國立臺灣大學電機資訊學院資訊網路與多媒體研究所

碩士論文

Graduate Institute of Networking and Multimedia

College of Electrical Engineering and Computer Science

National Taiwan University

Master's Thesis

基於遠端直接記憶體存取之跨機器低延遲圖形處理器發佈/訂閱模式

Design of a Low-Latency RDMA-Based Publish/Subscribe Framework for Inter-Machine GPU Communications

姜任懋

REN-MAO JIANG

指導教授: 洪士灝 博士

Advisor: Shih-Hao Hung, Ph.D.

中華民國 115 年 1 月

January, 2026

doi:10.6342/NTU202600379

國立臺灣大學碩士學位論文
口試委員會審定書
MASTER'S THESIS ACCEPTANCE CERTIFICATE
NATIONAL TAIWAN UNIVERSITY

基於遠端直接記憶體存取之跨機器低延遲圖形處理器發佈/訂閱模式

Design of a Low-Latency RDMA-Based Publish/Subscribe Framework for Inter-Machine GPU Communications

本論文係 姜任懋 （學號 R12944042）在國立臺灣大學資訊網路與多媒體研究所完成之碩士學位論文，於民國 115 年 1 月 19 日承下列考試委員審查通過及口試及格，特此證明。

The undersigned, appointed by the Graduate Institute of Networking and Multimedia on 19 January 2026 have examined a Master's Thesis entitled above presented by JIANG, REN-MAO (student ID: R12944042) candidate and hereby certify that it is worthy of acceptance.

口試委員 Oral examination committee:

洪士灝	施吉昇	
（指導教授 Advisor）		
		張原豪
		余家宏

系（所）主管 Director: 鄭卜壬

doi:10.6342/NTU202600379

Acknowledgements

首先，我最想感謝我的指導教授洪士灝老師，感謝當初老師願意收我為學生。老師在計算機系統研究領域中涉略非常深，見識也相當廣博。實驗室在系統領域的研究方向相當多元，給予了我們很大的彈性，讓我能依據自己的興趣去探索，而本質上大多聚焦於系統效能的改進，這正是老師的專長所在。這兩年多來，我從老師身上學到了許多，不僅是專業知識，更重要的是正確做研究的方法與態度。很感謝老師儘管平時雖然繁忙，卻從未缺席與我們的會議，也謝謝老師一步步帶領我修正研究論文的方向與錯誤，讓我能順利完成這篇碩士論文。

再來，我想要感謝 Edge AI Group 的夥伴們。謝謝澔恩和詠翔，我們一起完成了論文的發表，在課業上也時常互相扶持，我在你們身上學習到了很多寶貴的知識和技能。也謝謝宜興、益銓學長和志豪學長，平時在研究探索或報告上給予我許多的建議，並指正我的錯誤。

最後，也謝謝實驗室的其他同學、學長姐和學弟妹，感謝實驗室所有曾經幫助過我的人。

doi:10.6342/NTU202600379

摘要

隨著自駕車、人工智慧與即時影像處理等高效能運算應用的興起，GPU 已成為資料生成與處理的核心運算單元。現有的發佈/訂閱中介軟體大多針對 CPU 架構設計。然而對於 GPU 裝置記憶體之間的發佈/訂閱會涉及多次冗餘的主機記憶體及 GPU 裝置記憶體之間的複製與傳輸。這不僅造成顯著的延遲，更加劇了 PCIe 匯流排的頻寬競爭，限制了系統的整體平行處理能力。儘管現有的 GPU 發佈/訂閱方案能透過共享記憶體優化節點內的傳輸，但其仍受限於單一主機的 GPU 計算資源及記憶體，無法滿足分散式大規模 GPU 叢集的需求。

為解決上述問題，本論文提出了一種基於 Remote Direct Memory Access (RDMA) 的低延遲跨節點 GPU 感知發佈/訂閱框架——Remote GPU-Aware Publish/Subscribe (RGAPS)。RGAPS 將單一機器 GPU 發佈/訂閱的設計理念從單機環境延伸至跨機器 GPU 通訊。在架構設計上，本研究結合了 RDMA Write with Immediate 機制來實現繞過 CPU 的高效節點間傳輸，並利用接收端的共享 GPU 記憶體技術，將接收到的資料寫入為單一共享副本資料後供多個本地訂閱者並行存取。相較於傳統架構中開銷隨訂閱者數量呈線性增長，本方法消除了對冗餘記憶體複製操作的需求。

實驗結果顯示，在節點內分發的可擴展性方面，RGAPS 成功解耦了分發延遲與訂閱者數量的關聯；當傳輸 1GB 的大型負載給 8 個訂閱者時，本架構展現了極

doi:10.6342/NTU202600379

低的延遲與近乎恆定的效能表現，而傳統方法則因 PCIe 頻寬競爭導致延遲大幅攀升。總結而言，RGAPS 有效突破了跨節點 GPU 通訊的軟體與硬體瓶頸，為需要高頻寬、低延遲的大型分散式 GPU 應用及資料傳輸提供了一個具備高擴展性的通訊解決方案。

關鍵字：發布/訂閱系統、圖形處理器 (GPU)、遠端直接記憶體存取 (RDMA)、共享記憶體、中介軟體、高效能運算

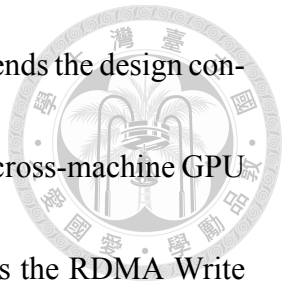
doi:10.6342/NTU202600379

Abstract

With the rise of high-performance computing applications such as autonomous driving, artificial intelligence, and real-time image processing, GPU has become the core computational unit for data generation and processing. Most existing publish/subscribe middleware is designed for CPU architectures. However, publish/subscribe operations between GPU device memories involve multiple redundant copies and transmissions between host memory and GPU device memory. This not only causes significant latency but also exacerbates PCIe bus bandwidth contention, limiting the system's overall parallel processing capability. Although existing GPU publish/subscribe schemes can optimize intra-node transmission through shared memory, they remain limited to the GPU computational resources and memory of a single host, failing to meet the needs of distributed large-scale GPU clusters.

To address the aforementioned issues, this thesis proposes a low-latency, cross-node GPU-aware publish/subscribe framework based on Remote Direct Memory Access (RDMA),

doi:10.6342/NTU202600379



named Remote GPU-Aware Publish/Subscribe (RGAPS). RGAPS extends the design concept of GPU publish/subscribe from a single-machine environment to cross-machine GPU communication. In terms of architectural design, this study combines the RDMA Write with Immediate mechanism to achieve efficient inter-node transmission that bypasses the CPU. Furthermore, it utilizes shared GPU memory technology on the receiving end to materialize received data as a single shared copy, allowing multiple local subscribers to access it in parallel. In contrast to conventional architectures where overhead scales linearly with the subscriber count, this approach eradicates the need for redundant memory copy operations.

Experimental results demonstrate that in terms of intra-node distribution scalability, RGAPS successfully decouples distribution latency from the number of subscribers. When transmitting a large 1GB payload to 8 subscribers, the proposed architecture exhibits extremely low latency and near-constant performance, whereas the traditional approach sees latency rise significantly due to PCIe bandwidth contention. In conclusion, RGAPS effectively overcomes the software and hardware bottlenecks of cross-node GPU communication, providing a highly scalable communication solution for large-scale distributed GPU applications requiring high bandwidth and low latency.

Keywords: Publish/Subscribe System, Graphics Processing Unit (GPU), Remote Direct Memory Access (RDMA), Shared Memory, Middleware, High-Performance Computing

doi:10.6342/NTU202600379

Contents

doi:10.6342/NTU202600379

doi:10.6342/NTU202600379

List of Figures

doi:10.6342/NTU202600379

List of Tables

doi:10.6342/NTU202600379

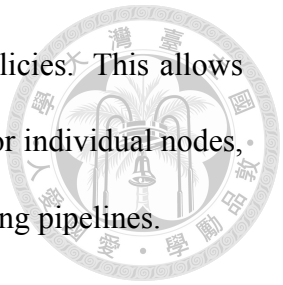
Chapter 1

Introduction

The publish/subscribe (pub/sub) pattern constitutes a fundamental messaging paradigm that decouples data producers ("publishers") from consumers ("subscribers") in terms of space, time, and synchronization. Instead of relying on direct addressing, publishers disseminate messages to specific "topics," allowing the middleware to abstract routing logic and distribute data asynchronously to all interested subscribers. This many-to-many communication model supports dynamic system composition and scalability.

To address diverse performance requirements—ranging from big data analytics to real-time control—academia and industry have specialized this abstraction through distinct design trade-offs. Apache Kafka [13] is widely regarded as a standard for the big data ecosystem, employing a "distributed commit log" architecture. It is optimized for high throughput, persistence, and stream replay capabilities through sequential disk I/O. However, its broker-centric design and reliance on disk operations often introduce latency overheads, limiting its suitability for scenarios requiring microsecond-level determinism. Robot Operating System 2 (ROS 2) leverages the Data Distribution Service (DDS) as its underlying connectivity standard to support modular robotics. It emphasizes real-time

doi:10.6342/NTU202600379

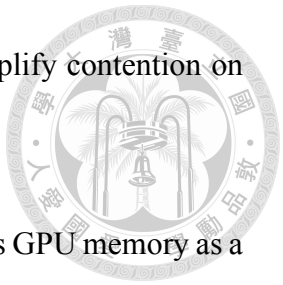


performance and provides configurable Quality of Service (QoS) policies. This allows developers to fine-tune parameters such as reliability and durability for individual nodes, facilitating the flexible composition of perception, control, and planning pipelines.

Eclipse Iceoryx [14, 16] targets deterministic, ultra-low latency Inter-Process Communication (IPC) within a single machine. By implementing a shared-memory management mechanism that exchanges data references (pointers) rather than copying payloads, it achieves "true zero-copy" transfer. This approach eliminates serialization overheads between kernel and user spaces, making it ideal for safety-critical domains like autonomous driving. Eclipse Zenoh [15, 17] addresses the connectivity challenges of Edge and Fog computing. It unifies traditional pub/sub pattern with distributed query mechanisms while maintaining minimal wire overhead. Its key feature, "location transparency," enables efficient routing across heterogeneous networks—from resource-constrained microcontrollers (MCUs) to cloud data centers—without exposing network topology complexity to the application.

While these systems provide strong abstractions for CPU-centric communication, many modern real-time and high-throughput pipelines are increasingly GPU-centric, where the dominant data objects are large tensors (e.g., images, feature maps, embeddings, or intermediate model outputs). In such pipelines, producers often generate results directly in GPU memory, and multiple downstream stages—also executed on GPUs—need to consume the same data concurrently. Under conventional pub/sub designs, the data plane typically assumes host-resident buffers, causing GPU-produced payloads to be staged through host memory. This introduces extra device-to-host transfers, possible serialization and buffering overheads, and in many deployments, repeated host-to-device transfers per subscriber. As message sizes scale from megabytes to gigabytes, these memory movements

doi:10.6342/NTU202600379

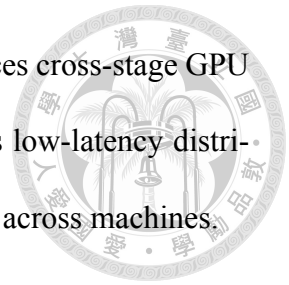


can become a primary source of end-to-end latency and can also amplify contention on shared I/O paths (e.g., PCIe), directly reducing pipeline parallelism.

To address this mismatch, GPU-Aware Pub/Sub (GAPS) [4] treats GPU memory as a first-class data plane: payloads remain in GPU memory whenever possible, and the messaging layer primarily distributes lightweight metadata to enable subscribers to access the same underlying GPU-resident buffer. In a single-machine setting, GAPS realizes this idea by enabling zero-copy fan-out through a shared GPU memory pool and inter-process GPU memory sharing, thereby avoiding per-subscriber payload duplication. However, GAPS is fundamentally constrained by host locality: publishers and subscribers must reside on the same machine (often on the same GPU), and each topic requires reserving GPU memory capacity for its shared pool. As topic counts grow or workloads become more diverse, these constraints can lead to (i) GPU memory pressure within a single machine and (ii) resource placement rigidity where publishers and subscribers are forced to compete for the same GPU compute resources.

Motivated by these limitations, this thesis presents a low-latency, RDMA-based framework named Remote GPU-Aware Pub/Sub (RGAPS). RGAPS extends the conceptual foundation of GAPS from a single host to cross-machine GPU-to-GPU dissemination. The key idea is to preserve the "publish once, fan-out many" property while allowing publishers and subscribers to be placed on different machines. RGAPS introduces a cross-machine data plane that couples (1) a GPU-accessible, RDMA-registered pinned host memory region on the publishing side with (2) an RDMA-writable pinned ring buffer on the subscribing side, enabling low-overhead transport between machines. After transport, the payload is materialized once into a shared GPU buffer on the subscriber machine, where it can be fanned out to multiple local subscribers without redundant copies. In do-

doi:10.6342/NTU202600379



ing so, RGAPS alleviates single-host GPU memory constraints, reduces cross-stage GPU resource contention through better placement flexibility, and enables low-latency distribution of large GPU-produced messages to multiple GPU consumers across machines.

RGAPS builds an RDMA-accelerated, distributed GPU publish/subscribe architecture that does not rely on GPUDirect RDMA, and instead uses pinned host memory as a staging layer for GPU–network–GPU transfers. Specifically, RGAPS assumes a two-host platform equipped with discrete GPUs, where GPU device memory is physically separate from host DRAM. On each host, the GPU communicates with the CPU and system memory through the PCIe interconnect, and the design explicitly targets environments in which GPU device memory cannot be directly exposed to the RNIC for network I/O. Consequently, RGAPS relies on host-resident pinned buffers at the network boundary: these buffers are DMA-capable for both GPU to and from host transfers and RDMA operations, enabling efficient staging without involving pageable memory. Across hosts, RGAPS assumes an RDMA-capable network and performs one-sided data movement between the two machines. The system is intended to run on a standard Linux software stack with user-space RDMA verbs support, where memory-registration primitives for pinned buffers are available to sustain high-throughput, low-latency DMA transfers.

In many emerging distributed AI systems, multiple GPU-accelerated agents cooperate to complete an end-to-end task, forming a pipeline of independent processes that continuously exchange intermediate results. Rather than interacting through small control messages, these agents often need to share large GPU-produced artifacts such as feature tensors, high-resolution image batches, point clouds, or 3D volumes. In this setting, the "interaction" between AI agents is fundamentally a high-frequency dataflow problem: one agent computes a large GPU-resident output and disseminates it to one or more

doi:10.6342/NTU202600379

downstream agents that immediately consume it for further GPU computation.

RGAPS is particularly well-suited for this class of multi-agent workloads when several conditions hold simultaneously. First, the exchanged artifacts are large GPU-resident payloads, typically on the order of 1 MB to 1 GB or more, where CPU-mediated transfers and redundant buffering introduce significant overhead. Second, the same payload must be consumed by multiple downstream agents, making fan-out a first-class requirement. Third, each participating agent performs substantial GPU computation, so the system should preserve GPU residency of payloads and avoid per-agent host–device copies that would inflate latency and amplify PCIe traffic. Overall, RGAPS fits AI-agent interactions that are best modeled as distributed, GPU-centric data pipelines: large intermediate outputs are produced on one GPU, delivered with low overhead, and then reused by multiple GPU-resident consumers for immediate follow-on computation.

doi:10.6342/NTU202600379

Chapter 2

Background

2.1 Two-Level Segregated Fit Memory Allocator (TLSF)

In environments where predictable timing is prioritized alongside average throughput, TLSF [6, 11] serves as a specialized solution for dynamic memory allocation in time-critical applications. Its key property is that allocation and deallocation have bounded, constant-time control overhead. This characteristic helps avoid the long tail-latency spikes commonly seen in general-purpose allocators when memory fragmentation increases.

In practice, TLSF achieves high performance in allocation and deallocation while keeping fragmentation under control. Internal fragmentation is typically low because TLSF fits requests more closely than coarse size-class schemes (e.g., pure power-of-two allocators), thereby reducing wasted space inside returned blocks. Furthermore, external fragmentation is generally mitigated through good fit quality and immediate coalescing, ensuring that free memory is less likely to be stranded as unusable fragments. While no variable-size allocator can eliminate fragmentation under all adversarial allocation patterns, TLSF is widely regarded as a strong engineering trade-off—offering deterministic

doi:10.6342/NTU202600379

latency with good average fragmentation behavior—and establishing it as a de facto standard for systems requiring strict timing guarantees.

In addition, TLSF can be extended to support concurrent allocation in shared-memory settings. By combining lightweight locking with careful handling of address translation across processes, multiple processes can safely allocate and free blocks from the same shared memory region. In practice, this enables a shared allocator to provide deterministic allocation behavior while remaining usable in multi-process runtimes, where the allocator ’s metadata and free lists must be consistently maintained under concurrent access.

2.2 GPU-Aware Publish/Subscribe (GAPS)

GPU-Aware Publish/Subscribe (GAPS) is a high-performance publish/subscribe messaging framework specifically designed to optimize intra-node communication for GPU-centric workloads. As illustrated in Figure 2.1, GAPS fundamentally decouples the data transmission path from the control signaling path to achieve true zero-copy communication between processes co-located on a single host.

The architectural core of GAPS relies on a dual shared-memory mechanism: a shared GPU memory region for high-volume payloads and a shared host memory region for lightweight metadata. The workflow operates as follows:

1. Direct GPU Write: The Publisher initiates the transmission by writing the payload directly into the pre-allocated shared GPU memory. This step ensures that large data structures (e.g., tensors or images) remain within the device memory, avoiding unnecessary device-to-host transfers.
2. Metadata Publication: Once the data is populated, the Publisher writes the corre-

doi:10.6342/NTU202600379

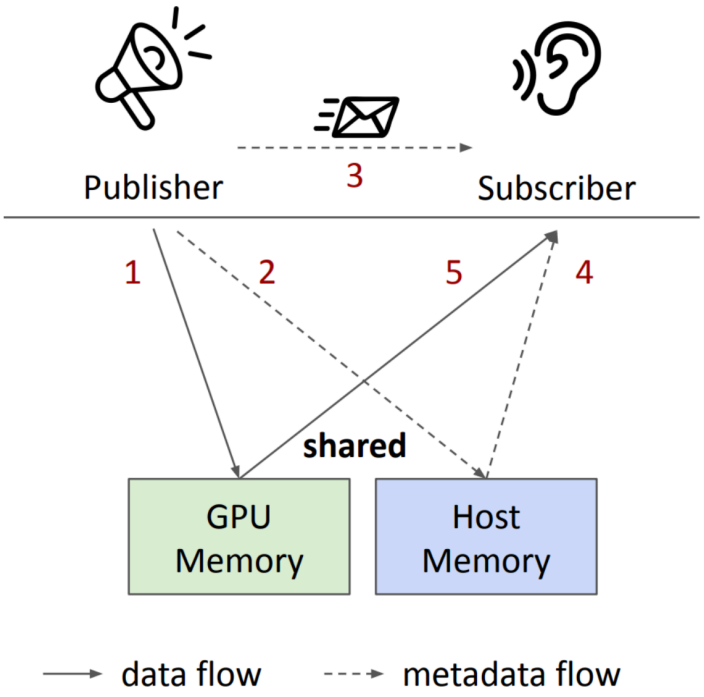


Figure 2.1: The architectural workflow of GAPS.

sponding metadata—specifically the payload size and the memory offset within the shared pool—into the shared host memory.

3. Notification: A lightweight notification signal is sent to the Subscriber via the underlying middleware (e.g., Iceoryx) to trigger the consumption process.
4. Metadata Retrieval: Upon receiving the notification, the Subscriber reads the metadata from the shared host memory to resolve the physical location of the payload.
5. Zero-Copy Fetch: Equipped with the correct offset and size, the Subscriber accesses the data directly from the shared GPU memory.

By leveraging this unified memory addressing scheme, GAPS ensures that the CPU is never involved in the actual movement of the payload. The data transfer complexity remains $O(1)$ regardless of the message size, providing the deterministic low latency required for real-time applications. Consequently, to mitigate the latency and bandwidth bottlenecks caused by redundant host-to-device memory copies and the inherent over-

doi:10.6342/NTU202600379

heads of traditional Inter-Process Communication (IPC) mechanisms, we utilize GAPS as the foundational architecture for our intra-node communication.

2.3 CUDA Host Pinned Memory

In the realm of high-performance GPU workloads, the transfer efficiency across the host-device interface frequently emerges as a primary performance constraint. A critical factor influencing this transfer efficiency is the type of host memory utilized—specifically, the distinction between standard Pageable Memory and Pinned (Page-locked) Memory. To elucidate the latency overhead observed in standard data transfers, this section contrasts the data paths involved in utilizing Pageable Memory (default *malloc*) versus Pinned Memory (*cudaMallocHost*) [9, 10].

Pageable memory relies on virtual addressing where physical pages can be swapped out by the operating system. Since the GPU's DMA engine requires physically contiguous and resident memory addresses to perform transfers, it cannot access pageable user-space memory directly. Consequently, the CUDA driver must intervene by employing a Kernel Driver Staging Buffer—a temporary, page-locked region in kernel space. This necessitates a two-step transaction process:

1. DMA Transfer: Data is first transferred from the GPU device memory to the kernel staging buffer via the DMA engine.

2. CPU Copy: Data is then transferred by the CPU from the kernel buffer into the target user-space region via a mandatory memory copy. This architecture incurs a "double-copy" penalty and consumes CPU cycles, thereby increasing the end-to-

doi:10.6342/NTU202600379

end latency.

Conversely, the Pinned (Page-locked) Memory mechanism offers a more efficient alternative. By allocating host memory using *cudaMallocHost*, the operating system locks the physical pages in RAM, preventing them from being swapped out and ensuring stable physical addresses. As a result, the GPU's DMA engine can perform a direct transfer from the device memory to the user-space buffer, bypassing the kernel staging buffer and eliminating the redundant CPU-mediated copy. This optimization not only maximizes PCIe bandwidth utilization but also significantly reduces host-side CPU overhead, making it critical for latency-sensitive applications.

The effectiveness of pinned versus pageable host memory is strongly influenced by the available PCIe bandwidth. To quantify this effect, we compare device-to-host transfer performance on two GPU platforms with different interconnects, as shown in Figure 2.2. On the RTX 3090 system with PCIe 4.0 $\times$ 16, using pinned memory improves device-to-host transfer latency by approximately 1.8–$2.0\times$ over pageable memory for data sizes from 4 MB to 1 GB. This behavior is consistent with the standard implementation of pageable transfers, where data is first DMA-transferred from device memory to a driver-managed pinned staging buffer in kernel space and then copied by the CPU into the user-space buffer. In contrast, pinned memory allows an effectively zero-copy DMA path directly into a user-visible buffer, thereby eliminating this additional host-side copy.

While the primary advantage of pinned memory lies in removing this host-side copy overhead, the realizable speedup is strictly bounded by the transmission capabilities of the hardware interface. Consistent with Amdahl's Law, optimizing a non-dominant copy stage provides negligible gains. This is clearly observed on the RTX 3090 system with

doi:10.6342/NTU202600379

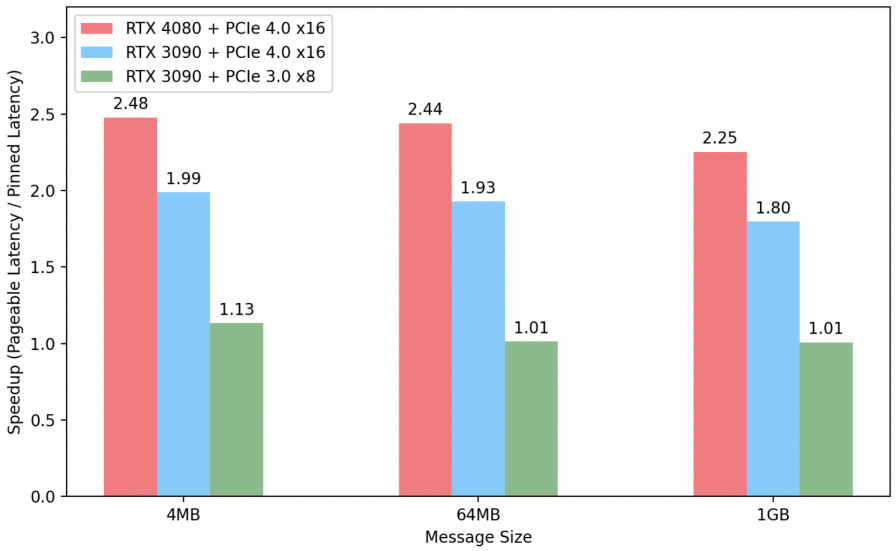
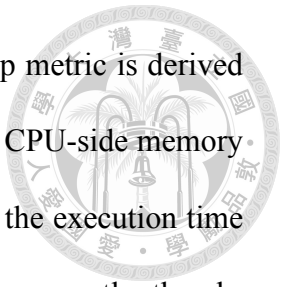


Figure 2.2: Impact of PCIe bandwidth and GPU micro-architecture on Pinned Memory speedup.

PCIe 3.0 $\times$ 8, where the speedup of pinned over pageable memory is only about 1.0–1.1$\times$ across the same data sizes, indicating that transfers are already PCIe-bound. In this lower-bandwidth configuration, the PCIe DMA dominates the overall latency, the extra kernel-to-user copy in the pageable path is effectively hidden under the PCIe transfer time, and the measured speedup consequently diminishes to $\approx 1.0\times$. This stark contrast demonstrates that software-level zero-copy mechanisms are only beneficial when the underlying interconnect provides sufficient bandwidth so that transmission latency does not mask the effect of eliminating the host-side copy.

While PCIe bandwidth establishes the theoretical upper bound for data transfer rates, our results indicate that the micro-architectural implementation—specifically the efficiency of the GPU's on-chip DMA Copy Engines—plays a pivotal role in realizing this potential. Under identical PCIe 4.0 $\times$ 16 constraints, the RTX 4080 (Ada Lovelace) exhibits a superior speedup factor compared to the RTX 3090 (Ampere), attributing to the Ada architecture's enhanced ability to reduce transaction initiation overhead and saturate the interconnect. Furthermore, it is critical to note that the magnitude of this speedup is

doi:10.6342/NTU202600379

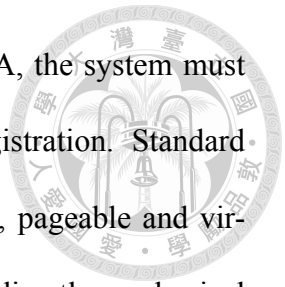


also intrinsically linked to host DRAM bandwidth. Since the speedup metric is derived relative to pageable memory performance—which is bottlenecked by CPU-side memory copy latency—variations in host memory throughput directly impact the execution time of the pageable transfer, thereby influencing the calculated gain. Consequently, the observed performance improvement reflects a composite effect of the GPU's superior DMA scheduling and the specific latency characteristics of the host memory subsystem.

2.4 Remote Direct Memory Access (RDMA)

RDMA fundamentally mitigates the performance bottlenecks inherent in traditional TCP/IP software stacks by offloading transport protocols, reliability, and flow control mechanisms directly to the Network Interface Card (RNIC). This architecture shortens the software data path, achieving microsecond-scale latencies and wire-speed throughput. First, through kernel bypass and zero-copy networking, RDMA transfers data directly between the registered memories of applications. This design eliminates redundant data copying between kernel and user space, minimizes cache pollution, and avoids expensive system calls and context switches, thereby significantly increasing per-core I/O capacity while lowering CPU utilization. Second, RDMA's unique one-sided operations (READ/WRITE) enable the initiator to access remote buffers without the involvement of the remote CPU, a feature particularly valuable for highly concurrent, latency-critical data paths. Finally, by relying on hardware-based Reliable Connection (RC) [8] and bypassing the operating system scheduler in the data plane, RDMA ensures deterministic performance with extremely low jitter and stable tail latency, which are critical for tightly synchronized high-performance computing environments.

doi:10.6342/NTU202600379



To enable the zero-copy and CPU-bypass capabilities of RDMA, the system must first undergo a mandatory preparation phase known as Memory Registration. Standard user-space memory allocated by the operating system is, by default, pageable and virtual. The RDMA Network Interface Card (RNIC), however, operates directly on physical addresses to perform DMA (Direct Memory Access) and requires the target memory to remain resident in physical RAM throughout the transaction.

The transformation from a standard buffer to an RDMA-accessible region is mediated by the *ibv_reg_mr* verb [1, 5]. This operation performs two critical functions [3]:

1. Memory Pinning: It instructs the operating system kernel to "pin" (page-lock) the specified virtual memory range, preventing it from being swapped out to the disk. This guarantees that the physical data path remains stable.
2. Address Translation: It creates a mapping between the virtual addresses and their corresponding physical addresses, which is then loaded into the RNIC's Memory Management Unit (MMU).

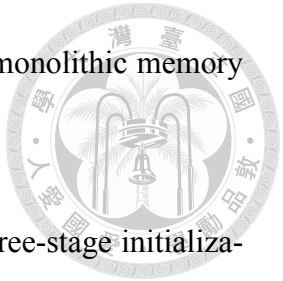
Once the memory region is successfully registered, the RNIC obtains the authority to perform direct DMA read or write operations on this region. Consequently, data can flow directly between the RNIC and the pinned memory region, completely bypassing the host CPU and the operating system kernel during the actual data transfer.

2.5 Shared CUDA-RDMA Pinned Memory

To bridge the architectural gap between GPU computation and network transmission, we introduce a unified memory management abstraction termed Shared CUDA-RDMA Pinned Memory. This mechanism consolidates the requirements of inter-process commu-

doi:10.6342/NTU202600379

nication (IPC), GPU access, and RDMA transmission into a single, monolithic memory region.



The construction of this specialized memory region follows a three-stage initialization process:

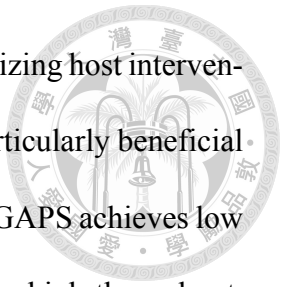
1. Shared Memory Allocation: First, the host allocates a standard pageable memory segment using POSIX shared memory APIs (*shm_open*, *ftruncate*, and *mmap*). This step ensures that the memory is addressable by multiple processes.
2. CUDA Pinning: The allocated range is then promoted to CUDA-accessible pinned memory via *cudaHostRegister*. Unlike *cudaMallocHost* which allocates new memory, *cudaHostRegister* page-locks an existing virtual address range and maps it into the GPU's address space, enabling the GPU DMA engine to access it directly.
3. RDMA Registration: Finally, the same memory range acts as the target for *ibv_reg_mr*, registering it with the RNIC. This operation pins the pages (if not already pinned) and generates the necessary translation entries for the RDMA hardware.

The result region is a dual-accessible DMA buffer. This region resides in the host's physical memory but exposes direct access interfaces to both the GPU and the RDMA NIC. Consequently, data can flow seamlessly from the GPU's device memory to this host buffer via PCIe, and immediately out to the network via the RNIC, without any CPU-mediated memory copying or context switching.

2.6 Related Work

GAPS [4] presents a GPU-aware publish/subscribe design tailored for the low-latency dissemination of large messages in GPU-centric real-time pipelines. Its primary contribu-

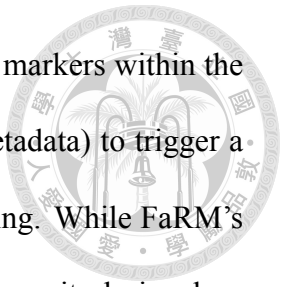
doi:10.6342/NTU202600379



tion is establishing a "GPU-native" data path on a single host by minimizing host intervention and eliminating redundant data copies—optimizations that are particularly beneficial when multiple subscribers consume the same payload. Consequently, GAPS achieves low and stable latency for intra-node fan-out, making it highly suitable for high-throughput, large-payload GPU workloads. Despite these advantages, GAPS is fundamentally constrained by single-host locality. Its design assumes co-location of publishers and subscribers, limiting its applicability to distributed pipelines where producers and consumers reside on different nodes. Furthermore, GAPS typically provisions a dedicated shared GPU memory pool per topic; as the number of topics grows, the total memory footprint increases, potentially creating a bottleneck given finite GPU memory capacity. Finally, concentrating computation and communication within a single host (and often the same GPU) can intensify resource contention—such as for memory bandwidth and copy engines—thereby degrading throughput scalability and latency predictability as workloads scale.

Furthermore, by harnessing RDMA capabilities, FaRM [2] operates as a main-memory distributed system that exposes a unified address space across nodes, offering superior performance compared to traditional TCP/IP architectures. Beyond one-sided RDMA Reads, FaRM implements a lightweight message-passing primitive using RDMA Writes into a receiver-resident circular ring buffer. In this design, the receiver polls the head position to detect message arrival, interpreting a non-zero value as the length of a new message. However, because correctness relies on the "non-zero means valid" invariant, the receiver must explicitly zero out the buffer region after processing and before advancing the head pointer. This requirement introduces additional memory-write overhead on the critical path. An alternative optimization—often adopted to avoid this cost—is to utilize RDMA

doi:10.6342/NTU202600379



Write with Immediate for notification. Instead of embedding validity markers within the payload buffer, the sender attaches a small immediate value (e.g., metadata) to trigger a completion event at the receiver, eliminating the need for buffer zeroing. While FaRM's ring-buffer mechanism is effective for CPU-centric remote memory access, its design does not target topic-based publish/subscribe fan-out, nor does it address the specific requirements of GPU-resident payload distribution.

KafkaDirect [12] is an RDMA-accelerated variant of Apache Kafka designed to address the bottlenecks of Kafka's TCP/IP data path—specifically OS involvement and redundant memory copying. It introduces RDMA into three critical stages: production, replication, and consumption. Its core principle exploits one-sided RDMA operations to transfer record payloads without intermediate copies. On the production path, producers write record batches directly into broker-managed topic-partition (TP) files and use RDMA Write with Immediate to notify brokers; multi-producer coordination is handled via RDMA atomic operations. For replication, KafkaDirect adopts a push-based design where leaders directly write updates to followers, bypassing the conventional pull-based fetch loop. While KafkaDirect demonstrates substantial performance gains for log-structured messaging, its architecture is fundamentally centered on broker-managed log semantics (e.g., persistence, offsets, and replication). In contrast, our work focuses on GPU-oriented publish/subscribe, where the primary objective is the efficient dissemination of large payloads directly into GPU memory for immediate execution, rather than broker-side storage management or replication protocols.

doi:10.6342/NTU202600379

Chapter 3

Methodology

3.1 Overview of Data Plane

This study proposes a high-performance framework designed to extend the capabilities of intra-node GPU messaging to distributed clusters. The core philosophy of our design is to minimize data movement overheads and eliminate CPU intervention throughout the transmission pipeline. As illustrated in the architectural overview (Figure 3.1), the end-to-end data plane is structurally divided into three distinct paths, creating a streamlined "shortcut" from the publisher's GPU memory to the subscriber's view.

The entire data plane is organized into three distinct data paths:

Data Path 1: Intra-node Aggregation. On the publishing node, the system utilizes a shared CUDA-RDMA pinned memory pool as the staging ground for outgoing data. Multiple publishers can concurrently allocate memory segments within this pool using the TLSF algorithm and populate them with data. This design allows the system to aggregate messages from various GPU sources into a unified, network-ready memory region without requiring intermediate kernel buffers.

doi:10.6342/NTU202600379

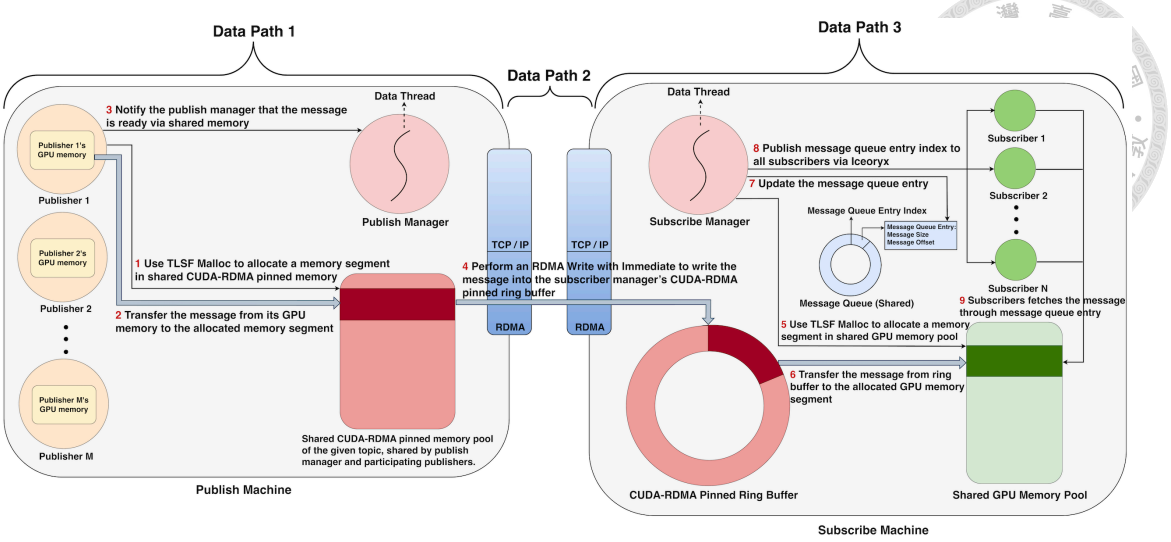


Figure 3.1: Architectural overview of the entire data plane.

Data Path 2: Inter-node Transmission. The bridge between the publish and subscribe machines is established via RDMA. Once the message is ready, the publish manager initiates an RDMA Write with Immediate operation. This mechanism directly transfers the message from the publish manager's pinned memory into the subscribe manager's pre-allocated CUDA-RDMA pinned ring buffer on the subscribe machine. Crucially, this transmission bypasses the operating system kernels on both sides, ensuring high bandwidth utilization and deterministic latency.

Data Path 3: Local Dissemination. Upon arrival at the subscribe machine, the data must be efficiently distributed to multiple consumers. Instead of a naive approach where the subscribe manager publishes the full message via standard middleware, then forcing every subscriber to redundantly copy the data into its private GPU memory, the subscribe manager performs a single transfer from the ring buffer to a shared GPU memory pool. The manager then writes the message offset and size into a specific message queue entry, and publishes the lightweight index of this entry via Iceoryx. Subscribers use this index to retrieve the entry and calculate the address of the message in the shared pool. This approach ensures that the system's performance remains stable regardless of the number

doi:10.6342/NTU202600379

of local subscribers.

By orchestrating these three paths, the proposed architecture establishes a highly efficient pipeline that effectively hides the complexity of network transmission from the application layer while significantly reducing the latency overheads typically associated with traditional distributed communication.

3.2 Control Plane Architecture and Initialization Process

Complementing the high-speed data plane, the control plane is responsible for the publisher and subscriber join processes, resource allocation, and membership management. Unlike the data plane which leverages RDMA for throughput, the control plane operates over standard TCP/IP for inter-node coordination and Unix Domain Socket (UDS) for local process communication, ensuring reliable setup of the distributed environment.

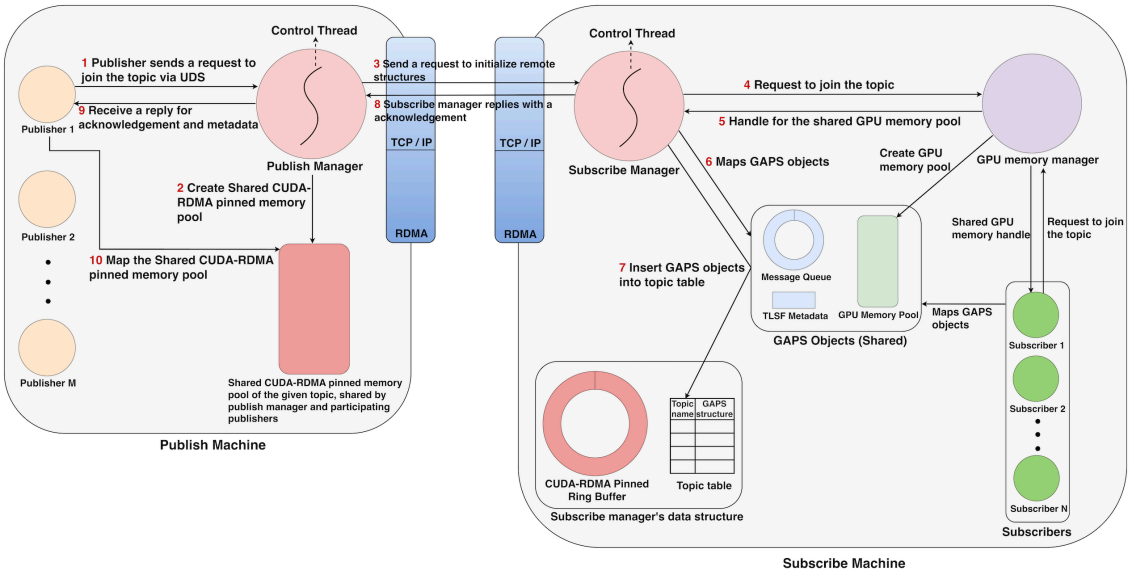


Figure 3.2: Control plane architecture and initialization process.

Topic initialization and publisher/subscriber registration. As depicted in the left panel of the architecture diagram (Figure 3.2), the initialization process involves a coor-

doi:10.6342/NTU202600379

dinated sequence across the cluster:

1. Request and allocation: The process typically begins when a publisher sends a request to join a specific topic via Unix Domain Socket. Upon validating this request, the publish manager creates the shared CUDA-RDMA pinned memory pool—the critical resource for Data Path 1.

2. Remote coordination: The publish manager then transmits an initialization request over TCP/IP to the remote subscribe manager, triggering the setup of per-topic infrastructure.

3. Resource creation: Upon receiving the request, the subscribe manager initiates the resource setup sequence. First, it requests access to the shared GPU memory pool from the GPU memory manager, which allocates the pool and returns a shareable handle via Unix Domain Socket for the subscribe manager to map. Simultaneously, it handles the GAPS objects (POSIX shared memory region containing the TLSF metadata and message queue). This step follows a "first-arrival" initialization policy: whichever entity (the subscribe manager or a local subscriber) accesses the topic first is responsible for creating these GAPS objects. In this flow, if the objects do not exist, the subscribe manager creates them; otherwise, it simply maps the existing ones.

4. Topic registration: Distinctively, regardless of who initialized the shared resources, the subscribe manager performs a unique administrative action: it inserts the topic information into the local topic table. This step formally registers the topic within the node's management structure.

5. Finalization: An acknowledgment is propagated back to the publish manager and

doi:10.6342/NTU202600379

subsequently to the publisher, enabling the publisher to begin data transmission.

Subscriber registration and dynamic coupling. On the subscriber node (right panel), the registration process is designed to be asynchronous. When a new subscriber joins, it executes the same resource setup sequence. It requests the shared GPU memory pool handle from the GPU memory manager and maps it. Regarding the GAPS objects, the subscriber also adheres to the "first-arrival" policy: if it is the first entity to join the topic (preceding even the subscribe manager's remote initialization), it creates these POSIX shared memory region. Conversely, if the objects have already been established (by the subscribe manager or another subscriber), it simply maps them to participate in the communication. This design allows for flexible system bootstrapping where the order of process arrival does not hinder connectivity.

3.3 Data Path 1

To quantify the architectural advantages of the proposed framework, we perform a comparative analysis of the intra-node data aggregation phase. Figure 3.3 schematically contrasts Data Path 1 of our proposed solution against a traditional baseline.

Pageable-UDS (baseline). As illustrated in the right panel, the traditional approach relies on standard IPC mechanisms and pageable memory. The transmission follows a pattern involving sequential memory copies:

1. Device-to-host copy: The publisher first performs a *cudaMemcpy* to transfer the message from its GPU memory to a private, pageable host memory buffer.

2. IPC transmission: To forward the data to the publish manager, the publisher utilizes

doi:10.6342/NTU202600379

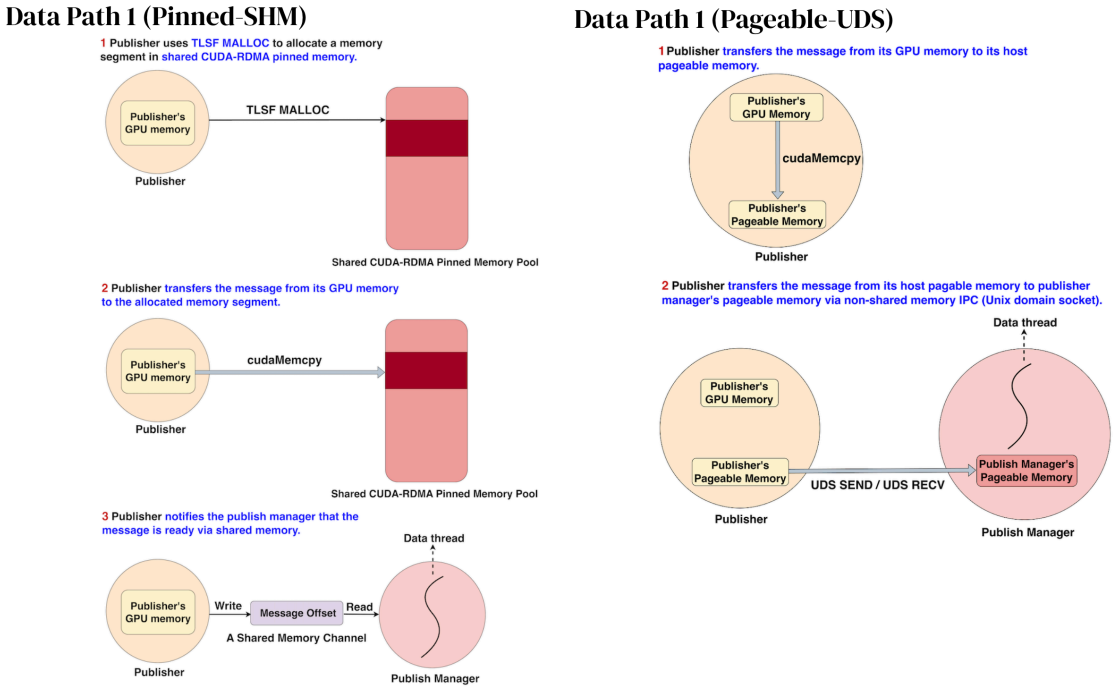


Figure 3.3: Comparative analysis of Data Path 1: Pinned-SHM vs. Pageable-UDS.

a Unix Domain Socket. This operation triggers a system call where the kernel copies the data from the publisher's user space to a kernel buffer, and subsequently to the publish manager's pageable memory.

This architecture incurs significant overhead due to the involvement of the OS kernel and the serialized memory copies required to move data across process boundaries.

Pinned-SHM (proposed). In contrast, the left panel depicts the proposed Pinned-SHM mechanism, which leverages the shared CUDA-RDMA pinned memory pool to eliminate redundant overheads. The workflow is streamlined as follows:

1. Zero-copy allocation: The publisher employs the TLSF algorithm to allocate a memory segment directly within the shared pinned memory pool. Since this pool is pre-mapped into the address spaces of both the publisher and the publish manager, no physical data movement is required to the shared memory region.

doi:10.6342/NTU202600379

2. Direct placement: The publisher performs a single *cudaMemcpy* to transfer the message from its GPU memory directly into the allocated shared segment.

3. Lightweight notification: Instead of sending the full message via the standard IPC method, the publisher simply writes the message offset and size into a shared memory channel to notify the publish manager.

By adopting this design, Pinned-SHM ensures that the data is written to host memory exactly once and is immediately visible to the publish manager. This significantly reduces the kernel-user context switches and memory copy overheads, streamlining the handover process to be independent of the message size.

3.4 Design of RDMA Ring Buffer

Before elucidating the inter-node transmission mechanisms in Data Path 2, it is imperative to rigorously define the underlying data structure: the RDMA Ring Buffer. This circular buffer resides within the subscribe manager's memory domain but is actively written to by the publish manager via RDMA Write. To support the continuous streaming of variable-sized messages across this cross-node channel without data corruption, we introduce a foundational state-checking mechanism termed "Exceed".

Fundamentally, the ring buffer operates on a circular structure. To prevent buffer overflows where a new write operation inadvertently overwrites unconsumed data, the system must evaluate the relationship between the publish manager's write cursor (X)—calculated as the current write offset plus the incoming message size—and the subscribe manager's read cursor (Y). We classify this evaluation into two distinct geometrical sce-

doi:10.6342/NTU202600379

narios, as illustrated in Figure 3.4:

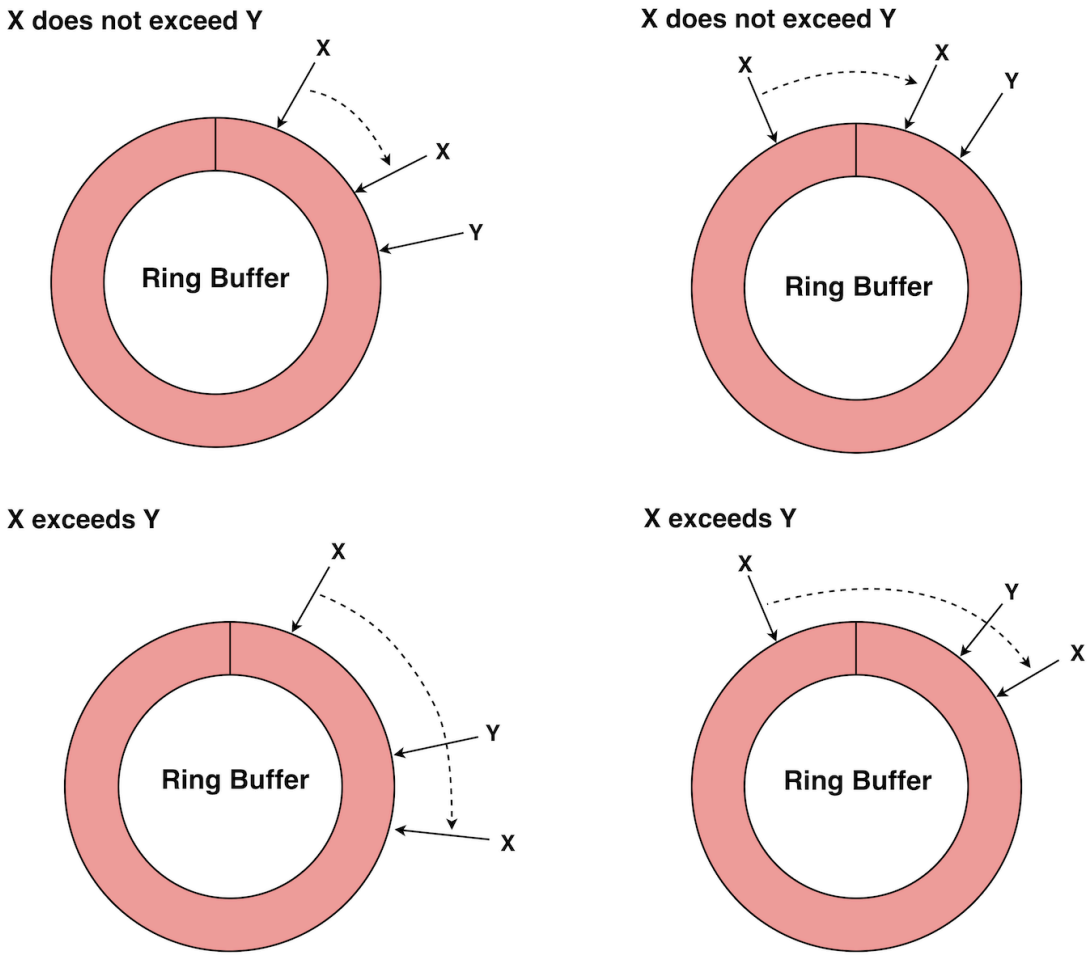
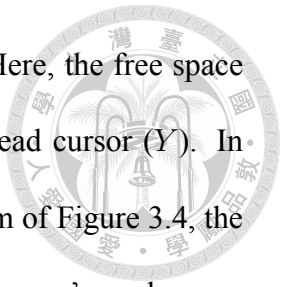


Figure 3.4: The "Exceed" state-checking mechanism in the RDMA Ring Buffer.

Case 1: Non-wrapped scenario (linear gap). In this layout, the free space available for writing is contiguous within the linear address space of the buffer. The "Exceed" check is straightforward. In the safe state (X does not exceed Y), as shown in the top-left diagram of Figure 3.4, if the publish manager's write cursor (X) falls short of the subscribe manager's read cursor (Y), the write is deemed safe. Conversely, in the overflow state (X exceeds Y), corresponding to the bottom-left diagram, if the message size extends X beyond Y, an "Exceed" condition is triggered. This indicates insufficient contiguous space, necessitating a wait state or a reset of the pointers.

Case 2: Wrapped scenario (circular gap). This layout addresses the complexity when

doi:10.6342/NTU202600379



the active data region wraps around the physical end of the buffer. Here, the free space is logically "sandwiched" between the current write offset and the read cursor (Y). In the safe state (X does not exceed Y), depicted in the top-right diagram of Figure 3.4, the new message fits within the gap without overtaking the subscribe manager's read cursor (Y), thus maintaining circular integrity. However, in the overflow state (X exceeds Y), shown in the bottom-right diagram, the projected cursor X advances past the limit Y. This signifies a collision where the new data would overwrite unread frames, and the "Exceed" flag is raised to prevent this violation.

By rigorously applying this "Exceed" logic before every RDMA Write operation, the system ensures data consistency and strictly prevents buffer overflow for variable-sized payloads.

To orchestrate high-throughput transmission without incurring the latency penalties of frequent network round-trips, the system employs a mechanism based on "conservative writes" and "lazy synchronization."

Pointer architecture and ownership. As illustrated in Figure 3.5, the state of the ring buffer is managed through three distinct pointers distributed across the publish manager and subscribe manager:

1. *Tail*: The write cursor. Maintained locally by the publish manager, it indicates the offset where the next message will be written.

2. *Head*: The true read cursor. Maintained by the subscribe manager, it tracks the actual consumption progress on the subscribe manager side.

3. *Delayed Head*: A cached read cursor. This is a local shadow copy of the remote

doi:10.6342/NTU202600379

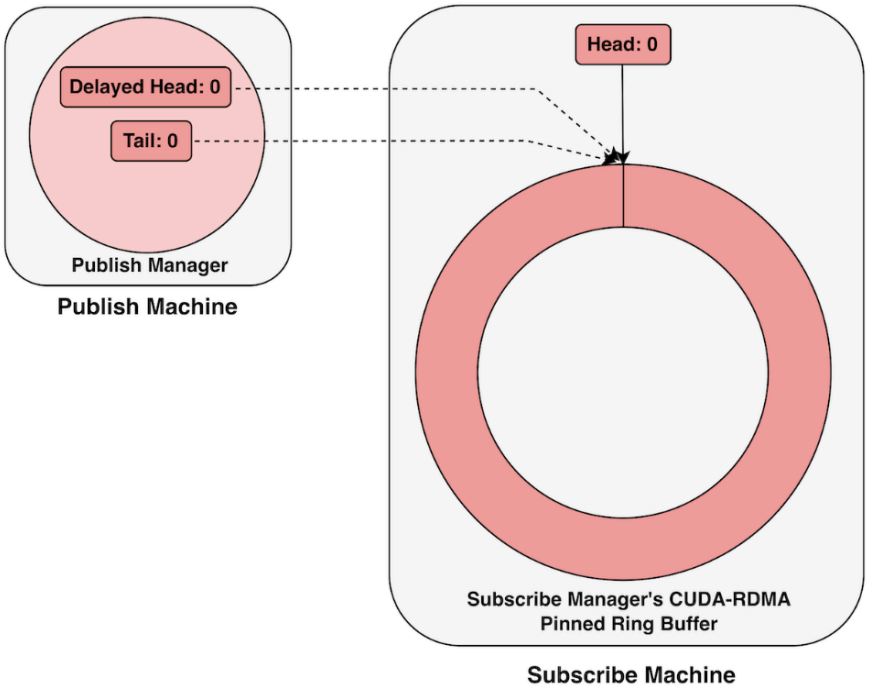


Figure 3.5: Initial state of the RDMA Ring Buffer.

Head residing on the publish manager's side. It represents a "conservative" view of the subscribe manager's progress.

The "conservative write" strategy. The core principle of this design is that the publish manager makes writing decisions solely based on local information. Before initiating an RDMA Write, it performs an "Exceed" check using its local pointers:

Constraint: The current *Tail* + *Message Size* must not exceed the *Delayed Head*.

Since the *Delayed Head* is always less than or equal to the true remote *Head* (in terms of free space progression), satisfying this local constraint guarantees that valid space exists in the remote buffer. This effectively eliminates the need to query the remote *Head* via RDMA Read or update the *Delayed Head* via RDMA Write for every message.

Watermark-based lazy synchronization. To keep the *Delayed Head* updated, we introduce a watermark-based update mechanism. The subscribe manager does not syn-

doi:10.6342/NTU202600379

chronize every *Head* advancement. Instead, it accumulates the size of consumed messages. When this accumulated value reaches a configurable watermark, the subscribe manager initiates a one-sided RDMA Write to update the *Delayed Head* on the publish manager with its current, authoritative *Head* value. This batching strategy significantly reduces control plane traffic.

Initial state verification. At the system initialization, as shown in the figure, all pointers (*Tail*, *Head*, *Delayed Head*) are reset to zero. To ensure the logical validity of this distributed state, we verify three circular invariants:

1. Production integrity: *Tail* does not exceed *Delayed Head* (The writer never overwrites the unacknowledged reader boundary).

2. Consumption integrity: *Head* does not exceed *Tail* (The reader never overtakes the writer).

3. Lag integrity: *Delayed Head* does not exceed *Head* (The cached view never "leads" the truth).

The satisfaction of these conditions confirms that the system is in a consistent state, ready to commence the cycle of conservative writes and batched updates.

Following the system initialization, the data plane enters the production phase. In this stage, the publish manager actively streams variable-sized messages into the subscribe manager's ring buffer using a sequence of RDMA Write operations.

Message encapsulation and layout. To ensure that the subscriber can correctly parse and verify the integrity of each incoming frame, the system enforces a strict encapsulation

doi:10.6342/NTU202600379

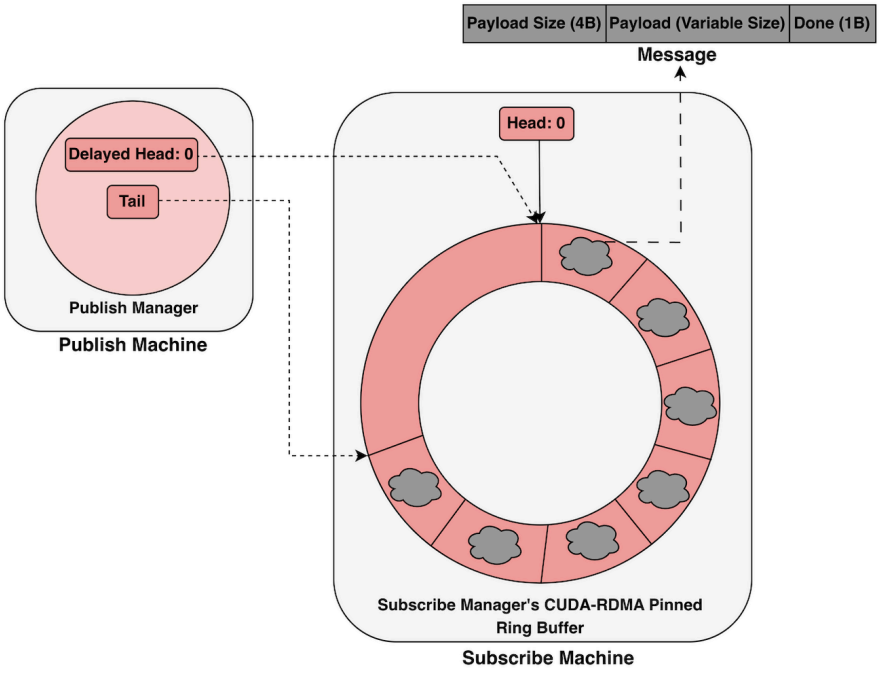


Figure 3.6: Multiple messages written to the RDMA Ring Buffer.

protocol. As depicted in Figure 3.6, every message written to the ring buffer is structured with specific metadata:

1. Payload Size Header: Preceding the actual data, a 4-byte integer indicates the length of the variable-size payload. This allows the subscribe manager to determine exactly how many bytes to read for the current message.

2. Message Payload: The actual application data with variable size.

3. Done Flag: Appended at the very end of the message, this 1-byte flag serves as a completion signal. Since RDMA Writes are not atomic across large payloads, the subscribe manager polls this flag to verify that the entire message has fully landed in its memory.

Dynamic state verification. During this production process, the state of the distributed pointers evolves. Specifically, the publish manager advances its local *Tail* pointer as

doi:10.6342/NTU202600379

it writes new messages, while the *Head* and *Delayed Head* remain temporarily stationary (until a synchronization event occurs). Consequently, the validation logic simplifies. Since the read cursors do not move, the invariants regarding lag and consumption integrity naturally hold. The system's focus therefore narrows exclusively to production integrity: the publish manager must ensure that the updated *Tail* does not exceed the *Delayed Head*.

This constraint acts as the primary guardrail during the write phase. By rigorously enforcing that the writer's *Tail* never overtakes the conservative estimate of the reader's *Delayed Head*, the system prevents the overwriting of unacknowledged messages, ensuring safe and continuous transmission.

Parallel to the production process, the subscribe manager continuously monitors the ring buffer to process incoming messages. This consumption phase involves a precise sequence of memory retrieval, pointer advancement, and crucial buffer maintenance operations.

Consumption workflow and memory sanitization. The retrieval process is driven by the local *Head* pointer. As illustrated in Figure 3.7, the subscribe manager accesses the memory address pointed to by *Head* and executes the following opeations:

1. Header parsing: The subscribe manager first reads the 4-byte payload size. If this value is non-zero, it indicates the presence of a valid, unconsumed message.

2. Payload processing: Based on the size header, the manager fetches the full payload and facilitates the downstream processing.

3. Pointer advancement: Upon successful processing, the *Head* is advanced by the total size of the message to point to the next potential message.

doi:10.6342/NTU202600379

4. Memory zeroing: A vital step in our design is the immediate sanitization of the consumed memory region. The subscribe manager must explicitly overwrite the consumed message slots with zeros. This prevents "phantom messages"—residual non-zero data from previous cycles—from being misinterpreted as valid headers when the buffer pointers wrap around in future iterations.

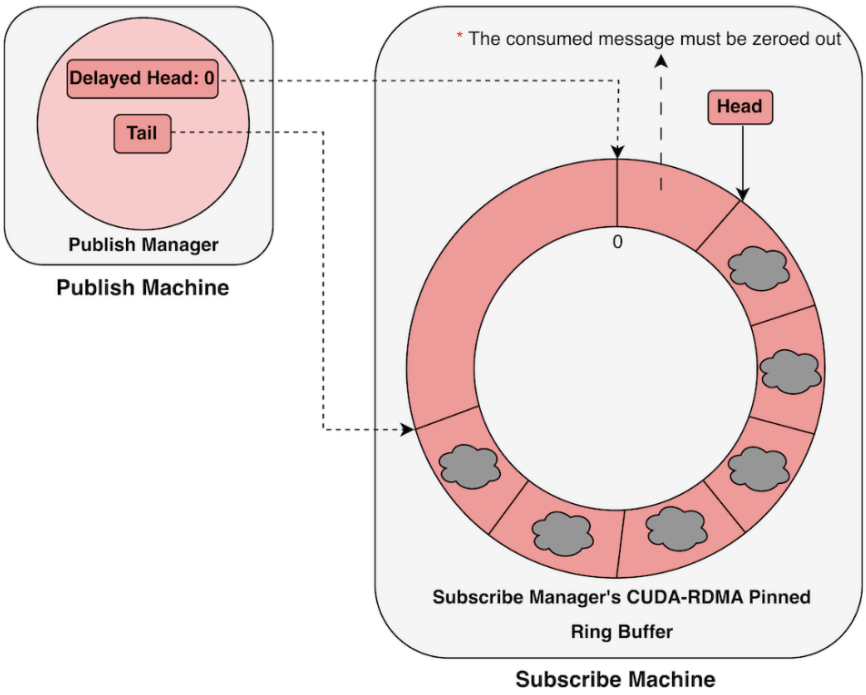


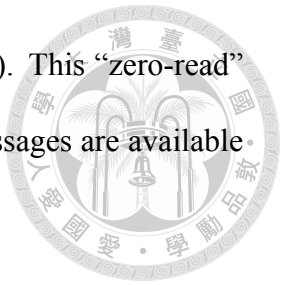
Figure 3.7: Consumption of messages from the RDMA Ring Buffer.

Dynamic state verification. In this phase, the system state evolves solely through the movement of the *Head* pointer, while the *Tail* and *Delayed Head* remain static relative to the consumption action. Consequently, the verification logic focuses exclusively on consumption integrity: the subscribe manager must ensure that the *Head* does not exceed the *Tail*.

In practice, this constraint is enforced implicitly through the data structure itself rather than explicit pointer comparison. When the subscribe manager reads the memory at the current *Head*, encountering a payload size of 0 signifies that it has caught up to the writer

doi:10.6342/NTU202600379

(i.e., the memory has not yet been written to by the publish manager). This "zero-read" condition acts as the natural boundary, confirming that no further messages are available and ensuring the reader never overtakes the writer.



To sustain continuous transmission, the publish manager must eventually become aware that the subscribe manager has consumed data, thereby freeing up space in the ring buffer. This feedback loop is established through the synchronization of the *Delayed Head*.

Watermark-based update mechanism. Minimizing control plane overhead is paramount for high performance. Consequently, the subscribe manager does not synchronize its progress for every single message processed. Instead, it employs a configurable watermark strategy. The manager tracks the accumulated size of consumed messages. Only when this accumulation exceeds a pre-defined threshold—configurable to a fraction of the buffer size (e.g., 1/2, 1/4, or 1/8)—does the system trigger an update. Alternatively, to ensure the publish manager maintains the most precise view of the subscribe manager's actual read position, this watermark can be configured to zero, enforcing immediate updates after every message.

Remote synchronization via RDMA Write. Once the watermark is reached, as illustrated in Figure 3.8, the subscribe manager initiates a one-sided RDMA Write operation. This transmission updates the publish manager's local *Delayed Head* pointer with the current authoritative value of the *Head*. This update effectively "releases" the consumed memory space from the publish manager's perspective, allowing the *Tail* to advance further in subsequent write cycles.

Dynamic state verification. In this phase, the state change involves advancing the *Delayed Head* forward to catch up with the *Head*. Therefore, the validation logic centers

doi:10.6342/NTU202600379

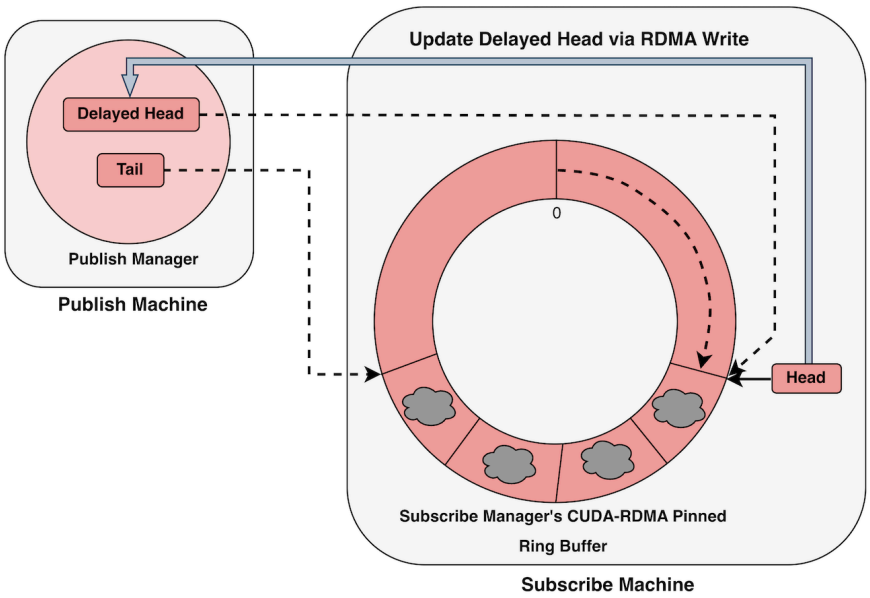


Figure 3.8: Updating the Publish Manager's Delayed Head.

specifically on the lag integrity: the system must ensure that the *Delayed Head* does not exceed the *Head*.

This invariant is guaranteed by the causality of the update mechanism. Since the *Delayed Head* is updated using a snapshot of the *Head*, and the authoritative *Head* only moves forward (monotonically increases) as messages are consumed, the cached *Delayed Head* will always represent a value less than or equal to the current *Head*. It serves as a strictly conservative lower bound of the subscribe manager's progress, ensuring it never leads the actual read cursor.

While the "zero-out" operation ensures the logical correctness of the consumption phase, it introduces a computational cost that warrants detailed analysis. To quantify this overhead, we conducted a micro-benchmark comparing the latency of locally zeroing out pinned memory against the latency of performing an RDMA Write of the same size.

Latency characteristics and bandwidth saturation. Figure 3.9 presents the ratio of memory zeroing latency to RDMA Write latency across varying message sizes. A distinct

doi:10.6342/NTU202600379

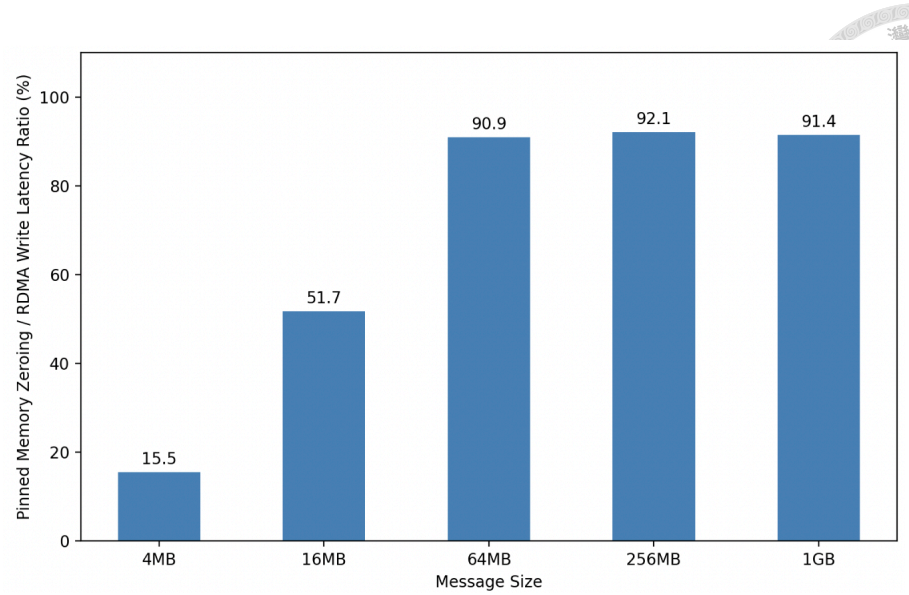


Figure 3.9: Overhead of Pinned Memory Zero-out relative to RDMA Write latency.

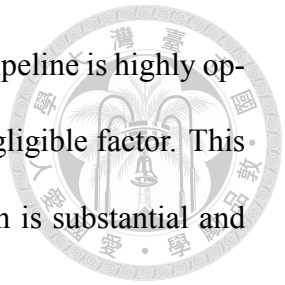
trend emerges as the payload size increases. For large payloads (> 64 MB), the ratio rises sharply and plateaus above 90%. For instance, when handling 1 GB messages, the duration necessary to clear the memory matches approximately 91.4% of the duration of data transmission across the network.

Architectural root cause. This counter-intuitive phenomenon—where a local memory operation rivals the cost of remote transmission—is a direct consequence of the disparity between modern interconnect speeds and DRAM bandwidth.

- High-speed interconnect: The test environment utilizes InfiniBand NDR (400 Gb/s) and PCIe Gen4 x16. These technologies provide massive throughput, significantly reducing the latency of RDMA Writes.

- Memory bandwidth bottleneck: Conversely, the zero-out operation is bound by the DRAM bandwidth and the CPU's ability to issue store instructions. Unlike RDMA, which offloads data movement to the RNIC, zeroing requires the CPU to explicitly mutate memory state.

doi:10.6342/NTU202600379



Consequently, in high-performance clusters where the network pipeline is highly optimized, the latency of local memory maintenance becomes a non-negligible factor. This analysis indicates that the computational cost of memory sanitization is substantial and cannot be ignored for our RDMA ring buffer design.

As identified in the previous analysis, the mandatory memory zero-out operation constitutes a significant performance bottleneck for large payloads. To circumvent this redundancy, we redesign the notification mechanism by transitioning from the standard RDMA Write to the RDMA Write with Immediate primitive.

Mechanism shift: From polling memory to polling completion queue. The fundamental improvement lies in how the receiver detects the arrival of new messages.

- Legacy approach (Polling Memory): In the naive design, the receiver must continuously poll the specific memory address (the message header) to detect changes. This dependency necessitates the expensive zero-out operation to distinguish the valid new message from the stale residuals of previous transmissions.
- Optimized approach (Polling Completion Queue): The RDMA Write with Immediate primitive allows the sender to attach a 4-byte immediate value to the write operation. Upon completion, this value is not written to the message memory region but is instead delivered directly to the receiver's Completion Queue (CQ). Consequently, the receiver detects message arrival by actively polling the CQ via *ibv_poll_cq* rather than scanning the memory buffer.

Leveraging this capability, we restructured the ring buffer communication protocol:

1. Payload size migration: Instead of embedding the 4-byte payload size within the

doi:10.6342/NTU202600379

message header in the RDMA Ring Buffer, the publish manager now packs this metadata directly into the Immediate Data field of the Completion Queue Entry.

2. Completion-driven retrieval: When the subscribe manager successfully polls a Completion Queue Entry, it extracts the payload size directly from the Completion Queue Entry metadata.

3. Elimination of sanitization: This is the critical optimization. Since the notification of a new message—along with its size—is now confirmed solely through the Completion Queue Entry, the specific values residing in the memory become irrelevant. The subscribe manager no longer needs to rely on checking for "non-zero headers" in memory to identify valid messages.

By adopting RDMA Write with Immediate, we render the "ghost message" problem obsolete, thereby allowing us to completely eliminate the pinned memory zero-out cost. This optimization removes the CPU-bound memory bottleneck, ensuring that the system's throughput is limited only by the underlying high-bandwidth interconnect.

3.5 Data Path 2

Having established the rigorous design and optimization of the proposed RDMA Ring Buffer, we now integrate this structure into the broader system architecture. Data Path 2 represents the critical inter-node link where data traverses the network fabric from the publish manager to the subscribe manager. To quantify the architectural benefits of our design, we contrast the proposed transmission mechanism against a standard industry baseline.

TCP/IP (baseline). The conventional approach relies on the standard TCP/IP stack

doi:10.6342/NTU202600379

for inter-node communication. In this model, the publish manager transmits data streams over sockets, where payloads are ultimately received into the subscribe manager's host pageable memory. This architecture suffers from inherent inefficiencies:

1. Kernel overhead: Packet processing involves significant CPU intervention and mode switching (User-Kernel-User transitions).

2. Memory copies: Data must be copied from the NIC buffer to the kernel socket buffer, and finally to the application's pageable memory.

3. Lack of GPU affinity: Since the data lands in pageable memory, it cannot be directly accessed by the GPU without further staging or explicit pinning operations.

RDMA-Write-Imm (proposed). In contrast, our proposed RDMA-Write-Imm architecture fully operationalizes the CUDA-RDMA pinned ring buffer design detailed in the preceding sections.

1. Direct placement: The publish manager utilizes the RDMA Write with Immediate primitive to deposit message payloads directly into the subscribe manager's pre-allocated, pinned ring buffer.

2. CPU bypass: This operation is handled entirely by the RNIC, bypassing the remote CPU and OS kernel. The subscribe manager is notified solely via the Completion Queue, freeing its CPU resources for application logic.

3. Ready-to-consume: Since the destination is pinned memory (registered for CUDA access), the data is immediately available for GPU transmission without additional host-side copying.

doi:10.6342/NTU202600379

By replacing the heavy TCP/IP stack with the lightweight RDMA-Write-Imm pathway, the system achieves low-latency transmission, ensuring that the high-bandwidth capabilities of the underlying interconnect are fully translated into application-level throughput.

3.6 Data Path 3

The final stage of the pipeline, Data Path 3, governs the local distribution of messages from the subscribe manager to multiple distinct subscribers running on the destination node. This phase is critical for determining the system's scalability regarding the number of subscribers.

Baseline: Iceoryx-based Payload Distribution The baseline architecture employs Iceoryx, an industry-standard middleware renowned for its low-latency, shared-memory-based communication.

- Strengths: For CPU-to-CPU communication, Iceoryx is highly efficient, delivering constant-time latency regardless of payload size due to its zero-copy shared memory design.

- Bottleneck in GPU Context: However, efficiency degrades significantly when subscribers require data in GPU memory. In this model, the subscribe manager publishes the full message payload via Iceoryx to the host pageable memory of all subscribers. Subsequently, each individual subscriber must perform its own *cudaMemcpy* to transfer the data from host pageable memory to its private GPU memory. This results in N redundant host-to-device memory transfers for N subscribers, causing

doi:10.6342/NTU202600379

linear bandwidth saturation on the PCIe bus.

Proposed: Shared GPU Memory with Notification To eliminate these redundant transfers, the proposed design completely decouples payload placement from notification logic, leveraging a shared GPU memory pool.

1. Unified Allocation: Upon retrieving a message from the ring buffer, the subscribe manager first utilizes the TLSF algorithm to allocate a single memory segment within the shared GPU memory pool.

2. Single Host-to-device Transfer: The subscribe manager then executes a single *cudaMemcpy* operation to transfer the message from the pinned ring buffer directly to this allocated shared GPU segment. Crucially, this operation occurs exactly once, regardless of the number of downstream subscribers.

3. Index-based Notification: Instead of broadcasting the message, the subscribe manager utilizes Iceoryx solely to publish lightweight metadata—specifically, the message queue entry index.

4. Zero-copy Access: Upon receiving this index, each subscriber independently computes the address of the message within the shared GPU pool. Since the GPU memory is pre-mapped into every subscriber's address space, they can immediately access the message for processing without performing any additional memory copies.

By shifting from "payload distribution" to "notification-based distribution," the proposed architecture significantly minimizes PCIe bandwidth consumption, ensuring scalable performance even with a high density of subscribers.

doi:10.6342/NTU202600379

3.7 Programming Interface

RGAPS exposes a lightweight, topic-scoped programming interface designed for GPU-centric pipelines where a publisher produces large results directly in GPU memory and disseminates them to one or more GPU-resident subscribers with minimal end-to-end overhead. In a typical usage scenario, a publisher first computes a result buffer on the GPU (i.e., the payload resides in a device memory region) and then publishes this buffer as a message to a topic.

A publisher is represented by a per-topic handle, `rgaps_publisher_t`, which encapsulates the topic-scoped state required for efficient publication. This includes a pointer to the per-topic shared CUDA–RDMA pinned memory pool on the publish machine and allocator metadata (e.g., TLSF metadata) for variable-sized message allocation within that pool. Similarly, a subscriber is represented by a per-topic handle, `rgaps_subscriber_t`, which maintains the topic-scoped state required for message consumption. This includes a pointer to the per-topic shared GPU memory pool on the subscriber machine, allocator metadata for allocating incoming payload buffers, and the local subscription context used to receive message metadata and locate payloads.

The RGAPS API consists of the following publisher and subscriber primitives:

- `rgaps_publisher_t *publisher_init(char *topic_name)`

  Attaches the calling process as a publisher to the specified topic and returns an initialized publisher handle. The returned publisher handle binds the publisher to the per-topic shared CUDA–RDMA pinned memory pool and the associated allocator metadata needed for subsequent publications. A `NULL` return indicates initialization

doi:10.6342/NTU202600379

failure.

- `int publisher_push(rgaps_publisher_t *publisher, size_t message_size, void *device_pointer)`

  Publishes one message to the topic associated with the publisher. The payload is assumed to already reside in GPU device memory at `device_pointer`. Internally, the function uses the per-topic state stored in `rgaps_publisher_t` (e.g., the shared CUDA–RDMA pinned pool pointer and TLSF metadata) to stage the message and disseminate it downstream. The function returns a status code indicating success or failure.

- `void publisher_destroy(rgaps_publisher_t *publisher)`

  Detaches the publisher from its topic and releases resources associated with the topic-scoped publisher handle, including any local state created during publisher initialization.

- `rgaps_subscriber_t *subscriber_init(char *topic_name)`

  Attaches the calling process as a subscriber to the specified topic and returns an initialized subscriber handle. The returned subscriber handle provides access to the per-topic shared GPU memory pool on the subscriber machine, allocator metadata for payload placement, and the subscription context used to receive publications. A `NULL` return indicates initialization failure.

- `void *subscriber_pull(rgaps_subscriber_t *subscriber)`

  Retrieves one published message for the topic associated with the subscriber and returns a GPU device pointer to the payload buffer placed in the per-topic shared

doi:10.6342/NTU202600379

GPU memory pool. This design allows downstream GPU kernels to consume the payload directly from GPU memory without requiring an additional per-subscriber host–device copy.

- `void subscriber_destroy(rgaps_subscriber_t *subscriber)`

  Detaches the subscriber from its topic and releases resources associated with the topic-scoped subscriber handle, including the subscription context and any local state created during subscriber initialization.

doi:10.6342/NTU202600379

Chapter 4

Evaluation

4.1 Experiment Setup

In order to validate the efficiency of our design, we conducted experiments on a fully symmetric distributed environment consisting of two high-performance workstations—one designated as the publish machine and the other as the subscribe machine. Each node is powered by an AMD Ryzen Threadripper 3970X processor (32 cores, 64 threads) backed by 128 GB of DDR4-2666 system memory. For the critical data path components, we utilize NVIDIA GeForce RTX 3090 GPU (24 GB device memory) and NVIDIA ConnectX-7 RNIC in each node. Crucially, to maximize data throughput and minimize PCIe bottlenecking, both the GPU and the RNIC are attached to the host system via independent PCIe Gen4 x16 interfaces. The two machines are directly connected via a high-speed InfiniBand NDR 400 Gb/s link, ensuring that network bandwidth is sufficient to saturate the PCIe bus. For comparative analysis, the baseline shared-memory communication is managed by the Iceoryx v2.0.6 publish/subscribe middleware. The detailed hardware and software specifications are tabulated in Table 4.1.

doi:10.6342/NTU202600379

Table 4.1: The hardware and software configurations of publish machine and subscribe machine.

	Publish Machine	Subscribe Machine
Hardware		
CPU	AMD Ryzen Threadripper 3970X (32 cores / 64 threads)	AMD Ryzen Threadripper 3970X (32 cores / 64 threads)
GPU	NVIDIA GeForce RTX 3090 (10496 CUDA cores, 24GB) PCIe Gen4 x16	NVIDIA GeForce RTX 3090 (10496 CUDA cores, 24GB) PCIe Gen4 x16
RAM	128 GB DDR4-2666 (4 × 32 GB)	128 GB DDR4-2666 (4 × 32 GB)
Adapter	NVIDIA ConnectX-7 — PCIe Gen4 x16	NVIDIA ConnectX-7 — PCIe Gen4 x16
Interconnect	InfiniBand NDR 400 Gb/s	
Software		
OS	Ubuntu 22.04 (kernel: 6.8.0-51-generic)	Ubuntu 22.04 (kernel: 6.8.0-87-generic)
Compiler	nvcc v11.5, gcc 11.4.0, g++ 11.4.0	nvcc v11.5, gcc 11.4.0, g++ 11.4.0
CUDA	12.7	12.7
Iceoryx	2.0.6	2.0.6

4.2 Data Path 1

The first critical stage of the transmission pipeline, Data Path 1, involves the transfer of message payloads from the publisher's GPU memory to the host system memory. To quantify the benefits of our memory management strategy, we compared the latency of the proposed Pinned-SHM approach against the baseline Pageable-UDS configuration in Data Path 1.

Figure 4.1 illustrates the device-to-host memory transfer latency across varying message sizes ranging from 4MB to 1GB, demonstrating a consistent performance advantage for the proposed design. Specifically, for small payloads (4MB), the proposed method achieves a latency of 224 μs compared to 451 μs for the baseline, representing a 2.01x speedup. As the payload size increases to 64MB, the proposed method records 2,906 μs versus 5,709 μs (approximately 1.96x speedup). For gigabyte-scale transmission (1GB),

doi:10.6342/NTU202600379

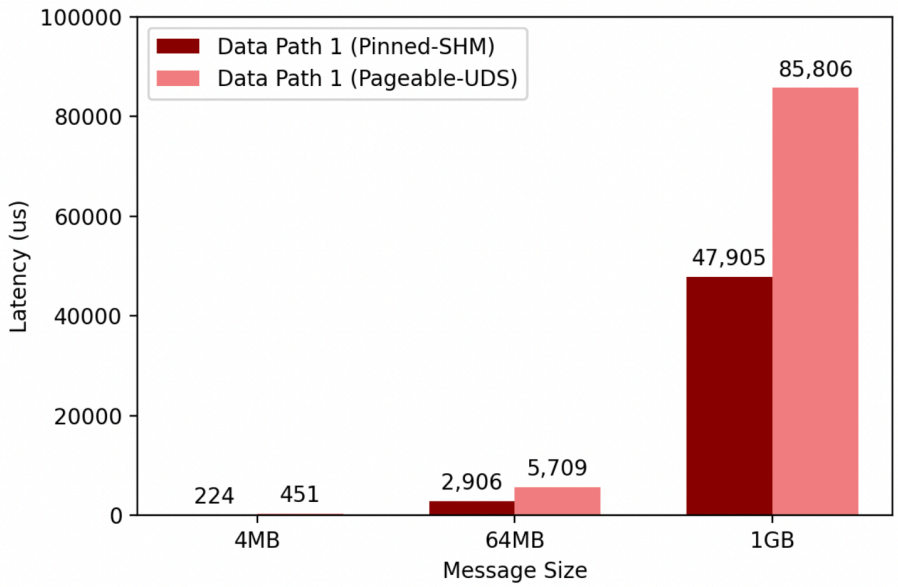


Figure 4.1: Device-to-host memory transfer latency of Pinned-SHM and Pageable-UDS in Data Path 1.

the Pinned-SHM approach maintains its dominance with a latency of 47,905 μs, whereas the Pageable-UDS baseline requires 85,806 μs (approximately 1.79x speedup). By eliminating the implicit "double-copy" overhead inherent in pageable memory transfers, the proposed architecture effectively reduces the latency of the device-to-host data path by approximately half.

To pinpoint the exact sources of latency improvements, we conducted a granular breakdown of the data flows within Data Path 1. Table 4.2 and Table 4.3 provide a detailed comparison of the time consumption for each step in the proposed Pinned-SHM architecture versus the baseline Pageable-UDS approach.

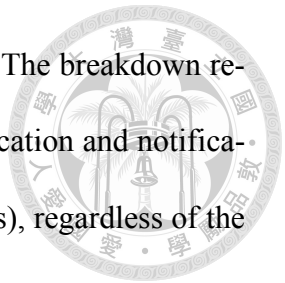
Table 4.2: Latency breakdown of Data Path 1 (Pinned-SHM).

Message Size	**T1: TLSF Malloc**	**T2: cudaMemcpy**	**T3: Notification via shared memory**
4 MB	1	224	1
64 MB	1	2,906	1
1 GB	1	47,905	1

Unit: μs

As shown in Table 4.2, the proposed Pinned-SHM workflow consists of three distinct

doi:10.6342/NTU202600379



stages: TLSF memory allocation, memory transfer, and notification. The breakdown reveals a highly optimized profile wherein both the TLSF memory allocation and notification via shared memory exhibit consistently negligible latency ($\approx 1\mu$s), regardless of the message size. This confirms that the complexity of our dynamic memory management and signaling is effectively constant and minimal. Consequently, the total latency is almost exclusively dominated by the *cudaMemcpy* operation, signifying that the PCIe link bandwidth has become the sole performance bottleneck for the system.

Table 4.3: Latency breakdown of Data Path 1 (Pageable-UDS).

Message Size	**T1: cudaMemcpy**	**T2: IPC (UDS)**
4 MB	451	1,184
64 MB	5,709	17,810
1 GB	85,806	208,318

Unit: μs

In contrast, Table 4.3 exposes the severe bottlenecks inherent in the baseline Pageable-UDS design, where the workflow involves a device-to-host copy followed by data transmission via Unix Domain Socket. The Unix Domain Socket transmission stage introduces massive latency that grows linearly with message size; for instance, transferring a 1GB message consumes 208,318 μs. This starkly highlights the inefficiency of copying large data payloads through kernel socket buffers.

Overall, the comparison demonstrates that the Pinned-SHM design transforms the Data Path 1 bottleneck. By shifting from a software-bound mechanism (heavy IPC and memory copying between user and kernel space) to a hardware-bound mechanism (pinned memory and direct DMA), we achieve orders-of-magnitude improvements in efficiency.

doi:10.6342/NTU202600379

4.3 Data Path 2

Following the local device-to-host transfer, Data Path 2 handles the transmission of messages across the network fabric from the publish machine to the subscribe machine. To demonstrate the efficiency of our network design, we compared the latency of the proposed RDMA Write with Immediate mechanism integrated with the RDMA ring buffer design against a standard TCP/IP baseline over the same physical interconnect.

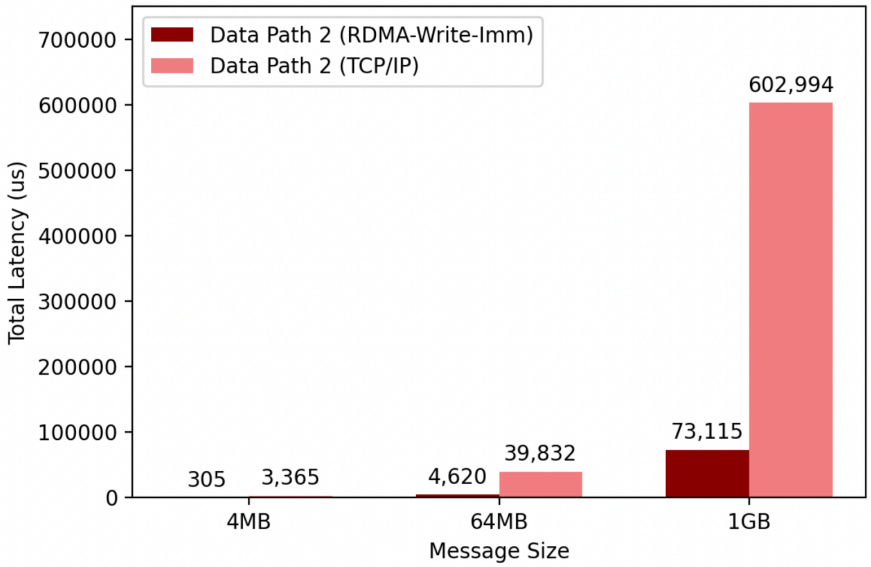


Figure 4.2: Total latency of Data Path 2 (RDMA-Write-Imm) and Data Path 2 (TCP/IP).

Figure 4.2 presents the total transmission latency across varying message sizes, revealing a substantial performance disparity between the two protocols. Specifically, for relatively small messages (4MB), the proposed RDMA Write with Immediate approach achieves a latency of 305 μs, whereas the TCP/IP baseline requires 3,365 μs, representing an approximately 11x speedup. This indicates that the overhead of the TCP/IP stack dominates even at smaller scales.

doi:10.6342/NTU202600379

4.4 Data Path 3

The final stage, Data Path 3, involves distributing the received message from the subscribe manager's host memory to the GPU memory of multiple subscriber applications. To evaluate the scalability of our framework, we measured the latency behavior as the number of subscribers (N) increases from 1 to 8. Table 4.4 and Table 4.5 compare the detailed latency breakdown of the proposed Pinned-Shared architecture against the baseline Pageable-Iceoryx approach.

Table 4.4: Latency breakdown of Data Path 3 (Pinned-Shared).

Message Size	T1:	T2:	T3: Notification via Iceoryx (Avg Latency)			
	TLSF Malloc	cudaMemcpy	N = 1	N = 2	N = 4	N = 8
4 MB	1	216	6	7	9	11
64 MB	1	2,872	6	7	9	11
1 GB	1	43,677	6	7	9	11

* N: Number of subscribers Unit: μs

As shown in Table 4.4, the proposed method exhibits highly scalable latency regardless of the subscriber count. The system performs the heavy *cudaMemcpy* operation exactly once, depositing the data into a shared GPU memory segment. The breakdown confirms that the dominant cost in our design is the single host-to-device transfer, which is largely independent of the number of subscribers. The subsequent notification step incurs negligible overhead. Consequently, the total latency remains stable even as the system scales to 8 subscribers.

In contrast, Table 4.5 reveals the severe scalability limitations of the baseline approach. The *cudaMemcpy* operation must be repeated N times, causing the total latency to grow linearly.

doi:10.6342/NTU202600379

Table 4.5: Latency breakdown of Data Path 3 (Pageable-Iceoryx).

Message Size	T1: Publish message via Iceoryx				T2: cudaMemcpy (Avg Latency)			
	N=1	N=2	N=4	N=8	N=1	N=2	N=4	N=8
4 MB	6	7	7	10	396	520	866	2,238
64 MB	9	9	11	12	5,625	7,439	12,468	24,357
1 GB	9	10	11	12	85,172	118,273	191,378	382,018

* N: Number of subscribers Unit: μs

Overall, the results demonstrate that the proposed shared GPU memory design effectively decouples the distribution latency from the number of consumers. By avoiding PCIe contention among subscribers and eliminating redundant data copying, the proposed framework proves its suitability for high-throughput, one-to-many multicast environments. This architectural advantage is crucial for enabling low and scalable latency when multiple GPU subscribers consume the same large message.

To further investigate the scalability limitations observed in the baseline Pageable-Iceoryx architecture, we analyzed the detailed latency behavior when multiple subscribers attempt to retrieve data simultaneously. Figures 4.3, 4.4, and 4.5 characterize the per-subscriber GPU transfer cost in this baseline, where each subscriber receives its own host-side copy of the payload and then performs a host-to-device transfer (*cudaMemcpy*) into its private GPU memory. Importantly, subscribers start this transfer concurrently; that is, multiple subscribers initiate the transfer at roughly the same time after receiving the published message. Consequently, the measured latency directly reflects the impact of resource sharing and contention on the common data path, specifically the PCIe link and GPU copy engines.

As shown in Figures 4.3, 4.4, and 4.5, when the number of subscribers increases, the per-subscriber *cudaMemcpy* latency increases substantially because multiple transfers compete for the same underlying bandwidth. This confirms that, in the baseline, the fan-

doi:10.6342/NTU202600379

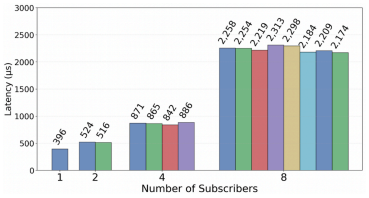


Figure 4.3: cudaMemcpy per individual subscriber (4 MB).

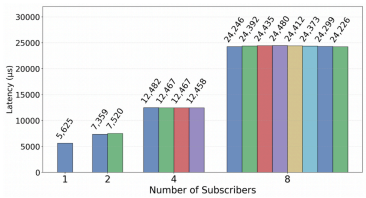


Figure 4.4: cudaMemcpy per individual subscriber (64 MB).

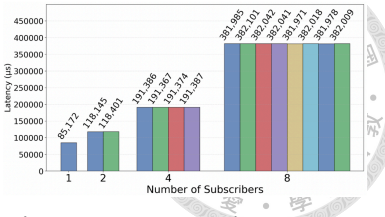


Figure 4.5: cudaMemcpy per individual subscriber (1 GB).

out cost is dominated by duplicated host-to-device transfers that scale with the subscriber count.

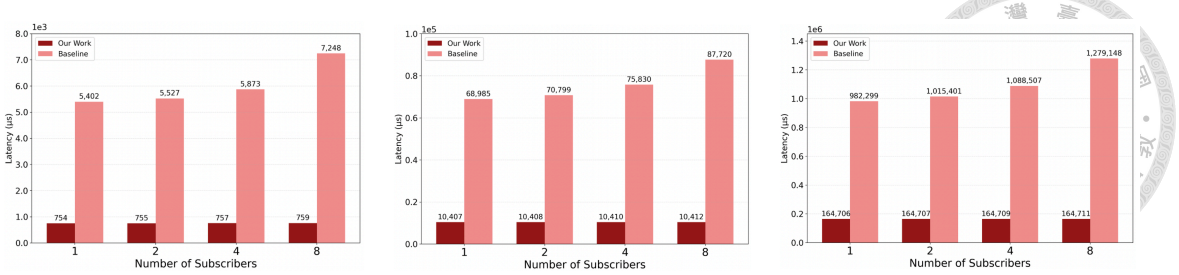
Another key observation is that latencies across subscribers are tightly clustered within the same configuration, indicating that when transfers run concurrently, subscribers experience similar effective bandwidth. At 8 subscribers, the spread is very small: for 1 GB, values are all around 381.97–382.10 ms (Figure 4.5); for 64 MB, around 24.23–24.48 ms (Figure 4.4); and for 4 MB, around 2.17–2.31 ms (Figure 4.3). This "flat" distribution suggests the system is operating in a bandwidth-saturated regime where each subscriber receives a comparable share of the available transfer capacity, rather than showing outliers caused by scheduling artifacts.

Overall, these results highlight a fundamental limitation of the baseline design for large payloads and multiple subscribers: because each subscriber performs its own host-to-device transfer, the system incurs N redundant GPU transfers per message, leading to higher latency and poor scalability as the subscriber count grows.

4.5 Total Latency

Figure 4.6 (a), Figure 4.6 (b), and Figure 4.6 (c) compare the average end-to-end latency per subscriber between RGAPS and the baseline for message sizes of 4 MB, 64

doi:10.6342/NTU202600379



(a) 4 MB (b) 64 MB (c) 1 GB

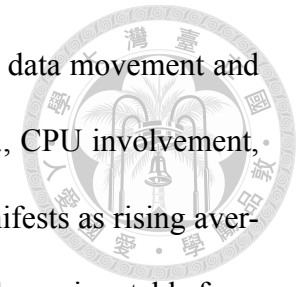
Figure 4.6: Total latency comparison across different message sizes.

MB, and 1 GB, respectively. For each configuration, latency is measured from the time the publisher issues a publish operation on the publish machine to the time a subscriber on the subscribe machine can access a GPU-resident payload (i.e., obtain a device-accessible buffer pointer). The reported latency is averaged across all subscribers under the same fan-out degree $N \in \{1, 2, 4, 8\}$.

Across all three figures, RGAPS consistently achieves substantially lower latency than the baseline. In Figure 4.6 (a), RGAPS remains near 754–759 μs, whereas the baseline ranges from 5,402 μs at $N = 1$ to 7,248 μs at $N = 8$. In Figure 4.6 (b), RGAPS stays around 10,407–10,412 μs, while the baseline increases from 68,985 μs to 87,720 μs as N grows. In Figure 4.6 (c), RGAPS reports approximately 164,706–164,711 μs, compared with 982,299–1,279,148 μs for the baseline. Overall, these results correspond to a speedup of roughly $6\times$–$10\times$, with the largest gains observed at higher fan-out.

A key trend visible in Figure 4.6 (a)–4.6 (c) is that RGAPS' average latency is nearly insensitive to the number of subscribers, while the baseline's latency increases with N. RGAPS keeps the dominant message copying cost largely constant per message on the subscribe machine: the incoming message is staged and transferred into GPU memory once, after which all subscribers consume the same GPU-resident message via notification fan-out; adding subscribers primarily adds only lightweight notification overhead.

doi:10.6342/NTU202600379



In contrast, the baseline's delivery path induces more per-subscriber data movement and contention, so increasing N amplifies shared-resource pressure (e.g., CPU involvement, DRAM traffic, and DMA/PCIe copy-engine contention), which manifests as rising average latency. This difference in scaling behavior explains why RGAPS remains stable from $N = 1$ to $N = 8$ in all three figures, whereas the baseline degrades with fan-out, and why RGAPS' end-to-end advantage becomes more pronounced as the number of subscribers increases.

doi:10.6342/NTU202600379

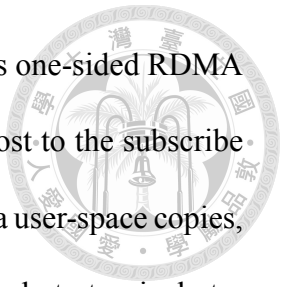
Chapter 5

Conclusion

This thesis presents RGAPS, a distributed GPU-aware publish/subscribe system that extends the original GAPS design from single-host, single-GPU zero-copy communication to cross-host GPU-to-GPU pub/sub. While GAPS achieves extremely low-latency fan-out inside one machine by sharing a per-topic GPU memory pool among processes, it is inherently constrained by single-host locality. Specifically, all publishers and subscribers must co-reside on the same machine, leading to GPU compute contention when the workload scales. Furthermore, each topic requires a dedicated shared GPU pool, causing the total GPU memory footprint to grow with the number of topics and eventually exceed the capacity of a single GPU. RGAPS addresses these limitations by enabling topics and workloads to be distributed across multiple machines, thereby alleviating both GPU memory capacity pressure and compute-resource contention.

Experimental results demonstrate that RGAPS reduces end-to-end overhead at critical stages compared with conventional pageable-memory and socket-based baselines. In Data Path 1, using pinned host buffers significantly improves GPU-to-host transfer latency, and replacing payload-carrying IPC with metadata-only shared-memory signaling

doi:10.6342/NTU202600379

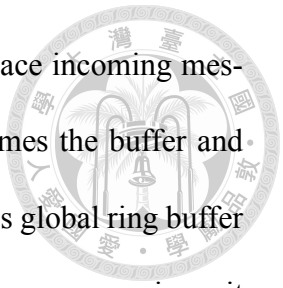


removes a major source of overhead. In Data Path 2, RGAPS employs one-sided RDMA WRITE (with immediate) to move large payloads from the publish host to the subscribe host without traversing the kernel networking stack or introducing extra user-space copies, providing a low-latency and high-throughput transport for multi-megabyte to gigabyte-scale messages. In Data Path 3, RGAPS preserves a key advantage of GAPS in a distributed setting: the payload is transferred into GPU memory once per message on the subscriber machine, after which multiple subscribers can consume the same GPU buffer without incurring per-subscriber copy costs. This enables scalable fan-out while maintaining stable latency behavior as the number of subscribers increases, aligning with RGAPS ' s target use cases: high-frequency, large-payload, latency-sensitive GPU pipelines.

Despite these advantages, RGAPS also has limitations. Because the evaluation platform does not support GPUDirect RDMA [7], RGAPS currently stages network I/O through host DRAM: payloads must be written into (and read from) pinned host buffers that are registered as RDMA memory regions before and after the RDMA transfer. This DRAM staging introduces additional memory traffic and extra PCIe/host-memory touch points, which can add non-trivial latency for large messages and consume CPU-memory bandwidth. A natural next step is to integrate GPUDirect RDMA so that the RNIC can execute direct data transfers to and from the GPU's onboard memory, thereby bypassing host DRAM staging and further reducing end-to-end latency. Moreover, this work focuses on a two-host architecture; extending RGAPS to multi-hop, multi-host topologies requires additional coordination for routing, admission control, and resource management (e.g., multi-topic placement policies and buffer provisioning).

In the current RGAPS design, the subscribe manager on the subscribe machine maintains a single global CUDA–RDMA pinned ring buffer for all topics. The publish manager

doi:10.6342/NTU202600379



on the publish machine uses one-sided RDMA Write operations to place incoming messages into this ring buffer, after which the subscribe manager consumes the buffer and forwards messages into the local GPU pub/sub fan-out path. While this global ring buffer simplifies buffer management and reduces the number of registered memory regions, it becomes a scalability bottleneck under high load. As message rates increase or message sizes grow, the global ring buffer can be saturated quickly, which triggers backpressure. The publish manager must stall until the subscribe manager advances the consumption pointer and frees sufficient contiguous space for subsequent RDMA Write operations. In addition, the current implementation uses a single worker thread on each side for the global ring-buffer data path, effectively forming a single logical communication channel between the publish manager and the subscribe manager. As a result, the design does not fully exploit the available CPU cores on either host, nor does it leverage the subscribe machine's host DRAM capacity to provision larger staging buffers when the workload scales.

To address this limitation, a more scalable extension is to replace the global ring buffer with multiple per-topic ring buffers. Specifically, RGAPS can allocate a dedicated CUDA–RDMA pinned ring buffer for each topic, and associate it with a per-topic transfer thread pair. One thread in the publish manager and one thread in the subscribe manager handle RDMA Write operations and buffer consumption for that topic. This approach increases the effective staging capacity by allowing buffers to be provisioned independently per topic and expanded using host DRAM. It also improves throughput scalability by enabling parallelism across topics, better utilizing multi-core CPUs and distributing RDMA work across multiple concurrent channels. Overall, the global ring-buffer saturation problem is primarily a scalability issue, and per-topic RDMA receive ring buffers with parallel transfer threads provide a straightforward path toward higher aggregate publish rates and

doi:10.6342/NTU202600379

larger message volumes in RGAPS.

RGAPS currently implements a live-only delivery semantic for late-joining subscribers. Concretely, a subscriber that dynamically joins a topic will only receive messages published after the subscription becomes active, and it will not automatically receive any previously published messages. This design choice aligns with RGAPS's primary target of large-payload, latency-sensitive GPU pipelines, where the critical objective is to minimize end-to-end overhead on the hot path rather than to maintain durable history.

As future work, RGAPS can be extended with optional QoS semantics for late joiners. One direction is full history replay, where a newly joined subscriber can retrieve all prior messages for a topic, or a bounded prefix such as the last K messages. Supporting this semantic requires maintaining a per-topic message log or history buffer, together with indexing metadata that allows a subscriber to locate and replay older payloads in order. Another direction is latest-value delivery, where a late joiner receives only the most recently published message upon joining and then continues with live traffic. This semantic is useful for state-like topics in which only the latest sample matters, and it can be realized by retaining a per-topic last-message pointer and ensuring the corresponding payload remains accessible long enough for new subscribers to consume it. Together, these extensions would allow RGAPS to cover a broader spectrum of pub/sub QoS needs, ranging from pure streaming workloads to state synchronization and history-aware analytics, while keeping the current live-only mode as the low-overhead default.

doi:10.6342/NTU202600379

References

[1] Dotan Barak. RDMAmojo: RDMA Programming Blog, 2024. https://www.rdmamojo.com/. Accessed: 2025-12-25.

[2] Aleksandar Dragojević, Dushyanth Narayanan, Orion Hodson, and Miguel Castro. FaRM: Fast Remote Memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, Seattle, WA, 2014. USENIX Association.

[3] Xinhao Kong, Jingrong Chen, Wei Bai, Yechen Xu, Mahmoud Elhaddad, Shachar Raindel, Jitendra Padhye, Alvin R. Lebeck, and Danyang Zhuo. Understanding RDMA Microarchitecture Resources for Performance Isolation. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, Boston, MA, 2023. USENIX Association.

[4] Hao-En Kuan, Yung-Hsiang Yang, Zen-Mou Jiang, Chi-Sheng Shih, and Shih-Hao Hung. Towards Low-Latency GPU-Aware Pub/Sub Communication for Real-Time Edge Computing. In *2025 IEEE 31st International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*. IEEE, 2025.

[5] Linux man-pages project. ibv_reg_mr(3) — Linux manual page, 2024. https://man7.org/linux/man-pages/man3/ibv_reg_mr.3.html. Accessed: 2025-12-25.

doi:10.6342/NTU202600379

[6] Miguel Masmano, Ismael Ripoll, Alfons Crespo, and Jorge Real. TLSF: a New Dynamic Memory Allocator for Real-Time Systems. In *Proceedings. 16th Euromicro Conference on Real-Time Systems, 2004. ECRTS 2004.*, pages 79–88. IEEE, 2004.

[7] NVIDIA Corporation. *GPUDirect RDMA*, 2024. https://docs.nvidia.com/cuda/gpudirect-rdma/. Accessed: 2025-12-25.

[8] NVIDIA Corporation. *RDMA Aware Networks Programming User Manual: Transport Modes*, 2024. https://docs.nvidia.com/networking/display/rdmaawareprogrammingv17/transport+modes. Accessed: 2025-12-25.

[9] NVIDIA Corporation. CUDA Runtime API: 6.10. Memory Management, 2026. https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__MEMORY.html. Accessed: 2026-01-02.

[10] NVIDIA Corporation. How to Optimize Data Transfers in CUDA C/C++, 2026. https://developer.nvidia.com/blog/how-optimize-data-transfers-cuda-cc/. Accessed: 2026-01-02.

[11] sysprog21. tlsf-bsd: Two-Level Segregated Fit memory allocator implementation, 2024. https://github.com/sysprog21/tlsf-bsd. Accessed: 2025-12-25.

[12] Konstantin Taranov, Steve Byan, Virendra Marathe, and Torsten Hoefler. KafkaDirect: Zero-copy Data Access for Apache Kafka over RDMA Networks. In *Proceedings of the 2022 International Conference on Management of Data (SIGMOD '22)*, pages 2191–2204. ACM, 2022.

doi:10.6342/NTU202600379

[13] The Apache Software Foundation. Apache Kafka, 2024. https://kafka.apache.org/. Accessed: 2025-12-25.

[14] The Eclipse Foundation. Eclipse iceoryx: True Zero-Copy Inter-Process-Communication, 2024. https://iceoryx.io/. Accessed: 2025-12-25.

[15] The Eclipse Foundation. Eclipse Zenoh: Zero Overhead Pub/Sub, Store/Query and Compute, 2024. https://zenoh.io/. Accessed: 2025-12-25.

[16] The Eclipse iceoryx Project. eclipse-iceoryx/iceoryx: True Zero-Copy Inter-Process-Communication Source Code, 2024. https://github.com/eclipse-iceoryx/iceoryx. Accessed: 2025-12-25.

[17] The Eclipse Zenoh Project. eclipse-zenoh/zenoh: Blue Mountain (Rust) Implementation Source Code, 2024. https://github.com/eclipse-zenoh/zenoh. Accessed: 2025-12-25.

doi:10.6342/NTU202600379