Enhancing RPC on Slingshot for Aurora's DAOS Storage System

Jerome Soumagne
Hewlett Packard Enterprise
Spring, Texas, USA
jerome.soumagne@hpe.com

Alexander Oganezov
Hewlett Packard Enterprise
Spring, Texas, USA
alexander.oganezov@hpe.com

Ian Ziemba
Hewlett Packard Enterprise
Spring, Texas, USA
ian.ziemba@hpe.com

Steve Welch
Hewlett Packard Enterprise
Spring, Texas, USA
welch@hpe.com

Philip Carns
Argonne National Laboratory
Lemont, Illinois, USA
carns@mcs.anl.gov

Kevin Harms
Argonne National Laboratory
Lemont, Illinois, USA
harms@alcf.anl.gov

John Carrier
Hewlett Packard Enterprise
Spring, Texas, USA
john.carrier1@hpe.com

Johann Lombardi
Hewlett Packard Enterprise
Spring, Texas, USA
johann.lombardi@hpe.com

Mohamad Chaarawi
Hewlett Packard Enterprise
Spring, Texas, USA
mohamad.chaarawi@hpe.com

Zhen Liang
Hewlett Packard Enterprise
Spring, Texas, USA
zhen.liang@hpe.com

Scott Peirce
Hewlett Packard Enterprise
Spring, Texas, USA
scott.peirce@hpe.com

Abstract

DAOS is an open-source software-defined high-performance storage solution designed for massively distributed solid-state drives and non-volatile memory. It is a key component of the Aurora exascale system that aims to deliver high storage throughput and low latency to application users. Utilizing the Slingshot interconnect, DAOS leverages Remote Procedure Call (RPC) to communicate between compute and storage nodes. While the preexisting RPC mechanism used by DAOS was already designed for high-performance computing fabrics, it required a number of scalability, performance, and security enhancements in order to be successfully deployed on Aurora.

We present and discuss in this paper the improvements that were made to address this set of challenges in each of the components that DAOS relies on: the Collective and RPC Transport (CaRT) layer, the Mercury RPC library, and the libfabric Slingshot (cxi) provider. While some of these enhancements were tailored to the DAOS storage system specifically, they also serve as a broader reference for implementing scalable high-performance data services over RDMA fabrics. This paper focuses on the design and implementation of each functionality that was improved, enabling DAOS to provide both high throughput and low latency to Aurora users.

CCS Concepts

• **Networks** → **Application layer protocols**; *Programming interfaces*; • **Software and its engineering** → Software design engineering; *Client-server architectures*; • **Information systems** → **Application servers**; *Storage network architectures.*

Keywords

Remote Procedure Call, Data Services, High-Performance Computing, Slingshot, DAOS, Mercury, RDMA, libfabric

ACM Reference Format:
Jerome Soumagne, Alexander Oganezov, Ian Ziemba, Steve Welch, Philip Carns, Kevin Harms, John Carrier, Johann Lombardi, Mohamad Chaarawi, Zhen Liang, and Scott Peirce. 2025. Enhancing RPC on Slingshot for Aurora's DAOS Storage System. In *Proceedings of Cray User Group (CUG '25).* ACM, New York, NY, USA, 12 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

The design and deployment of distributed data services in high-performance computing (HPC) was once the exclusive domain of monolithic parallel file systems. However, the emergence of new storage technologies and the increasing complexity of HPC applications has prompted exploration of new data service architectures that not only take advantage of platform capabilities but are also more closely tailored to application needs [3, 10]. These HPC data services must achieve high bandwidth and low latency while also providing ease of deployment, scalability, and fault tolerance; and as such they rely heavily on the underlying network infrastructure. However, most network infrastructures are designed with an emphasis on application-level message-passing interface (MPI) first and foremost. They do not necessarily support or expose the set of features that are required for implementing data services at scale [18], which instead rely on Remote Procedure Call (RPC) to communicate [14].

In this study we examine how one prominent distributed storage system, DAOS [9], has addressed these challenges. DAOS is an open-source software-defined high-performance storage solution designed for massively distributed solid-state drives and non-volatile memory. It has notably been deployed in production as the primary storage solution for the Aurora system [4] at the Argonne Leadership Computing Facility (ALCF). DAOS has the particularity of operating end-to-end in user space (and as such it operates as any user-level service), removing significant bottlenecks in the kernel system call path and enabling data access times to be several orders of magnitude faster than in traditional storage systems. Utilizing Aurora's Slingshot interconnect [11], DAOS leverages Remote Procedure Call (RPC) to communicate between compute and storage nodes. While the preexisting RPC mechanism used by DAOS was already designed for high-performance computing fabrics, it required a number of *scalability*, *performance*, and *security* enhancements in order to be successfully deployed on Aurora. We present and discuss in this paper each of these issues along with the solutions that were taken to address them. Although we focus on DAOS in this study, we encourage the reader to consider that most of these solutions are inherent to the deployment of high-performance client-server data services at scale over RDMA fabrics. Our findings are therefore highly relevant to a wide range of distributed services, including file systems, resource managers, and performance monitoring systems. We outline, to the best of our knowledge, best practice for achieving maximum performance and usability for such services on the Slingshot interconnect.

The remainder of this paper is organized as follows. In Section 2 we present an overview of the Aurora platform and components of DAOS. We then show in Section 3 how the DAOS metadata path was augmented to reach the desired level of scalability, allowing for servers to sustain incoming metadata traffic. In Section 4 we focus on the improvements that were made to the bulk data path to ensure performance could be met, while ensuring both data integrity and data protection requirements. In Section 5 we discuss security and how the default solution defined by Slingshot had to be augmented to meet DAOS requirements. In Section 6 we discuss how data services requirements differ from traditional HPC applications and why they reside in their own class of applications, requiring special considerations.

2 Background

In this section we summarize the architecture of the Aurora platform and its DAOS storage software.

2.1 Aurora Platform Architecture

The Aurora exascale system is a leadership-class supercomputer and an HPE Cray EX-based system comprising 10,624 compute nodes, each node consisting of 2 Intel® Xeon® Max Series (codename Sapphire Rapids) CPUs with 52 physical cores supporting 2 hardware threads per core and 6 Intel® Data Center Max Series (codename Ponte Vecchio) GPUs. The system is interconnected in a dragonfly topology with HPE Slingshot 200 Gb, a high-performance interconnect that provides a 200 Gb/s bandwidth per link, with each node having 8 Slingshot NICs.

As previously mentioned, the storage system on Aurora uses DAOS as its primary storage solution, which seeks to deliver over 30 TiB/s with 1,024 storage server nodes. Each server node is equipped with 2 Intel® Xeon® Gold Series (codename Ice Lake) CPUs with 26 physical cores supporting 2 hardware threads per core, 16 Intel® Optane™ Persistent Memory (PMem) 200 series modules with 512 GiB of capacity, and 16 Samsung NVMe™ solid-state drives (SSDs) with 15.3 TiB of capacity each. Each node is connected to the network fabric through 2 Slingshot NICs.

DAOS aims to provide not only high bandwidth but also low-latency, high-IOPS storage containers to HPC applications. In addition, DAOS enables data-centric workflows combining simulation, data analytics, and AI by providing features such as transactional non-blocking I/O, advanced data protection with self-healing, end-to-end data integrity, fine-grained data control, and elastic storage. Unlike traditional storage software stacks that were designed primarily for rotating media, DAOS is built from the ground up to exploit non-volatile memory technologies. In Aurora's configuration, DAOS servers maintain their metadata on persistent memory, while bulk data goes straight to NVMe SSDs.

2.2 DAOS Software Architecture

To understand some of the challenges that we will be focusing on in the subsequent sections, one must understand how DAOS operates. As shown in Figure 1, DAOS relies on multiple components to successfully implement an RPC framework and communicate between compute and storage nodes (and between storage nodes). The RPC mechanism used for DAOS to communicate over Slingshot relies on three main components: the **Collective and RPC Transport (CaRT)** layer, the **Mercury RPC** [16] library, and **libfabric** [5], specifically the Slingshot (`cxi`) provider. In the following discussion, we distinguish two types of RPCs that DAOS may issue: *standard RPCs*, used for control messages and metadata operations, and *bulk RPCs*, used for the transfer of large data payloads over RDMA.

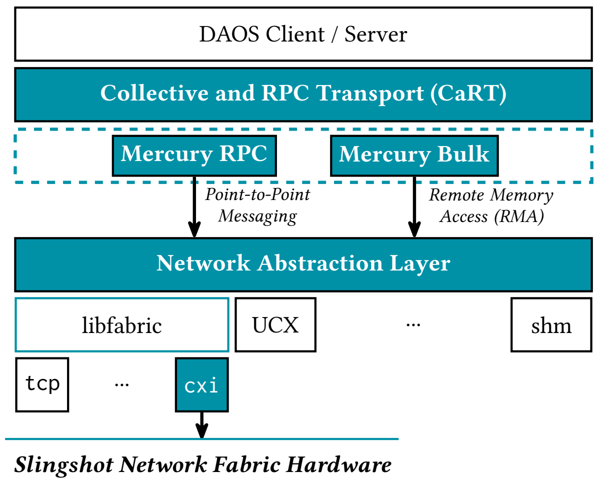

DAOS Client / Server
Collective and RPC Transport (CaRT)
Mercury RPC
Mercury Bulk
Point-to-Point Messaging
Remote Memory Access (RMA)
Network Abstraction Layer
libfabric
UCX
...
shm
tcp
...
cxi
Slingshot Network Fabric Hardware

Figure 1: Overview of DAOS network stack.

DAOS. The DAOS layer itself follows a client-server model. A DAOS server node is usually composed of multiple *DAOS engines*, one for each physical socket (2 per server node on Aurora). Within an engine, the storage is statically partitioned across multiple *DAOS targets* to optimize concurrency (16 per engine on Aurora). Each target has its private storage, its own pool of service threads, and a dedicated network endpoint that can be directly addressed over the fabric, independently of the other targets hosted on the same storage node (e.g., each target on Aurora is bound to a separate port accessing one NIC per engine and physical socket). The number of targets exported by an engine instance is configurable and depends on the underlying hardware. Leveraging CaRT, membership of the DAOS servers is recorded into the system map, assigning a unique rank/tag to each engine process/target. DAOS *pools* reserve distributed storage across multiple server targets; and, within pools, DAOS *containers* define how DAOS *objects* can be distributed across any target of the pool for both performance and resilience. On Aurora, DAOS clients reside on compute nodes, and applications commonly access the DAOS client library through higher-level I/O libraries (MPI-IO, HDF5 [15], etc.). Depending on DAOS object *placement*, the DAOS client library, which also interfaces through CaRT to send RPCs, may directly contact any of the server targets.

CaRT. The CaRT layer provides a network provider-agnostic RPC functionality to the rest of the DAOS stack and is responsible for issuing both standard and bulk RPCs over Mercury. It provides rank and group membership notions as well as implementing collective communication over different virtual spanning trees. Using these notions, engine failure detection is performed through the implementation of a Scalable Weakly Consistent Infection-style Process Group Membership (SWIM) [2] protocol. CaRT also provides some degree of flow control through optional *endpoint credits* (EP credits) and *quota* mechanisms, as well as providing a mechanism for handling RPC timeouts and retries. EP credits limit the number of outstanding RPCs to a specific endpoint target, while the quota mechanism restricts the total number of outstanding RPCs that a client can have at any given time. Both mechanisms seamlessly enqueue and process overflow RPCs in the background, allowing each client to utilize an optimal number of in-flight RPCs and associated resources without adverse effects on other clients or the system as a whole.

Mercury. Underneath CaRT, the Mercury library provides both the RPC (for small data payloads) and bulk data transfer (for large data payloads) functionality. The RPC mechanism is based on a two-sided *connectionless* and *non-blocking* communication model. The server preemptively posts receive buffers to handle incoming RPC requests; and the client, after a lookup, issues an RPC request and waits for the server to respond by making explicit network progress. This mechanism is further described in Section 3. The bulk data transfer mechanism, on the other hand, is based on a one-sided communication model, where the client registers the memory region to be transferred and the server issues a remote memory access (RMA) operation to transfer the data without intermediate copies—this mechanism presupposes that the client has exchanged its memory region information with the server through an RPC. We describe this mechanism further in Section 4. In order to implement both communication models, Mercury is built on top of a network abstraction layer, which provides, through a system of plugins, a set of primitives to various communication interfaces, including libfabric, UCX [12], and shared memory.

Libfabric. Mercury's libfabric plugin directly interfaces with libfabric, which provides its own set of communication primitives to Mercury, including both messaging and RMA operations as well as NIC hardware resource management (fabric, domain, endpoint) and memory registration, which is required before a region can be the target of an RMA operation. Libfabric is able to natively respond to Mercury's non-blocking operation needs by providing event-based data transfers, with their completion reported into completion queues after network progress is manually made. It is also able to provide connectionless semantics by defining reliable datagram (RDM) endpoints, which require only the insertion of target addresses into an address vector before communication can be initiated. Libfabric is capable of interfacing through different fabric hardware by defining a set of providers. The Slingshot (`cxi`) provider, which is the one used in the context of DAOS on Aurora, interfaces with `libcxi` to interact with the Cassini NIC, adopting Portals 4 [1] ideas relating to message matching, completion, and triggered operations (though it does not conform to the entire API).

3 Standard RPCs and Metadata Path

As briefly described in Section 2, standard RPCs in DAOS are used both for control messages and for metadata operations. However, they also serve as a foundation for bulk RPCs, which also require memory descriptors to be exchanged (this is further described in Section 4). Hence, optimizations that can be made to the standard RPC path benefit also the bulk RPC path by reducing overall latency. RPCs use a two-sided communication model, which means that the server has to actively post receive buffers to handle incoming RPC requests that are sent by clients. In this section we discuss the challenges that appeared when scaling up clients and the solutions that were taken to address these challenges. We focus primarily on scaling the number of clients, since the Aurora architecture can support far more clients than servers and server targets operate independently from one another.

3.1 Standard Receives

RPCs can be thought of as a series of messages that are exchanged between clients and servers. Each message is composed of a header and a payload. The header contains metadata that is used to identify the message (request identifier of function to execute, protocol version information), and the payload contains the actual data that is being transferred (i.e., function arguments that the server must pass when executing the RPC). When designing an RPC system, one naturally considers implementing it using standard send and receive operations, where the client sends a message to the server and the server simply receives it.

Our first approach is described in Figure 2, which was used in the initial implementation of DAOS. In this approach the server first preposts N receive buffers to handle incoming RPC requests. Each buffer is of fixed size R_u, which is the maximum size of what we refer to as *unexpected messages* from the RPC system standpoint. Preposting is important so that messages (RPC requests) received do not get also treated as unexpected messages by the underlying

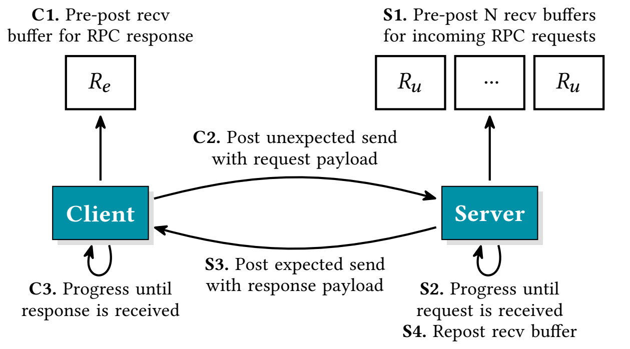

C1. Pre-post recv buffer for RPC response
S1. Pre-post N recv buffers for incoming RPC requests
R_e
R_u
…
R_u
C2. Post unexpected send with request payload
Client
Server
S3. Post expected send with response payload
C3. Progress until response is received
S2. Progress until request is received
S4. Repost recv buffer

Figure 2: Handling of RPC receive buffers (expected and unexpected). For simplicity send buffers are not represented.

network protocol. This situation would mean in libfabric's case that the underlying provider has to either buffer them (resulting in an expensive copy when a new matching receive is posted) or, worse, drop them (in which case the client would time out and have to retry the operation until it can be received). In this context, message sends for the RPC requests, while they are considered as unexpected, can be received efficiently only by preposting a sufficient number of receive buffers to handle them. Note that RPC responses on the other hand are always referred to as *expected messages*, since the client already knows that it will be waiting for a response message from a specific server (the one that it sent the RPC request to). Thus, a receive buffer is preposted also on a client for each RPC request to accommodate that before the request is sent. As shown in that same figure, because sends and receives are non-blocking, both client and server must manually progress the network until the messages are asynchronously received. Once an RPC request has been processed by the server and a response sent back to the client, the server can then repost its receive buffer to handle the next incoming RPC request.

Constraints. This approach has multiple constraints. First, the consumption of receive buffers by the server is independent of the actual size of the RPC request, which means that a client sending a request with a zero-size payload will still consume a full R_u-size buffer on the server. Second, the server must always have posted N receive buffers to handle incoming RPCs, where N is the maximum number of client RPCs that the server can concurrently handle before the underlying provider starts treating them as unexpected messages. Third, there is only a certain limit to which the server can scale up the number of receive buffers that get posted since network resources are limited. In the case of Slingshot (assuming the message matching mode `RX_MATCH_MODE` is `hybrid`), message matching begins fully offloaded to hardware; if resources become exhausted, hardware will transition message matching to a hybrid of hardware and software matching, which increases latency.

3.2 Multi-Receives

To address the constraints of standard receives, we use a feature that is exposed by libfabric and referred to as *multi-receive* (or *multi-recv*).

3.2.1 Design. Multi-receive allows the server to post a single receive buffer that can be used to receive multiple incoming messages. As described in Figure 3, instead of posting N receive buffers of size R_u each, the server can post a single buffer or M buffers of size $N \times R_u$, which can then each be used to receive at least N incoming RPC requests (if the size of each request is R_u) or up to $N \times \frac{R_u}{R_{header}}$, where R_{header} is the size of the RPC header. As messages are received and the buffer gets filled up (meaning the available remaining space in the buffer is less than R_u), the server can then repost the buffer once it is no longer in use. Not only does this approach allow for a better utilization of the receive buffers by consuming only what is needed for each incoming message, but it also reduces the number of receives that the server has to post to handle incoming RPC requests. Furthermore, it reduces the number of times the server has to repost receive buffers. This action improves performance by significantly reducing how often the server must interact with the underlying network stack (reducing the number of hardware commands). Furthermore, by reducing the number of receives that the server has to post, we can more easily remain in line with available network resources and, as we scale up, keep the total number of receives posted to a minimum so that we remain within the limits of what the underlying network provider can support (and in the case of Slingshot remain in a `hardware` matching mode, meaning it uses only hardware resources to match incoming messages against posted receives).

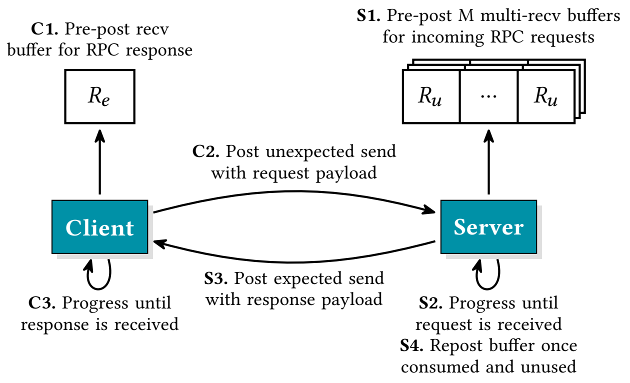
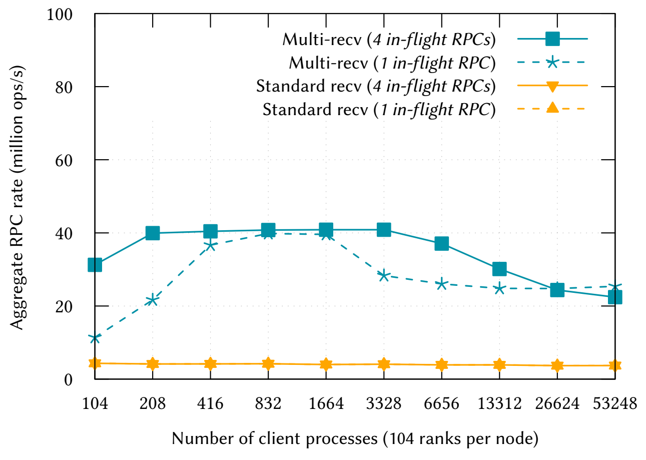
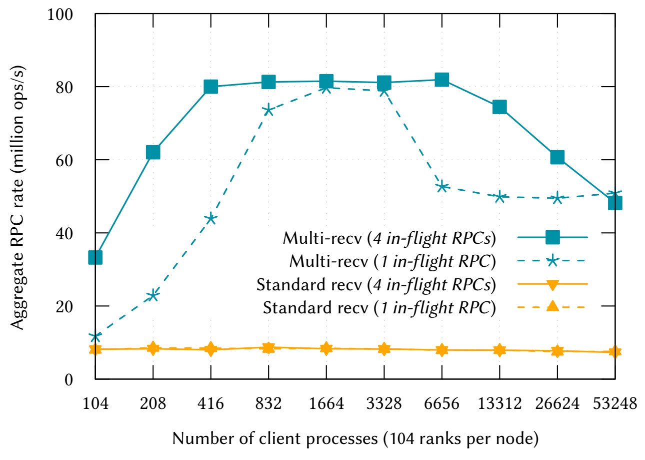

C1. Pre-post recv buffer for RPC response
S1. Pre-post M multi-recv buffers for incoming RPC requests
R_e
R_u
…
R_u
C2. Post unexpected send with request payload
Client
Server
S3. Post expected send with response payload
C3. Progress until response is received
S2. Progress until request is received
S4. Repost buffer once consumed and unused

Figure 3: Handling of RPC receive buffers using multi-receive (expected and unexpected). For simplicity send buffers are not represented.

Constraints. One of the biggest constraints is that a multi-recv buffer can only be reposted once it is no longer in use, meaning the buffer space associated to the RPC request has been reclaimed (once the RPC payload has been processed). Thus, if the server is not able to repost the buffer in time, it will not be able to receive any more incoming messages until the buffer is reposted. This situation can lead to a risk of buffer starvation where we effectively have all allocated buffers in use without any receive posted. One simple workaround would be to post more than a single multi-recv buffer (and there should be at least two to rotate), Because multi-recv buffers are larger, however, we have to be mindful of the memory used, and we do not want to fall into the same constraints



(a) 1 server node.



(b) 2 server nodes.

Figure 4: Standard receive and multi-receive aggregate RPC rates using DAOS server node configurations (2 NUMA nodes / 1 server process with 16 threads per NUMA node).

as the standard receive approach. We therefore implemented a fallback mode where, when we have only one multi-recv buffer left available, the RPC payload gets copied to free up multi-recv buffer space, which allows the server to always repost the buffer and continue receiving incoming messages without having to wait for the payload to be processed. While this has an induced cost since the payload has to be copied (as opposed to being used directly from the network buffer), it has allowed us to maintain a stable system and ensure that the server can continue to operate even in cases of high server load.

3.2.2 Evaluation. We evaluate the performance of this approach by comparing the RPC rate of servers and switching between both standard receive and multi-recv approaches. All tests are run on the Aurora system using mercury 2.4.0 (used in DAOS 2.6.2), libfabric 1.22.0, and the `cxi` provider.

Single-threaded server. First, as represented in Figure 5, we use a single-threaded server process, switching between the two methods and scaling the number of clients up to 512 nodes ($104 \times 512 = 53,248$ processes). In this case we can already see that the multi-recv approach allows for higher throughput, remaining above 1 million ops/s up to $6,656$ client processes, while the standard receive approach provides only 0.6 million ops/s for the same level. Note also that while the multi-recv approach uses 4 multi-recv buffers of 2 MB each that never get filled up in this case, the standard receive approach uses 512 receive buffers of 4 KB each that may get consumed when reaching 512 simultaneous ranks (though this also depends on how packets are routed and the server processing time). In the standard receive case, the server automatically starts posting additional receive buffers to handle incoming RPC requests, potentially reaching the hardware limit (in which case the `cxi` provider switches to a `software` matching mode).

Multithreaded server. To better emulate a typical DAOS use case on Aurora, we consider two scenarios: one where we have a single

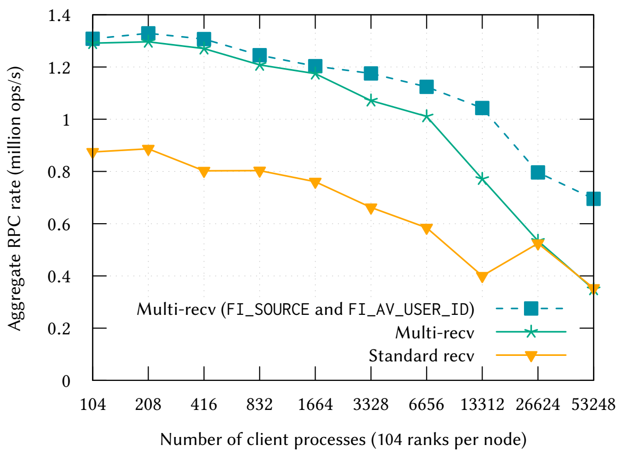


Figure 5: Standard receive and multi-receive aggregate RPC rates with `FI_SOURCE` and `FI_AV_USER_ID` optimization.

server node and one where we have two server nodes, which is sufficient to demonstrate the benefits of this approach. As described in Section 2, each server node is composed of 2 engines running on one socket (each associated with a single NIC), and each engine uses 16 threads that are bound to a single core within that socket (and `cxi` endpoint). Each thread effectively acts as an independent server thread with its own resources and set of multi-recv buffers. We scale up the number of client nodes that are issuing RPC requests to the server threads in a round-robin fashion to evenly distribute the RPC load. The results are shown in Figure 4, which represents the aggregate RPC rate (i.e., the total number of RPC requests processed by all server threads) as a function of the number of processes / client nodes that are issuing RPC requests. We can see that the multi-recv approach allows for an RPC rate one order of magnitude higher compared with the standard receive approach. Notably, as we scale up, we are much more sensitive to network congestion when using the multi-recv approach because the number of NICs issuing RPC requests (which is 8 per client node) is much higher than the number of server NICs (which is 2 per server node). On the other hand, standard receives do not exhibit the same sensitivity because they are already limited by other factors taking a higher fraction of the RPC time (such as the number of receive buffers that are posted and buffer reposting time). Noticeably also, as we scale up the number of server nodes, the overall aggregate RPC rate doubles, which is expected because we have twice the number of server nodes to handle incoming RPC requests, reaching ≈ 80 million ops/s for the multi-recv approach and ≈ 8 million ops/s for the standard receive approach. Increasing the number of RPC requests in flight (i.e., the number of outstanding RPC requests issued by a client), while helping in saturating the performance when using the multi-recv approach, does not help in the case of standard receives. We can also see that these results are on par with the previous experiment where multi-recv allows for throughput multiplied by the total number of server threads, in this latter case $1.3 \times 32 \times 2 \approx 80$ million ops/s.

3.3 RPC Source Lookup

Another important aspect of the RPC path is the ability to quickly identify the source of an incoming RPC request. This is important because the server must be able to determine the address of the client that sent an RPC request to potentially issue a bulk transfer or send an RPC response back to it. In our initial approach, the server had to manually parse the header of the incoming message to determine the source of the RPC request, which requires additional packing and unpacking and extra address lookups. We now use an optimization within libfabric, which allows for the server to determine the source of an incoming message by using the `FI_SOURCE` and `FI_SOURCE_ERR` capability flags, which control the behavior when reading the completion queue applied to receive operations. More concretely (and some of this is later described in Figure 12) this flag allows for the server to resolve the source address of the incoming message within the network provider, as opposed to our having to manually parse the header of the incoming message. This translates into an improvement that allows the server to directly determine the source of the incoming message; and using the optional `FI_AV_USER_ID` flag, we can also associate a user-defined ID with libfabric's addresses, allowing for an even more direct mapping to the source address without any additional hash table lookup. Performance results are shown in Figure 5, where we can see that using this optimization allows for a higher sustained RPC rate compared with our initial approach.

3.4 Receiver-Not-Ready Protocol

As we previously mentioned, we must always try to ensure that servers have enough receive buffers preposted so that the underlying network provider, and in this case the `cxi` provider, does not have to handle incoming messages unexpectedly, which would result in costly memory copies.

The Receiver-Not-Ready (RNR) protocol more recently introduced by the `cxi` provider is a flow control mechanism that is meant to ensure just that: it can be used to prevent the sender from

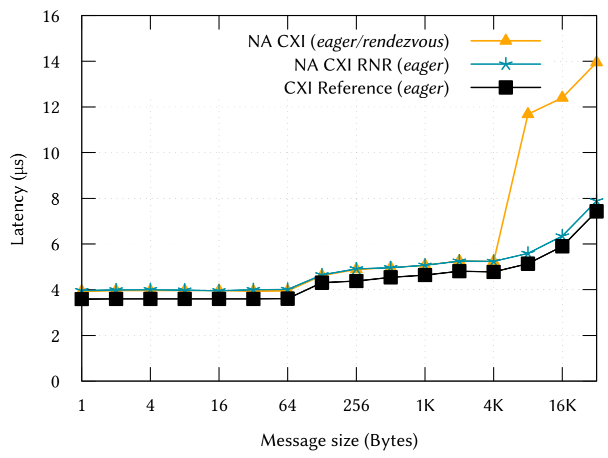


Figure 6: RNR latency comparison.

overwhelming the receiver with data and, in the context of RPCs, from being overwhelmed by incoming RPC requests. The RNR protocol works by sending all messages eagerly until the receiver is no longer ready to receive messages, at which point the sender must stop and retry sending messages until the receiver is ready to receive messages again. One direct consequence of using such a protocol for RPCs is that it is more fit to the eager nature of RPCs, where the data either fits in the RPC buffer or is too large, in which case bulk RPCs are used. While the previous approach could add latency to the RPC path, depending on the rendezvous threshold, the RNR protocol as shown in Figure 6 allows for all RPC data to be sent eagerly, regardless of the message size.

4 Bulk RPCs and Bulk Data Path

Bulk RPCs complement standard RPCs by providing a mechanism for transferring large data payloads over RDMA. In this section we discuss the challenges that accompany RMA operations in the context of a client-server model such as DAOS.

4.1 Bulk RPCs

Bulk RPCs are used for the transfer of data that cannot fit into a standard RPC (i.e., the unexpected message size described in Section 3 would be exceeded). Bulk data may describe data arrays that may be a one- or two-dimensional type of arrays. As shown in Figure 7, to transfer bulk data using RMA, a client must first register the memory region (MR) to be transferred. When an MR is registered, memory gets exposed to the network (read or write), and a key gets associated to the MR. This key must then be exchanged with the server so it can access the now exposed memory. We do this exchange by using a standard RPC, packing along with it what we refer to as a *bulk MR descriptor*, which contains both the MR key and the memory region information (address and length). Once the server receives that RPC, it unpacks the bulk MR descriptor; and having also its own local buffer registered (used to either read or write the data), it can then issue the RMA to the client buffer target. Similar to send and receive operations, RMA operations are non-blocking, and progress must be made on the server until the operation completes.

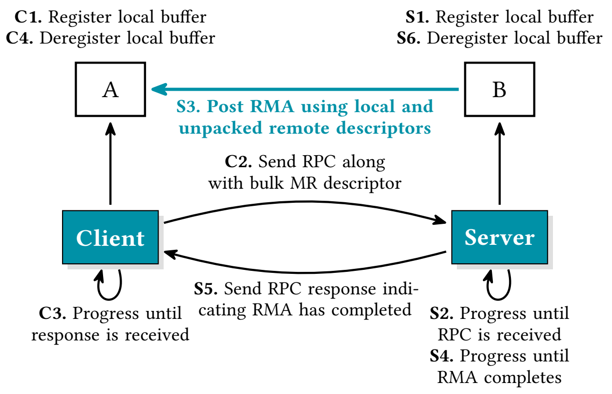

C1. Register local buffer
C4. Deregister local buffer
S1. Register local buffer
S6. Deregister local buffer
A
B
S3. Post RMA using local and unpacked remote descriptors
C2. Send RPC along with bulk MR descriptor
Client
Server
S5. Send RPC response indicating RMA has completed
C3. Progress until response is received
S2. Progress until RPC is received
S4. Progress until RMA completes

Figure 7: Handling of bulk RPCs.

Once the RMA has completed, the RPC response that gets sent back to the client has the additional role of indicating server RMA completion (RMAs being one-sided operations, the client has otherwise no way of knowing when the RMA completes). This is important because only then can the client deregister the MR and potentially free the memory.

Constraints. The bulk RPC mechanism may be constrained by the following factors: (1) the time it takes to register and deregister memory regions, (2) the time it takes to exchange MR descriptors between the client and server, and (3) the time it takes to issue and complete the RMA operation. While (3) typically depends on the network latency and bandwidth, it can also be optimized by having more concurrent RMA operations or, assuming that (2) is not a bottleneck, by having more bulk RPCs in flight. The issue that we will focus on in the next sections is the cost induced from (1). Registration and deregistration of memory regions can be costly, especially when the MR is not reused and is used only for a single RMA operation. This is because the registration and deregistration of MRs typically require calls into the kernel, which can be expensive. While the server MR registration can be done by both preallocating and preregistering local buffers, it is usually impossible to do the same for the client MR registration, since the client application using DAOS in this case is totally unaware of the MR registration process and passes its own allocated memory regions (that then need to be registered) down to the DAOS client library. This is therefore a hot code path that must be optimized.

4.2 MR Caching

MR caching is a technique that can be used to optimize the registration and deregistration of memory regions. In this section we discuss the different MR caching methods that are available in libfabric and the `cxi` provider and evaluate their performance as well as the constraints that they may impose.

4.2.1 Design. In libfabric, MR allocation is performed through the `fi_mr_reg()`, `fi_mr_bind()`, and `fi_mr_enable()` calls, which respectively register a memory region, bind it to a specific endpoint, and enable it for RMA operations, while MR deallocation is done through the `fi_close()` call. We focus on the following MR definitions provided by the `cxi` provider, distinguishing between standard (unoptimized) and optimized MRs specific to Slingshot:

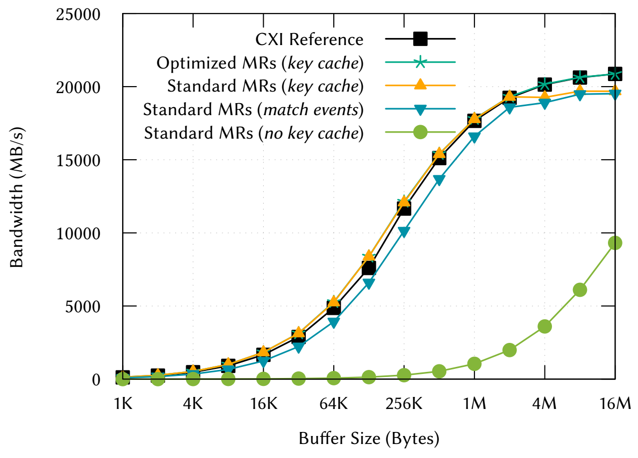
(1) `OPTIMIZED_MRS`. Optimized MRs offer a higher operation rate over standard MRs. However, because optimized MR allocation and deallocation are expensive (i.e., they always require calls into the kernel), optimized MRs should be used only for persistent MRs. This typically maps to MPI/SHMEM RMA windows that are persistent.
(2) `PROV_KEY_CACHE`. The provider key cache is a performance optimization for libfabric `FI_MR_PROV_KEY`, meaning that the provider is responsible for the allocation of MR keys. The `fi_close()` of the MR becomes a no-op in this case at the cost of leaving the corresponding MR exposed to the network. This is intended to be used for applications where `fi_close()` of the MR is on the critical path.

Constraints. For DAOS, since MRs are ephemeral and both the allocation and deallocation may be on the critical path (especially on the client as mentioned above), optimized MRs cannot be used. The limited number of optimized MRs, which have a fixed key range of $[0, 99]$, also presents a risk due to recycling MR keys when using `FI_MR_PROV_KEY`. Multiple regions could end up using the same key by allocating and deallocating the MR, leading to undefined behavior and potential memory corruptions. Additionally, for our use case, leaving MRs exposed (as is the result of using `PROV_KEY_CACHE`) is an issue. When MR operations unexpectedly complete, this may translate into reading from or writing to unknown memory. Unexpected completion can occur when the MR is deregistered before the RMA operation completes, which can happen when, for instance, the client application has its RPC request timing out. In this case the client may deregister the MR and free the memory before the RMA operation completes, without knowing whether the server is still accessing its memory. This can lead to two memory corruption scenarios: (1) the server writing to memory that has either been freed or reused for another purpose, most likely resulting in application crashes, or (2) the server reading from that same memory, resulting in potential data corruption.

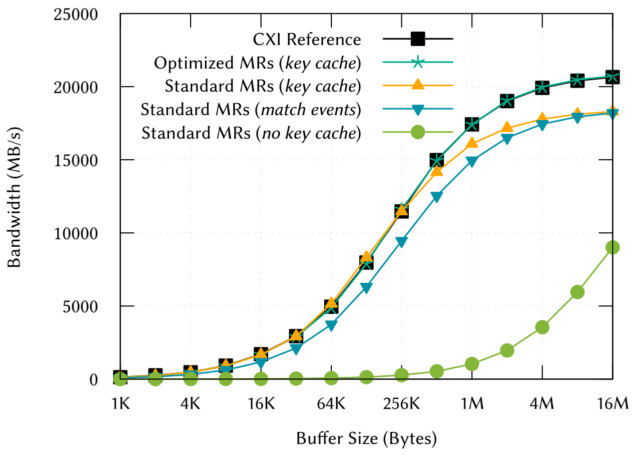
As a middle ground solution, the `cxi` provider extends its MR caching with a method called `MR_MATCH_EVENTS`. While standard MRs do not have a call into the kernel for MR allocation, there is still a call into the kernel for MR deallocation. To avoid this kernel call, MR match events need to be enabled so that the provider can track the number of operations pending and completed against a remote MR. If the pending and completed counts are equal when the MR is closed, the additional cleanup step that is otherwise needed can be skipped. The cost it introduces is where the target of an RMA operation was previously passive (i.e., no events); this will enable MR events. It requires the owner of the MR to process event queues in a timely manner or have large event queue buffers so that MR events can be matched.

4.2.2 Evaluation. For this series of tests, we again make use of the Aurora system using mercury 2.4.0 (used in DAOS 2.6.2), libfabric 1.22.0, and the `cxi` provider.

MR Methods Overheads. We illustrate the performance of the different caching methods previously described by measuring the time spent in registration and deregistration. We use Mercury's network



(a) RMA read.



(b) RMA write.

Figure 8: RMA point-to-point bandwidth using different MR methods.

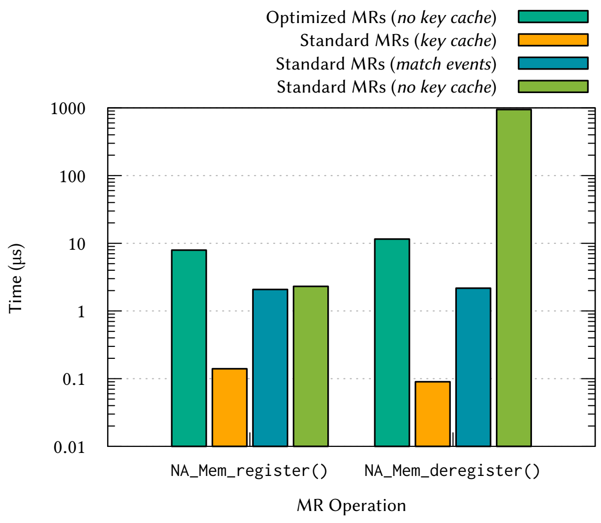
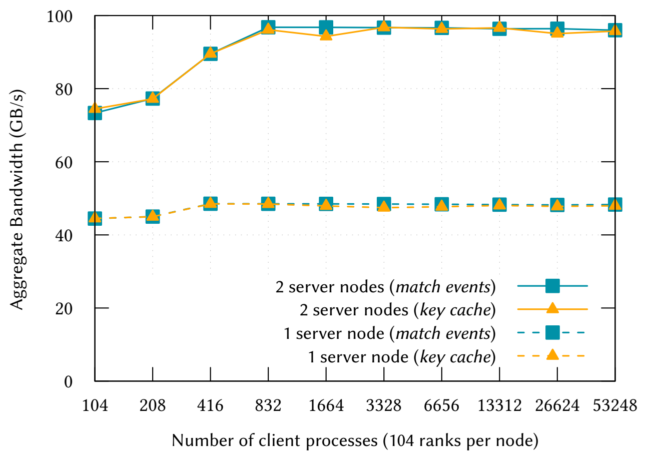


Figure 9: MR register and deregister times using different MR methods.

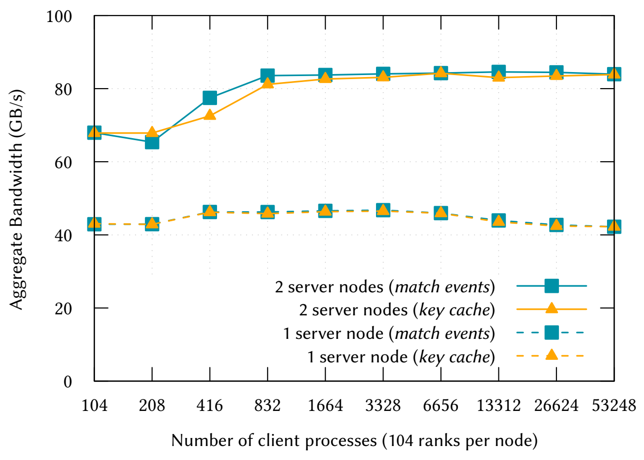
abstraction layer `NA_Mem_register()` and `NA_Mem_deregister()` calls as the unit for measuring the MR registration and deregistration times, respectively. We measure the time spent in these calls for different MR types and caching methods. The results are shown in Figure 9. As expected, using optimized MRs without key caching shows that time is spent in the kernel for both MR allocation and deallocation. When using standard MRs without key caching, the time spent in MR allocation is reduced, although the time spent in MR deallocation is considerably higher. When the provider key cache is used for standard MRs, the time spent in MR deallocation is reduced because the kernel call is avoided, although the MR is left exposed. Note also that in this case the MR allocation time is also reduced. When MR match events are enabled, the time spent in MR deallocation is reduced, because the kernel call is avoided, although both the registration and deregistration costs compared with using the provider key cache remain an order of magnitude higher, on the order of 2 µs.

Point-to-Point Bandwidth. To illustrate how these MR methods more concretely affect the performance of RMA operations, we measure the bandwidth of RMA read and write operations at Mercury's network abstraction layer, varying the combinations of the MR settings previously described. We force both the registration and deregistration of the MRs on every loop while issuing `NA_Put()` (write) or `NA_Get()` (read) calls. The results are shown in Figure 8. We compare the resulting bandwidth against the Slingshot reference baseline, which is measured by using the `cxi_write_bw` and `cxi_read_bw` benchmarks directly against `libcxi` (a layer below libfabric). As expected, using no cache at all with standard MRs is detrimental to the bandwidth in both cases. Using optimized MRs with key caching roughly matches the `cxi` reference, whereas using standard MRs with key caching shows a slight decrease in bandwidth. When MR match events are used with standard MRs, the bandwidth is slightly reduced compared with using key caching, although the difference is not significant and allows reaching the same bandwidth.

Bulk RPC Bandwidth. In this experiment we focus on the resulting aggregate bandwidth of bulk RPCs using the emulated DAOS server configuration that was previously used in Section 3.2.2, using both one and two DAOS server nodes, each node running two server processes of 16 threads each and each thread having again its own set of resources and network endpoint. While the previous experiment was issuing point-to-point RMA calls directly, in this experiment we operate at the Mercury layer, issuing bulk RPCs and forcing the registration and deregistration of the MRs on every loop to simulate a typical application use case. We also scale the number of client processes issuing bulk RPCs, with each client doing its own registration, sending a bulk RPC to the server (along with the MR key), which in turn issues a bulk transfer (RMA read or



(a) Client bulk RPC write / Server RMA read.



(b) Client bulk RPC read / server RMA write.

Figure 10: Aggregate bandwidth of bulk RPCs using standard MRs for 1 MB transfer size.

write) back to the client and, once it completes, responds back to the client, which deregisters its memory. This operation is repeated in a loop using a 1 MB transfer size. We encompass both the registration/deregistration and RPC times in our measurement. Results are shown in Figure 10, where we focus only on the MR methods that use standard MRs with match events and key caching. Using one and two server nodes, we can, with bulk RPC write, where the server issues an RMA read, saturate the network bandwidth (48 GB/s and 96 GB/s, respectively). In the case of bulk reads, the aggregate bandwidth is slightly lower, which was already observed in Figure 8b when using standard MRs. Nonetheless, the aggregate bandwidth is still on the order of 46 GB/s for one server node and 84 GB/s for two server nodes. In this experiment the additional cost of using MR match events is negligible, and the performance is similar to using the provider key cache.

4.3 Transfer Deadlines

We have seen in the preceding section that using MR caching methods can help optimize the registration and deregistration of memory regions at the detriment of leaving memory exposed to the network. These issues, inherent to existing RMA protocols, have already been described in [13]. In the case of `cxi`, preventing unwanted accesses to memory exposed by the MR cache can be done, but it also requires special handling at a potentially higher cost (as seen with MR match events),; and with other existing network protocols (such as `verbs`) there may not be any other option than having the MR cache leave memory exposed. In this section we discuss how transfer deadlines can also be used as a way of palliating unwanted accesses to memory regions—this can be done at the CaRT layer.

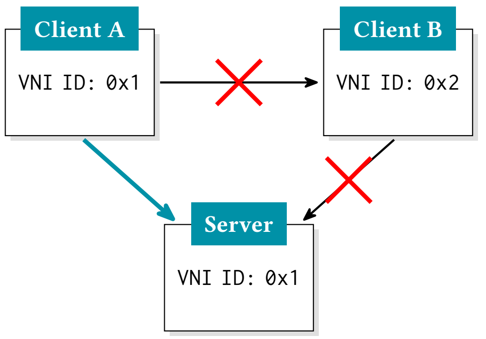
In Section 2 we briefly touched on the fact that CaRT provides a mechanism for handling RPC timeouts and retries. Using this timeout parameter, we can configure CaRT to cancel an RPC once it reaches a certain timeout deadline. When a bulk RPC is canceled by the client, the MR associated to it is also deregistered, leading to the scenarios that we described in Section 4.2 when MR caching is also being used. In this case, because the server may still be alive and processing RPCs, it may still at some point in time attempt to perform the bulk transfer operation requested by the now canceled bulk RPC. In a best effort attempt to prevent this type of scenarios, when MR match events are not available and the memory region remains exposed, we can in fact use the existing timeout information to pass it along with the bulk RPC descriptor. In this way the server can also use this information to set a deadline for its RMA operation and not attempt to access it if the deadline has already been reached. As a rule of thumb, we can set the client and server deadlines so that they account for twice the server timeout, meaning $D_{server} = T_{timeout} + T$, where $T_{timeout}$ is the server timeout and T is the start time of the bulk RPC operation and $D_{client} = 2 \times T_{timeout} + T$. This method, however, presupposes that both the client and the server have their clocks synchronized or that the timeout is sufficiently large to account for clock skews or potential network slowness.

5 Security

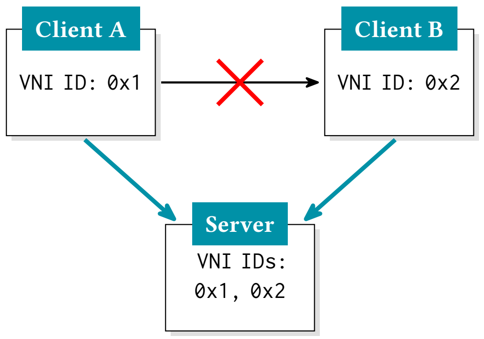
In this section we discuss how the default solution for application isolation defined for Slingshot had to be improved to meet the requirements of a client-server architecture. Although the following discussion is specific to Slingshot, it can be considered for any existing or future network fabric that must provide isolation capabilities.

5.1 Slingshot VNIs and DAOS Requirements

To communicate between peers over Slingshot, the `cxi` provider uses a proprietary address format. This format includes fields for NIC address (topological address of the NIC endpoint on the fabric) and PID (for Port ID or Process ID, also adopted from the Portals 4 specification [1]). While these two components are usually what are considered for source and destination addressing (as we touched on in Section 3.3), a third component of Slingshot network addressing is the Virtual Network ID (VNI). A VNI is a protection key used by the Slingshot network to provide isolation between applications. It defines an isolated port ID space for a given NIC. Using this isolated



(a) Standard VNI use case.



(b) Multiple VNIs use case.

Figure 11: Client-server configurations in a VNI-enabled environment.

space, endpoints must use the same VNI in order to communicate. This guarantees, as depicted in Figure 11a, that, for instance, two independent client applications cannot access each other's data.

In client-server environments, however, and more generally for multitenancy applications where multiple users and applications may interact with the same server or daemon process, the expectation is that independent clients can communicate with the same server. Furthermore, while client-server systems such as DAOS may implement a form of authentication and authorization to ensure that only authorized clients can access the server, none of these solutions fully isolate RDMA traffic from potentially malicious clients. Therefore, network isolation remains a required feature.

By default, the behavior on Slingshot networks is to enable only a single VNI per job, which gets allocated by the job launcher, in Aurora's case, the Parallel Application Launch Service. This VNI then gets handed over to the application through environment variables (`SLINGSHOT_VNIS/DEVICES/SVC_IDS`). This is well suited for an MPI type of jobs, where each process that participates in the job is already well defined. At endpoint creation time, the application may then define through the libfabric API, which VNI to use for that endpoint, or libfabric (and in this case the `cxi` provider) may automatically detect it from the environment. VNIs are specified as part of an *authorization key*, which includes both a VNI and a service ID (a software container defining a set of local Slingshot resources, VNIs, and traffic classes that a libfabric user can access). Since libfabric originally supported only a single authorization key per endpoint, one consideration was to have the server use one endpoint for each client. This is not very scalable, however, given the number of clients that may be connected to a server and the amount of resources available. Also, since DAOS servers operate by having one endpoint per target, changing the existing architecture to a new endpoint definition that relies on clients is not considerable. We therefore augmented the Slingshot infrastructure to enable servers to operate on a block of VNIs as depicted in Figure 11b.

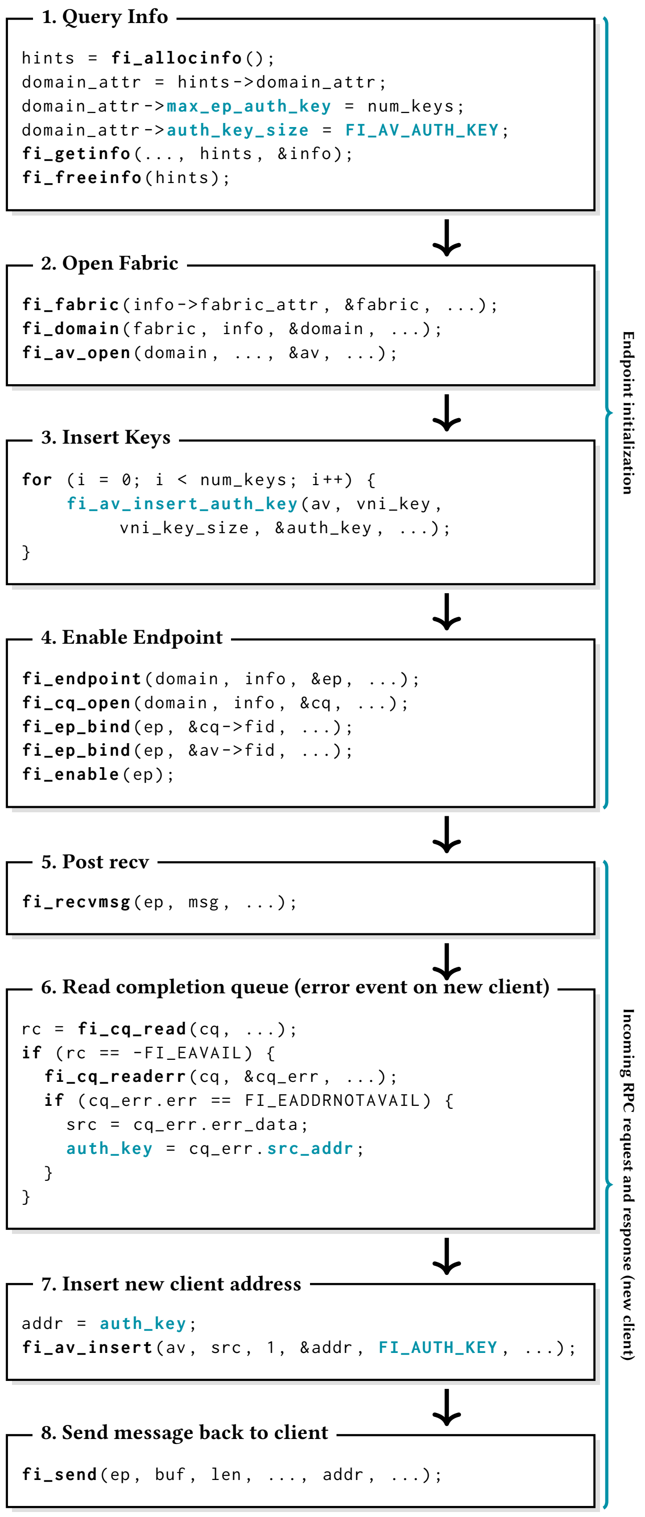
In this mode each server endpoint is configured with a list of the VNIs that it can accept (which is more than the single VNI originally supported by libfabric). The clients, on the other hand, use a VNI that remains specific to them so that RDMA traffic remains isolated. To achieve this, both the Mercury component of DAOS and libfabric had to be augmented to take advantage of that new feature.

5.2 Authorization Keys

As we touched on in the preceding section, libfabric originally supported only a single authorization key per endpoint. By supporting multiple authorization keys, we wanted to be able to associate multiple VNIs to a single endpoint. This feature was needed only for DAOS servers since, in the mode we described, DAOS client endpoints still needed only a single VNI per endpoint. A brief description of this new functionality that was added to libfabric is given in Figure 12. As highlighted in this figure, one of the important considerations when defining authorization keys is that all of the keys must be inserted and known before the endpoint gets enabled. Once it is enabled, no more key insertions can be made. Hence, the block of VNIs that clients will potentially use must be preconfigured and cannot be changed once the server has started.

The second difficulty with this type of approach is to retrieve the authorization key of a client when a new RPC request comes in so that the server can effectively respond back to that client. To that end, the approach we followed relies on the `FI_SOURCE` and `FI_SOURCE_ERR` capabilities of libfabric that we described in Section 3.3. Using these capabilities, when we receive an RPC from a new client peer, libfabric generates an error event in its completion queue for the receive operation that was posted (to handle incoming RPC request messages). With this error event, we are able to retrieve the source address information of the peer (allowing us to no longer have to manually pack and unpack it alongside the RPC header, as mentioned in Section 3.3). With this change, we also augment the error information returned by libfabric with the identifier of the authorization key that was previously inserted at initialization time and internally matched within the `cxi` provider against the peer of the incoming message. Using that authorization key, we are then able to seamlessly operate, as the authorization key gets attached to the source address and inserted into libfabric's address vector table. We can then use addresses directly without needing to further consider authorization keys.

Note that once a new client application that uses the same source address comes in, since it may be using a different authorization key, a new error event will be generated, and the address of the peer in libfabric's address vector table will be updated.


1. Query Info
2. Open Fabric
3. Insert Keys
4. Enable Endpoint
Endpoint initialization
5. Post recv
6. Read completion queue (error event on new client)
7. Insert new client address
8. Send message back to client
Incoming RPC request and response (new client)

```
hints = fi_allocinfo();
domain_attr = hints->domain_attr;
domain_attr->max_ep_auth_key = num_keys;
domain_attr->auth_key_size = FI_AV_AUTH_KEY;
fi_getinfo(..., hints, &info);
fi_freeinfo(hints);
```

```
fi_fabric(info->fabric_attr, &fabric, ...);
fi_domain(fabric, info, &domain, ...);
fi_av_open(domain, ..., &av, ...);
```

```
for (i = 0; i < num_keys; i++) {
    fi_av_insert_auth_key(av, vni_key,
         vni_key_size, &auth_key, ...);
}
```

```
fi_endpoint(domain, info, &ep, ...);
fi_cq_open(domain, info, &cq, ...);
fi_ep_bind(ep, &cq->fid, ...);
fi_ep_bind(ep, &av->fid, ...);
fi_enable(ep);
```

```
fi_recvmsg(ep, msg, ...);
```

```
rc = fi_cq_read(cq, ...);
if (rc == -FI_EAVAIL) {
  fi_cq_readerr(cq, &cq_err, ...);
  if (cq_err.err == FI_EADDRNOTAVAIL) {
    src = cq_err.err_data;
    auth_key = cq_err.src_addr;
  }
}
```

```
addr = auth_key;
fi_av_insert(av, src, 1, &addr, FI_AUTH_KEY, ...);
```

```
fi_send(ep, buf, len, ..., addr, ...);
```

Figure 12: Server workflow with authorization keys.

6 Related Work and Discussion

We focused in this paper on a number of critical points that relate to data services in general and not only to DAOS: Scalability, performance, and security but also resiliency and ease of deployment are all issues that a data service developer will face (on Aurora and on similar HPC systems). Most of these points have come from lessons learned, but they are also inherent to the fact that data services on HPC systems have been designed thus far mostly either as monolithic file systems running in kernel space (e.g., Lustre with LNet [17]) or as services that do not require the same level of performance, such as monitoring and management services. Regarding the former, this is of course not to say that kernel data services have not faced a similar set of challenges, since scalability and performance issues typically occur as well (e.g., Lustre LNet [8]), but their resolution usually does not benefit user space applications since they get privileged access to resources and are not constrained by the same set of security considerations (e.g., on Slingshot, kernel services can use a dedicated and predefined system service to communicate over; there is no involvement of the application launcher). Regarding the latter, monitoring and management services ordinarily do not rely on an HPC fabric but instead either have their own management interface or use TCP/IP wire protocols (e.g., Kafka-based [7] protocols commonly used to implement pub/sub types of services). In effect, these types of services do not require the same degree of scalability, and the amount of data that they process is much smaller than the one that HPC applications typically need to exchange.

Furthermore, HPC systems and HPC network fabrics are designed with a primary emphasis on MPI applications [6] (though the advent of AI and ML applications has now somewhat shifted this focus, MPI remains the de facto standard for HPC applications). Optimizing a network fabric and network software stack for user space data services is a completely different type of challenge. As we have shown in this paper, users of data services are ephemeral; they come and go. Hence, the number of peers accessing a data service varies significantly over time. This situation is opposed to the more static nature of MPI applications, where the number of peers is fixed at application launch time and remains constant over time. Both messaging and RMA requirements in this case are also different. As we have seen, data services using RMA will typically need to register memory regions dynamically, whereas MPI applications can define memory windows that are more persistent and accessed within an entire communicator. Similarly, because MPI applications rely on communicators, the maximum number of peers involved for communication can also be more clearly predetermined and the amount of resources allocated more easily managed. There are also no additional security needs besides guaranteeing that communication can occur only within the MPI job. User space data services such as DAOS, because they are pushing for the development and consideration by network developers of adapted APIs and modes of operation, are helping democratize user data services.

This work leads us to ensuring that data services can be considered as their own class of applications. Even though DAOS may be seen as a special application middleware library, it follows the same principles as do user space services.

7 Conclusion and Future Work

We have made the following contributions in this paper: we have shown how the DAOS metadata path was enhanced by switching DAOS servers to make use of a multi-receive operation mode, allowing us to considerably increase RPC rate while maintaining a reasonable usage of network resources; we have seen how to diminish memory registration overheads associated with RMA operations in the bulk data path by using on Slingshot an MR caching method that, combined with match events, prevents from leaving memory regions exposed and leading to potential vulnerabilities; and we have shown how, while using DAOS, the security of Aurora applications can be maintained with minimal overheads by augmenting libfabric with a concept that relies on the insertion of multiple authorization keys and allows for multiple jobs operating on separate VNIs to communicate with DAOS servers.

To better ensure flow control, we are now moving DAOS toward the Receiver-Not-Ready (RNR) protocol described in Section 3.4, and we are planning to extend some of the multi-recv work that was done for servers to clients in order to remove our current dependency on tag messaging. Tag messaging has been necessary to guarantee that an RPC response would match a previously sent RPC request. Removing the use of tags could lead to further performance improvements. Based on the work reported in this paper, we are pushing toward ensuring that user space data services are considered much sooner during the design of network fabrics.

Acknowledgments

This work was supported by the U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research, under Contract DE-AC02-06CH11357. This research used resources of the Argonne Leadership Computing Facility, a U.S. Department of Energy (DOE) Office of Science user facility at Argonne National Laboratory and is based on research supported by the U.S. DOE Office of Science-Advanced Scientific Computing Research Program, under Contract No. DE-AC02-06CH11357.

References

[1] Ronald Brightwell, William Schonbein, Kevin Pedretti, Karl Scott Hemmert, Arthur B. Maccabe, Ryan E. Grant, Brian W. Barrett, Keith Underwood, Rolf Riesen, Torsten Hoefler, et al. 2022. *The Portals 4.3 Network Programming Interface.* Technical Report. Sandia National Lab. (SNL-NM), Albuquerque, NM (United States). doi:10.2172/1875218

[2] Abhinandan Das, Indranil Gupta, and Ashish Motivala. 2002. SWIM: scalable weakly-consistent infection-style process group membership protocol. In *Proceedings International Conference on Dependable Systems and Networks.* 303–312. doi:10.1109/DSN.2002.1028914

[3] Matthieu Dorier, Philip Carns, Kevin Harms, Robert Latham, Robert Ross, Shane Snyder, Justin Wozniak, Samuel K. Gutiérrez, Bob Robey, Brad Settlemyer, Galen Shipman, Jerome Soumagne, James Kowalkowski, Marc Paterno, and Saba Sehrish. 2018. Methodology for the Rapid Development of Scalable HPC Data Services. In *2018 IEEE/ACM 3rd International Workshop on Parallel Data Storage & Data Intensive Scalable Computing Systems (PDSW-DISCS).* 76–87. doi:10.1109/PDSW-DISCS.2018.00013

[4] Argonne Leadership Computing Facility. 2024. Aurora Fact Sheet. Online. https://www.alcf.anl.gov/sites/default/files/2024-07/Aurora_FactSheet_2024.pdf

[5] Paul Grun, Sean Hefty, Sayantan Sur, David Goodell, Robert D. Russell, Howard Pritchard, and Jeffrey M. Squyres. 2015. A Brief Introduction to the OpenFabrics Interfaces – A New Network API for Maximizing High Performance Application Efficiency. In *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects.* 34–39. doi:10.1109/HOTI.2015.19

[6] Krishna Kandalla, Kim McMahon, Naveen Ravi, Terry White, Larry Kaplan, and Mark Pagel. 2023. Designing the HPE Cray Message Passing Toolkit Software Stack for HPE Cray EX Supercomputers. In *Proceedings of the Cray User Group* (Helsinki, Finland). https://cug.org/proceedings/cug2023_proceedings/includes/files/pap144s2-file1.pdf

[7] Jay Kreps, Neha Narkhede, Jun Rao, et al. 2011. Kafka: A distributed messaging system for log processing. In *Proceedings of 6th International Workshop on Networking Meets Databases (NetDB)* (Athens, Greece), Vol. 11. 1–7.

[8] Zhen Liang. 2010. Lustre SMP Scaling Improvements. Online. https://wiki.lustre.org/images/6/66/Lustre_smp_scaling_LUG_2010.pdf

[9] Zhen Liang, Johann Lombardi, Mohamad Chaarawi, and Michael Hennecke. 2020. DAOS: A Scale-Out High Performance Storage Stack for Storage Class Memory. In *Supercomputing Frontiers*, Dhabaleswar K. Panda (Ed.). Springer International Publishing, Cham, 40–54. doi:10.1007/978-3-030-48842-0_3

[10] Robert B. Ross, George Amvrosiadis, Philip Carns, Charles D. Cranor, Matthieu Dorier, Kevin Harms, Greg Ganger, Garth Gibson, Samuel K. Gutierrez, Robert Latham, Bob Robey, Dana Robinson, Bradley Settlemyer, Galen Shipman, Shane Snyder, Jerome Soumagne, and Qing Zheng. 2020. Mochi: Composing Data Services for High-Performance Computing Environments. *Journal of Computer Science and Technology* 35, 1 (01 Jan 2020), 121–144. doi:10.1007/s11390-020-9802-0

[11] Duncan Roweth, Greg Faanes, Marten Terpstra, and Jesse Treger. 2022. HPE Slingshot Launched into Network Space. In *Proceedings of the Cray User Group* (Monterey, CA). https://cug.org/proceedings/cug2022_proceedings/includes/files/pap121s2-file1.pdf

[12] Pavel Shamis, Manjunath Gorentla Venkata, M. Graham Lopez, Matthew B. Baker, Oscar Hernandez, Yossi Itigin, Mike Dubman, Gilad Shainer, Richard L. Graham, Liran Liss, Yiftah Shahar, Sreeram Potluri, Davide Rossetti, Donald Becker, Duncan Poole, Christopher Lamb, Sameer Kumar, Craig Stunkel, George Bosilca, and Aurelien Bouteiller. 2015. UCX: An Open Source Framework for HPC Network APIs and Beyond. In *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects.* 40–43. doi:10.1109/HOTI.2015.13

[13] Arjun Singhvi, Aditya Akella, Dan Gibson, Thomas F. Wenisch, Monica Wong-Chan, Sean Clark, Milo M. K. Martin, Moray McLaren, Prashant Chandra, Rob Cauble, Hassan M. G. Wassel, Behnam Montazeri, Simon L. Sabato, Joel Scherpelz, and Amin Vahdat. 2020. 1RMA: Re-envisioning Remote Memory Access for Multi-tenant Datacenters. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication* (Virtual Event, USA) *(SIGCOMM '20).* Association for Computing Machinery, New York, NY, USA, 708–721. doi:10.1145/3387514.3405897

[14] Jerome Soumagne, Philip H. Carns, and Robert B. Ross. 2020. Advancing RPC for Data Services at Exascale. *IEEE Data Engineering Bulletin* 43 (2020), 23–34. http://sites.computer.org/debull/A20mar/p23.pdf

[15] Jerome Soumagne, Jordan Henderson, Mohamad Chaarawi, Neil Fortner, Scot Breitenfeld, Songyu Lu, Dana Robinson, Elena Pourmal, and Johann Lombardi. 2022. Accelerating HDF5 I/O for Exascale Using DAOS. *IEEE Transactions on Parallel and Distributed Systems* 33, 4 (2022), 903–914. doi:10.1109/TPDS.2021.3097884

[16] Jerome Soumagne, Dries Kimpe, Judicael Zounmevo, Mohamad Chaarawi, Quincey Koziol, Ahmad Afsahi, and Robert Ross. 2013. Mercury: Enabling Remote Procedure Call for High-Performance Computing. In *2013 IEEE International Conference on Cluster Computing (CLUSTER).* 1–8. doi:10.1109/CLUSTER.2013.6702617

[17] Feiyi Wang, H Sarp Oral, Galen M Shipman, Oleg Drokin, Di Wang, and He Huang. 2009. *Understanding Lustre Internals.* Technical Report. Oak Ridge National Lab. (ORNL), Oak Ridge, TN (United States). National Center for Computational Sciences (NCCS). doi:10.2172/951297

[18] Judicael A. Zounmevo, Dries Kimpe, Robert Ross, and Ahmad Afsahi. 2013. Using MPI in High-Performance Computing Services. In *Proceedings of the 20th European MPI Users' Group Meeting* (Madrid, Spain) *(EuroMPI '13).* Association for Computing Machinery, New York, NY, USA, 43–48. doi:10.1145/2488551.2488556