
- Python 3.13.5.

Datasets / Inputs. The "./batch_experiments.sh" script contains all test-case scenarios (different MTUs) discussed in the paper.

Installation and Deployment. The script used to run the experimental batch ("./batch_experiments.sh") must be parameterized with client and server details, e.g., IP addresses of the data path and control path interfaces.

Artifact Execution

The following command will run 10 experiments of 30 seconds each with a sleep of 2 seconds between each experiment:

"./batch_experiments.sh server-MGMT-IP server-NIC-IP ./logs-many-trials/ 30 10"

B.4 Computational Artifact A_4

Relation To Contributions

This artifact ("ec-perf" repo submodule) is a benchmark of various erasure coding schemes. It includes a benchmarking front-end for open-source libraries (e.g., Intel ISA-L supporting MDS-based erasure coding), and a custom implementation of XOR-based erasure coding.

Expected Results

On a modern HPC-grade CPU with AVX512 support, 4–8 threads should be sufficient to reach 400 Gbit/s encryption speed for a 128 MiB buffer.

Expected Reproduction Time (in Minutes)

The expected computational time to run the benchmark comparing ISA-L library to the custom XOR implementation is 10 minutes.

Artifact Setup (incl. Inputs)

Hardware. A server with a multi-core CPU.

Software. GCC v13.3.0 compiler and OpenMP runtime (libgomp of GCC).

Datasets / Inputs. All test cases are contained within the benchmarking package.

Installation and Deployment. Project repository provides all instructions to run the deploy and run benchmark.

Artifact Evaluation (AE)

C.1 Computational Artifact A_1

The artifact evaluation part provides steps to reproduce the results obtained with the artifact A_1 of the reliability layer presented in the paper (Figures 3, 9-12).

Artifact Setup (incl. Inputs)

Initial setup includes executing the command:
"pip install -r sdr-model/requirements.txt".

Artifact Execution

(1) To reproduce the results, the user needs to execute all cells of the two Jupyter notebooks inside the "sdr-model" folder (e.g., both notebooks can be executed by pressing the "Run All" button in the Notebook interface of VS Code or the Jupyter browser frontend).
(2) Both notebooks use network parameters similar to those described in the paper. To modify these parameters, the user can edit the variables defined in "sdr-model/network-params.py".

Artifact Analysis (incl. Outputs)

- Resulting plots of the experimental outcomes will be stored in the "sdr-model/plots/" directory.
- Output plots should show trends similar to those presented in the paper. However, as described in the AD section.
- Slight variations in figures are possible due to the reduced number of iterations in the artifact and stochastic nature of experiments.

C.2 Computational Artifact A_2, A_3, A_4

The detailed instructions of how to setup, execute, and analyze the provided artifacts 2-4 is included in the subsections of the artifact description part above.

Reproducibility Report

D Overview of Reproduction of Artifacts

The following table provides an overview of each computational artifact's reproducibility status. Artifact IDs correspond to those in the AD/AE Appendices.

Artifact ID	Available	Functional	Reproduced
A_1	●	●	●
A_2	●	○	○
A_3	●	○	○
A_4	●	○	○
Badge awarded	yes	yes	yes

E Reproduction of Computational Artifacts

E.1 Timeline

The experiments conducted for artifact evaluation were performed from 4-6 August 2025.

E.2 Computational Environment and Resources

The experiments conducted for artifact evaluation were performed on a Dell Precision 5480 with a 13th Gen Intel(R) Core(TM) i9-13900H, 2600 Mhz, with 14 cores and 20 logical processors. Note that only pertains to artifact A_1, as none of the other artifacts were able to be properly configured, built, and installed using the direction provided (or those available in a linked repository). The installation of artifacts A_3 and A_4 was attempted on the CPU partition of Permutter at NERSC, whose nodes have 2 AMD EPYC 7763 CPUs and are connected with a HPE Slingshot 11 interconnect. The installation of A_2 was not attempted due to both a lack of access to the required hardware and the absence of any build instruction or system for the textttsdr-api directory.

E.3 Details on Artifact Reproduction

(1) Artifact A_1: We first note that this is the only artifact mentioned in the AE appendix. Following the instructions in that appendix was sufficient to install dependencies and run the scripts for this artifact. The generated plots are shown in Fig 17, 18, 19, 20, and 21. In all cases, the generated figures match their counterparts in the original paper well.

(2) Artifact A_2: No instructions are provided for building the code found in the `adr-api` directory, either in the AD appendix or in the readme file present in the directory. Additionally, there is no build system (Makefile, CMakeLists, etc) present from which instructions could be inferred, nor does the naive approach of simply running a C compiler on `sdr_write_bw.c` work (in our case this compiler was gcc [Ubuntu 11.4.0-1ubuntu1 22.04] 11.4.0) due to a missing header file, namely
`#include <contrib/lwlog.h>`. As we were unable to install the DOCA SDK on this device due to a lack of hardware support, we suspect this is the root cause of the error, but were unable to confirm this.

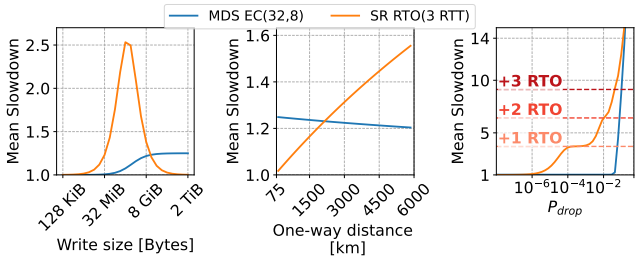


Figure 17: Corresponds to Figure 3 in the original paper.

(3) Artifact A_3: We were able to install iperf3 successfully, but unable to run `batch_experiments.sh` due to a lack of instructions on how to properly configure the two nodes and collect the input parameters necessary to run this script.

(4) Artifact A_4: We attempted to follow the instructions for installing ec-perf and its dependencies found in the readme of that repository, but they required root privileges on the system which we do not have on Perlmutter (as is common for many HPC users).

Overall, we decided to award all badges after discussing these results with the paper reviewers. They find that A_1 constitutes a major component of the original paper, and consider Figure 9 in the original paper (Figure 18 here) to be especially important to the paper's conclusions. As a result, we consider our ability to reproduce the results related to A_1 sufficient to grant all three requested badges.

That being said, this decision should not be construed as endorsing the results pertaining to the other three artifacts, which we were not able to evaluate fully. Our inability to access the required hardware were a major contributor to this result, though we note that important details are missing from the instructions provided for all remaining artifacts. These issues would likely have made reproducing the achieved results difficult even with access to the required hardware.

Disclaimer: This Reproducibility Report was crafted by volunteers with the goal of enhancing reproducibility in our research domain. The time period allocated for the reproducibility analysis was constrained by paper notification deadlines and camera-ready submission dates. Furthermore, the compute hours in the shared infrastructure (e.g., Chameleon Cloud) available to the authors of this report were limited and restricted the scope and quantity of experiments in the review phase. Consequently, the inability to reproduce certain artifacts within this evaluation should not be interpreted as definitive evidence of their irreproducibility. Limitations in the time allocated to this review and the compute resources available to the reviewers may have prevented a positive outcome. Furthermore, reviewers assess the reproducibility of the artifacts provided by the authors; however, they are not accountable for verifying that the artifacts support the main claims of the paper.

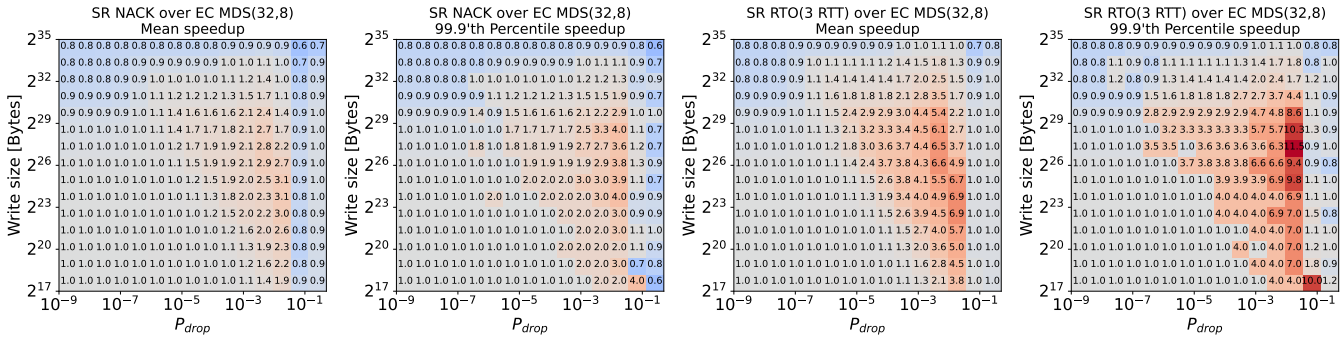


Figure 18: Corresponds to Figure 9 in the original paper.

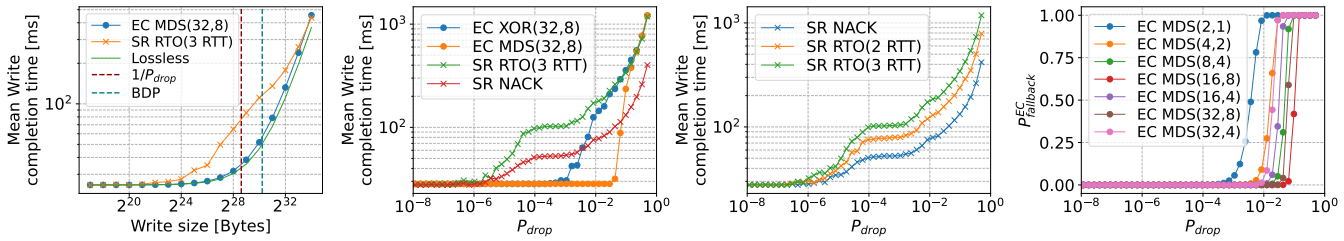


Figure 19: Corresponds to Figure 10 in the original paper.

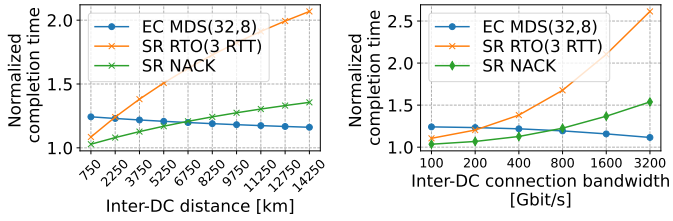


Figure 20: Corresponds to Figure 12 in the original paper.

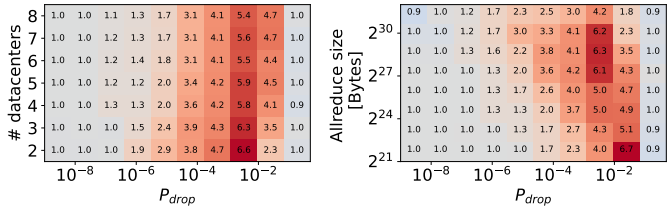


Figure 21: The sole plot produced by `ar-model.ipynb`. Corresponds to Figure 13 in the paper.