Summer 2019

A Fully Userspace Remote Storage Access Stack

Patrick Ian MacArthur
University of New Hampshire, Durham

A FULLY USERSPACE REMOTE STORAGE ACCESS STACK

BY

Patrick I. MacArthur

B.S. Computer Science, University of New Hampshire (2012)

DISSERTATION

Submitted to the University of New Hampshire
in Partial Fulfillment of
the Requirements for the Degree of

Doctor of Philosophy

in

Computer Science

September, 2019

This dissertation was examined and approved in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Science by:

Dissertation director, Robert D. Russell, Associate Professor of Computer Science

Radim Bartos, Professor of Computer Science

Philip Hatcher, Professor of Computer Science

Robert Noseworthy, Chief Engineer, InterOperability Laboratory

Bernard Metzler, Technical Leader, High Performance I/O IBM Research–Zürich

On Friday, June 28, 2019

ACKNOWLEDGMENTS

I would like to thank my advisor, Robert Russell, for his support, guidance, and encouragement through the process of reaching candidacy and completing this dissertation. I would also like to thank my committee for their time and input. In particular, I would like to thank Bernard Metzler for his insight and for taking me on as an intern at IBM Research in Zürich for several months during the beginning of 2016. Additionally, I would like to thank Jonas Pfefferle, Animesh Trivedi, and Patrick Stuedi from the RDMA team at IBM Research for their input towards urdma.

I would like to thank the University of New Hampshire InterOperability Laboratory (UNH-IOL) for the use of their RDMA cluster and nodes from the Linux Foundation OPNFV cluster for the development, maintenance, and testing of urdma. In particular, I would like to thank Bob Noseworthy and Timothy Carlin for their guidance and support over the years. I would also like to thank Kyle Ouellette for his flexibility and encouragement in the final stages of preparing this dissertation.

This material is based in part upon work supported by the National Science Foundation under Grant No. OCI-1127228 and under the National Science Foundation Graduate Research Fellowship under Grant. No. DGE-0913620, and work supported by IBM Research in Zürich.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

A FULLY USERSPACE REMOTE STORAGE ACCESS STACK

by

Patrick I. MacArthur

University of New Hampshire, September, 2019

As computer networking has evolved and the available throughput has increased, the efficiency of the network software stack has become increasingly important. This is because the latency introduced by software has gone from insignificant, compared to historically poor network performance, to the largest component of latency for a modern local-area network. Currently, the vast majority of code that accesses the hardware is part of the kernel, because the kernel is responsible for ensuring that user applications do not interfere with each other when accessing the hardware. Remote Direct Memory Access (RDMA) provides a solution for applications to perform direct data transfers over the network without requiring context switches into the kernel, but relies instead on specialized hardware interfaces to handle the virtual address mappings and transport protocols. This more intelligent hardware allows for direct control from the userspace application, eliminating the cost of context switches into the kernel. This in turn reduces the overall latency of message transfers.

Just like networking, storage is currently undergoing a similar evolution. For most of the recent history of computing, the most common durable storage mechanism has been mechanical hard disk drives, which can only be accessed at block level and have high latency compared to the software drivers used to access the data. However, the introduction of solid state disks (SSDs) based on Flash significantly decreased the latency, as there are no mechanical parts that need to move to access the data. Upcoming non-volatile memory solutions reduce this latency even further, and even allow byte-level access to the storage medium. Thus, just like with networking, software drivers become the bottleneck and we look for solutions to bypass the kernel to improve the efficiency of direct

userspace access to storage.

This thesis offers two contributions as part of a solution to these problems. The first part introduces urdma, a software RDMA driver which leverages the Data Plane Development Kit (DPDK) to perform network data transfers in userspace without specialized RDMA interface hardware. The second part examines remote locking protocols, which are required for synchronization in distributed storage systems. We define an RDMA locking mechanism referred to as Verbs Offload Locking Technology (VOLT), which allows acquisition of a remote lock object without any CPU usage by the target node. This offloading allows VOLT to be used with disaggregated memory servers that have limited onboard CPU resources, while also lowering the application overhead for remote locking. Finally, we define a bytecode framework using enhanced Berkeley Packet Filter (eBPF) bytecode for extending the capabilities of an RDMA-capable network interface card (NIC) with new operations, and show how this can be used to implement our remote locking operation.

Chapter 1

INTRODUCTION

This chapter introduces the concepts of storage networking and the problems that this dissertation is aiming to solve. Section 1.1 discusses the evolution of distributed file systems (DFSs) to meet high-level scalability, transparency, and performance goals. Section 1.2 discusses the history of storage networking protocols at a lower level, looking at the challenges introduced by increases in networking and storage performance. Section 1.3 discusses emerging storage-class memory technologies and how these change the traditional filesystem model. Finally, Section 1.4 discusses the problem areas that this dissertation seeks to investigate.

1.1 Evolution of Distributed File Systems

From the beginning of computer networking, one of the biggest problems has been making data stored in durable storage on one computer available on other computers. A storage medium is *durable* if data written to it is preserved even when power is removed, as opposed to temporary storage provided by current Random Access Memory (RAM) technologies. To this end, file transfer protocols such as UUCP [1] and FTP [2] were developed in the early history of computer networking. However, these protocols focused on transferring a file from one node to another in a non-transparent manner—users had to explicitly use these protocols to transfer data. While this worked well for long-distance transfers between mainframes, as inexpensive workstation computers became popular, this was increasingly inefficient for frequently accessed data on a local area network (LAN).

To that end, Sun Microsystems released the Network File System (NFS) [3], which allowed workstations to transparently mount a remote filesystem into their local filesystem. In turn, this meant that users of UNIX systems could access remote files as though they were local. Around the same time, Netware developed Netware Core Protocols [4] and Microsoft developed the Server

Message Block (SMB) [5] which allowed remote files to be accessed via a virtual drive on MS-DOS and Windows systems.

While this solved the problem for end systems, system administrators had to deal with the problem of managing the increasing amounts of storage tied to file server nodes. RAID [6] was developed to allow the operating system to present a single large logical disk backed by multiple smaller disks. Most RAID schemes involve striping data across each disk, such that block 0 is on the first disk, block 1 is on the second disk, and so on until block n wraps around to the first disk again. Note that when distributing a datastore onto multiple disks, each disk added to a RAID decreases the Mean Time to Failure (MTTF) of the system because only a small number of disks need to fail in order to bring down the system. Different RAID levels provide varying levels of protection, and thus more or fewer disks can fail before data is no longer accessible. For RAID 0, where data is striped with no parity or other error correction mechanism, even a single disk failure will bring down the system. However, multiple parity and error-correcting code (ECC) schemes have been developed for RAID which increase the number of disks which may fail without bringing down the whole disk system. For example, the modern RAID 6 scheme uses Reed-Solomon encodings to allow up to two drives to fail without bringing down the system.

However, attaching many disks to a file server still has scalability problems in terms of power usage and maintenance costs. The Fibre Channel protocol [7, 8] was developed to allow direct block access to disks on a remote computer via the standard Small Computer Systems Interface (SCSI) protocol [9], as opposed to the filesystem-level access offered by previously developed protocols. This in turn allowed the development of *storage targets*, which are purpose-built nodes on the network which offer block storage over a storage-area network (SAN), and decouple the file server nodes from their disks. This decreases the power requirements for the file servers and makes it easier to add more storage space to a network on the fly. However, Fibre Channel used its own physical layer protocol which is incompatible with the more widely used TCP/IP[1] and Ethernet, and thus the equipment for Fibre Channel is expensive and requires specialized knowledge to operate. The

[1]TCP/IP refers to the protocol stack combining two separate but related protocols, Transmission Control Protocol and Internet Protocol

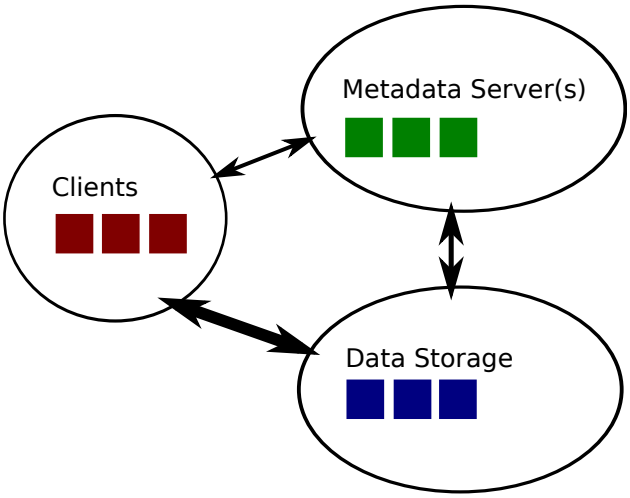

Figure 1-1: The typical structure of a distributed file system.

iSCSI protocol [10] is a transport for SCSI built atop TCP, and allows a SCSI initiator to send commands to a SCSI target over a commodity network. Later, Fibre Channel over Ethernet [11] was developed to allow use of the Fibre Channel protocol over Ethernet networks, which in theory allows Fibre Channel and standard TCP/IP traffic to co-exist on a single network.

The problem with this arrangement is that all access to a particular network filesystem is done through a central file server, which is both a single point of failure and a performance bottleneck. To solve this, the developers of next-generation distributed file systems sought to give clients direct access to the nodes containing the file data; thus avoiding the bottleneck of transferring large amounts of file data through a central file server. However, while the Fibre Channel and iSCSI protocols technically allow concurrent access to a single target by multiple initiators, they provide no synchronization mechanism between clients. Thus, these protocols cannot be used by themselves as a direct storage protocol for shared storage without a very high risk of data corruption due to multiple clients writing the same file at the same time. A distributed filesystem whose backend storage may be directly accessed by multiple clients must thus provide its own synchronization facilities.

The current generation of distributed file systems, including Google File System [12], Hadoop Distributed File System [13], Lustre [14], Ceph [15], and GPFS [16], provide such synchronization

systems. These distributed file systems are divided into three components as shown in Figure 1-1: data storage servers, metadata servers, and client nodes. Client nodes are able to directly access data storage servers, but must first contact metadata server(s) which inform the client which data storage servers hold the file data and handle the synchronization between multiple clients accessing the same file. However, the metadata servers otherwise are *not* involved in any transfers of file data. File data is stored in units called blocks or chunks. Different blocks within the same file may be stored on different data storage nodes, and each block may be itself replicated onto several data storage nodes. The metadata servers keep track of the location, permissions, and timestamps of each file, and possibly perform other control service roles. Finally, the clients are the systems actually accessing the data in the distributed file system. Typical operations in a DFS require the client to first talk to the metadata server to find the location of the chunks involved, and then send/receive data to/from the corresponding data storage nodes.

1.2 Storage Networking

Storage and network technologies have been related but defined by different standards bodies for much of the history of computing. The standards defining the Small Computer Systems Interface (SCSI) [9] protocols used by enterprise systems are defined by the American National Standards Institute (ANSI) T10 working group[2]. On the other hand, while the Internet architecture is loosely based on the seven-layer Open System Interconnect (OSI) model [17], the networking protocols in common use today are not defined by ANSI or the International Organization for Standardization (ISO)[3]. Instead, the Institute of Electrical and Electronics Engineers (IEEE)[4] defines the Ethernet physical and link layer protocol standards, and the Internet Engineering Task Force (IETF)[5] defines the application, transport, and network layers of the OSI model. This

[2]http://www.t10.org

[3]https://www.iso.org

[4]https://www.ieee.org

[5]https://www.ietf.org

separation between network and storage standards has led to some duplication of effort.

Despite the differences, both networking and storage technologies began with the assumption that both the network and the storage were slow compared to the host CPU. Early network technologies such as Token Ring [18] and pre-10 Gigabit Ethernet [19] could not transmit data fast enough to saturate a host CPU, and conventional hard drives with spinning platters have access latencies in the tens of milliseconds. Thus, the protocols and software drivers did not need to be efficient—rather, they just needed to hide the latency of the underlying hardware via intelligent queuing and buffering.

For storage requests, an operating system would take it upon itself to schedule disk requests in an efficient order using an algorithm such as the elevator algorithm [20], since any delay caused by the queuing would be amortized by decreasing the seek time for requests that the user made later. Additionally, operating systems perform read-ahead of a certain number of disk blocks when a disk block is accessed, under the assumption that the user will request subsequent blocks soon. In common implementations of the TCP networking protocol, incoming and outgoing data is buffered in the kernel, under the assumption that the application will send or receive more data. This means that multiple write() requests to a single socket may be coalesced into a single packet on the wire, making better use of network bandwidth at the expense of latency. These simple examples demonstrate ways in which the operating system sacrifices immediate latency in favor of higher overall system performance when dealing with a slow network or storage device.

However, in the late 1990's and early 2000's, faster network technologies arose, such as 10 gigabit Ethernet [19], Myrinet [21], InfiniBand [22], and others. Since TCP sockets requires the kernel to copy data into the kernel socket buffers, consuming CPU time equivalent to the amount of network data transferred, these new faster network technologies could easily provide enough bandwidth to saturate a host CPU. Thus, the software techniques previously used to hide network performance issues are no longer effective but instead increase the network latency as well as the CPU usage required to service applications using the network. TCP Offload Engines (TOEs) reduce some of this overhead by removing this burden from the host operating system. However, the upstream Linux kernel lacks support for TOEs for several reasons: TOEs remove visibility of

TCP connections from the operating system, any bugs in TOE firmware are harder to fix than software bugs in the kernel TCP stack, and previous generations of TOEs have eventually been outperformed by improvements to the software kernel stack [23].

Rather than simply shifting the burden of TCP onto hardware, modern high performance networks have moved to other protocols and software stacks avoiding the performance problems of TCP sockets altogether. Remote Direct Memory Access (RDMA) [24] provides a method for applications to perform data transfers to and from a remote application's virtual memory without involving the host CPU. In particular, applications can perform network data transfers without involving the kernel. RDMA is supported by the InfiniBand [22] protocol stack, along with RDMA over Converged Ethernet (RoCE) [25] which implements the transport and application layer protocols of InfiniBand on top of Ethernet, and the IETF-developed iWARP[6] [26, 27, 28] protocol which implements an RDMA protocol stack on top of existing TCP/IP. RDMA has become popular in the high-performance computing space.

However, RDMA requires specialized hardware and network protocols which can place data at an arbitrary virtual memory address. Software RDMA drivers, such as softiwarp [29] and softroce [30] allow the use of RDMA semantics without specialized hardware, but require a kernel driver to process incoming requests. This cannot provide the same level of performance as a hardware solution because software must perform the data processing that would otherwise be offloaded to hardware. However, software implementations have three major uses: (i) research and experimentation for new RDMA features, (ii) testing and debugging of RDMA applications without access to RDMA-capable hardware, or (iii) as a client endpoint for an RDMA server that uses real hardware. The last use case requires the implementation to interoperate with existing implementations, but the former two use cases do not.

Due to the nature of conventional mechanical hard disk drives, storage has lagged behind networking in terms of performance. However, design of solid-state disks (SSDs) based on NAND

[6]iWARP is not officially an acronymn in any of the standards issued by the IETF. Some sources claim that the name expands to Internet Wide-Area RDMA Protocol, but this is likely a "backronym" that was not intended by the committee that developed the protocols.

Flash has improved over the last 15 years to the point of replacing hard disk drives on both consumer and enterprise systems [31]. SSDs have no moving parts, completely removing the seek time latency component due to the disk having to seek to the correct track and rotate to the correct sector, which has in turn reduced the latency to tens of microseconds. This means that the classic elevator algorithm no longer improves efficiency for SSDs, so operating systems and even applications can enjoy better performance by directly submitting commands to queues on the SSD controller. However, NAND Flash cells wear out much faster than hard disk drive platters, which requires writes to be distributed across the disk using wear leveling techniques. Thus, SSD controllers have a Flash Translation Layer (FTL) which translates logical block addresses (LBAs) used by the operating system into the actual physical addresses of the Flash cells. The wear leveling, in turn, requires garbage collection of Flash cells that are no longer in use by the operating system. This means that the operating system (or application directly accessing the SSD) must inform the drive controller when a data block is no longer used by using a TRIM command, which marks the cell available again for the SSD to use.

These problems have led to the creation of Non-Volatile Memory Express (NVMe), a technology which allows direct attachment of SSDs to the PCIe bus, instead of to the SCSI or ATA buses used by hard disk drives. The NVMe specification uses command queues and completion queues heavily inspired by RDMA [32]. NVMe's command set is tuned for the requirements of SSDs; a command called TRIM allows an operating system or application to indicate that a block is no longer in use and may be garbage collected. The NVM over Fabrics protocol [33] allows remote block access to an NVMe device, and is analogous to iSCSI, but is implemented in terms of RDMA.

1.3 Storage Class Memory

Several new technologies completely change the landscape of durable storage in two ways: by providing throughput and latency only slightly worse than dynamic RAM (DRAM) [34], and by providing byte-level access as opposed to the block-level access offered by existing storage technologies. These are collectively referred to as storage class memory (SCM), and include technologies such as phase change memory (PCM), state transfer torque (STT-RAM), and memristors [31]. This requires a

transformation for storage protocols and drivers similar to that caused by RDMA for networking. In particular, it is now possible to support direct CPU load/store access to durable storage. With proper transaction memory support, applications can enjoy reads and writes to durable storage at minimal cost compared to DRAM access.

This means that the interface to durable storage must look more like a virtual memory system than a file system on a block storage device. In particular, an application will appear to have direct access to the underlying storage, instead of using system calls to read/write data via a file system. Current operating systems provide an equivalent of the *mmap* system call defined by the Portable Operating System Interface (POSIX) [35], which maps a range of bytes in a file into application virtual memory by leveraging the page cache. However, for SCM, the intermediate operating system page cache is not required nor desired, as it would require operating system involvement to load pages into memory and flush them back to disk. Thus, application data structures can be placed directly onto durable storage, and RDMA could be used to copy data structures to durable storage on remote nodes. Any file system abstraction will be built on top of this virtual memory architecture, as in BPFS [36].

While one usage of SCM is as a DRAM replacement, another area of research looks into *disaggregated memory* using dedicated memory nodes which are decoupled from compute nodes [37], not unlike how storage targets decoupled disks from file servers a decade ago. This decoupling opens up changes in memory architecture without affecting existing compute node or software architecture, including redundant storage, error-correcting codes at the software level, and virtual machine migration without moving any data.

One goal of using disaggregated memory is to define distributed data structures, and designing efficient synchronization for these remote data structures can be challenging. Synchronizing distributed data structures requires some form of distributed locking, and the efficiency of this locking is an extremely important component of the performance of the overall system. Existing locking solutions rely on sending application-level messages such as remote procedure calls (RPCs) [38] to request or release a remote lock object. This requires that the target CPU process all of these lock and unlock requests. Using RDMA, this processing can be done without involvement from the

target CPU via atomic compare-and-swap operations [22, 39]. However, this puts the burden onto the requesting system to poll the state of the remote lock value until it obtains the lock, increasing network traffic and CPU usage at the requester. This thesis examines a potential solution that relies on the RDMA queue pair itself as a mechanism to block RDMA operations until the lock is acquired, combining the RPC model with the offload capability of RDMA.

1.4 Thesis

This thesis will examine the following:

- What is the impact of kernel overhead on software RDMA and storage access solutions?
  - Can a fully userspace software RDMA driver offer better performance than an in-kernel software RDMA driver?
- Can this be used to define an efficient remote locking protocol that relies on blocking queue pair processing until the lock becomes available?
  - Can this locking protocol be used to implement the locking required for a distributed data structure, such as a B-tree?
  - Can this locking protocol correctly synchronize multiple applications using one-sided RDMA operations on a single remote object (RDMA READ and RDMA WRITE)?
  - Can the remote locking protocol survive the failure of a single node, whether that node is holding the lock at the time of failure or waiting for the lock at the time of failure?
  - Can the locking be used to maintain correct synchronization for multiple objects within a single multiversioned data structure?
  - Can this remote locking protocol be implemented in a bytecode language such that it would be implemented by existing InfiniBand Host Channel Adapters (HCA) [22]?

1.5 Summary

In this chapter, we introduced the concepts of storage networking and the problem of distributed data structures that this dissertation intends to solve. Section 1.1 discussed the evolution of distributed file systems to meet high-level scalability, transparency, and performance goals. Section 1.2 discussed the history of storage networking protocols at a lower level. Section 1.3 discussed emerging storage-class memory technologies and how these change the traditional filesystem model. Finally, Section 1.4 discussed the problem areas that this dissertation seeks to investigate.

Chapter 2

BACKGROUND

In this section, we discuss some background on RDMA, as it is the primary focus of this dissertation. We also discuss background for non-volatile memory (NVM) and distributed data structures, as these are motivating factors for the work done as part of this dissertation.

2.1 RDMA

Modern high performance computing (HPC) clusters use Remote Direct Memory Access (RDMA) to perform network data transfers between nodes' virtual address spaces without kernel involvement. This is done using specialized host channel adapters (HCAs) which offload the network packet processing from the host CPU. RDMA protocols are message oriented, as opposed to the stream-based TCP. A user application uses verbs [22, 40] to perform setup and data transfer operations on an HCA. The verbs do not make any assumptions about the threading mechanism or memory layout of the user application, allowing easy porting of the verbs library and drivers to higher-level languages such as Java [41].

RDMA message transfers are asynchronous with respect to the user application, and applications request data transfer operations by placing *work requests* onto *queue pairs* (QPs). Each QP consists of a send queue and a receive queue; each of which is associated with a *completion queue* (CQ)[1]. Each application on an RDMA fabric uses at least one QP to communicate with the remainder of the fabric. Once an application has posted a work request, the HCA will enqueue the

[1]Multiple QPs may be associated with the same CQ, and the send and receive queue on the same QP may be associated with different CQs, but each queue on the QP may only be associated with a single CQ. It is common for applications to use a 1:1 association between QPs and CQs.

operation to be performed in parallel with the user application and post a *work completion* to the CQ when the operation completes.

RDMA requires the application to inform the HCA which memory regions it will use in data transfers, a process known as *memory registration*. The HCA kernel driver pins the local virtual memory regions into physical memory and obtains the virtual-to-physical address mapping, which allows it to directly access the corresponding physical memory without intermediate data copies.

Once memory has been registered, an application may perform two types of operations on the memory. The SEND and RECV operations provide *channel semantics* similar to sockets. A RECV must be performed first on the receiving side, and then the sender issues a SEND request which transfers the data from virtual memory at the sender to the virtual memory region referenced by the head of the receive queue at the receiver, with no intermediate copies. Using this mechanism, the virtual memory addresses remain anonymous from the perspective of the remote endpoint.

The other operation type provides *memory semantics*, in which a virtual memory region is *advertised* to a remote endpoint through an application-specific mechanism. In most cases, this is done through channel semantics operations. For each memory semantics operation, the *requester* is the endpoint which initiates the operation and the *responder* is the endpoint which responds to the operation. The responder's CPU is not involved in the data transfer, and the application at the responder receives no notification by the RDMA hardware of the operation's completion. There are two memory semantics operations which we will be concerned with: RDMA WRITE and RDMA READ. In the case of RDMA WRITE, the requester pushes data into a memory region advertised by the remote endpoint. Likewise, in the case of RDMA READ, the requester pulls data from a memory region advertised by the remote endpoint, with no CPU involvement at the responder.

In this dissertation, we discuss three standards based RDMA protocols. Infiniband [22] defines its own self-contained network stack, including the application, transport, network, link, and physical layers of the OSI model [17]. RoCE [25] is a mapping of the Infiniband transport protocol over User Datagram Protocol (UDP) [42], IP [43, 44], and Ethernet [19]; our high-level discussion of InfiniBand also applies to RoCE. iWARP [26, 27] was independently developed by the IETF and

runs over TCP/IP.

2.1.1 InfiniBand

InfiniBand defines an entire network stack all the way down to the physical layer tuned for HPC [45, 24]. Its data-link layer protocol is controlled by a centralized Subnet Manager (SM); which allows for automatic addressing and population of forwarding tables in switches. InfiniBand uses credit-based link-by-link flow control, so unlike Ethernet, packets cannot be lost due to congestion. Finally, InfiniBand is designed to be implemented in hardware through the transport layer.

The InfiniBand transport protocol supports several transport services, the most common of which is reliable connected (RC) service. Channel semantic operation is supported on all transport services, while a reliable service is required for memory semantic operations. Applications make use of InfiniBand transport operations via the verbs API [22]. Upon registering a memory region with the verbs API, the HCA returns to the application a local key (*lkey*) and remote key (*rkey*) corresponding to the memory region. The application provides the lkey when it issues transfer operations on local memory; the lkey is never transmitted across the wire. For memory semantic operations, the application must communicate the rkey and the virtual address corresponding to the target memory region from the responder to the requester. The RDMA READ or RDMA WRITE packet headers sent by the requester include the rkey and virtual address. In InfiniBand, the RDMA READ packet sent by the requester is targeted at a specific virtual address, while the response packets are not; this means that the local application memory region at the sender is not exposed to the responder.

2.1.2 iWARP

The iWARP protocol suite describes several related protocols for RDMA. Direct Data Placement (DDP) [27] is used to tag a message with metadata describing the location in application virtual memory in which the message should be placed. RDMA Protocol (RDMAP) [26] describes the high-level SEND/RECV, RDMA READ, and RDMA WRITE operations usable from verbs. Together these protocols implement the RDMA data transfer operations. Unlike InfiniBand, iWARP

is designed to sit atop the TCP/IP protocol stack. While this causes iWARP to inherit some of the issues of TCP, many of these are mitigated by a hardware implementation of TCP specifically optimized for iWARP. The IETF defines MPA (Marker Placement and Alignment) [28] to map the message-based iWARP protocol over TCP [46], which is a byte stream protocol. As an alternative, the iWARP protocol stack also includes an adaptation [47] to map iWARP over SCTP (Stream Control Transmission Protocol) [48], which is message-oriented. However, there are no known implementations of iWARP over SCTP. Because iWARP runs over TCP/IP, several RDMA software emulations have been developed for iWARP, including softiwarp [29] and urdma [49].

DDP has two types of messages. *Tagged messages* contain a *steering tag* (STag) and offset describing a location in a pre-registered memory region at the destination into which to place the message data. The STag is analogous to the rkey used in InfiniBand; thus, the sender must be told the value of the STag for a remote memory region in order to send a tagged message targeted at that memory region. This may be done using *untagged messages*, which are not associated with a specific memory region at the receiver. Instead, untagged messages contain a queue number and message sequence number which is used to identify a destination buffer into which to place the message. Unlike tagged messages, the sender does not need to be aware of this destination buffer in advance; however, the receiver must have buffers queued up to receive these messages in advance.

RDMAP uses tagged and untagged messages to implement verbs transfer operations that are similar to those in InfiniBand. The SEND/RECV operation is implemented using untagged messages on a queue pair made visible to the application. Following the same logic, RDMA WRITE is implemented using tagged messages. RDMA READ requires two DDP messages, since it requires that data flow in the opposite direction of the request. The RDMA READ Request message is implemented as an untagged message from the requester to the responder containing the STag and offset of the sink buffer at the requester and the STag and offset of the source buffer at the receiver. The responder then sends a tagged RDMA READ Response message directed at the sink STag and offset specified in the request, containing the requested data.

2.2 Data structures

2.2.1 Durable data structures

Consistent and Durable Data Structures (CDDS) [50] defines a versioned B-tree structure which provides atomicity, consistency, and durability, but not isolation. This is done using atomic and copy-on-write operations. Each B-tree entry has a minimum and maximum version, and the B-tree as a whole has a current version. The lookup algorithm works on the current version at the time that it was first called, so that it always works on a consistent version of the tree. When an entry is inserted, the tree's version is incremented, the entry's minimum version is set to this new version, and the maximum version is unbounded. Entries are not deleted from nodes until a garbage collection cycle is run. Instead, the maximum version is set, and the old entry is skipped when a lookup is done for a later version.

2.2.2 Distributed data structures

Aguilera, et al., design a concurrent B-tree implementation [51] using Sinfonia [52]. Sinfonia provides a framework for keeping objects in a distributed in-memory database, using a collection of *memnodes* that store objects. These objects are modified via *minitransactions*, which act as a lightweight multi-compare-and-swap operation. Like other transaction systems, minitransactions contain a read set and a write set, and on commit, the object versions are first verified and locked, and then all changes that are part of the transaction are made with the locks held. The objects stored by Sinfonia may optionally be made durable on disk or non-volatile memory.

In Aguilera's B-tree implementation, each B-tree node only represents the latest version of the tree, unlike other implementations in which B-tree nodes can include entries from multiple versions of the tree. Nodes are accessed via the Sinfonia object system which does not provide RDMA access. However, to reduce the load on the memnodes that host the object containing the root or other B-tree nodes near the top of the tree, every memnode keeps a cache of the version of every B-tree node, even the nodes not stored on that particular memnode. This is intended to reduce the load on the memnode which holds the root node, but means that every transaction must modify

metadata stored on every node.

However, the Aguilera B-tree has some disadvantages. Almost every insert or delete operation must update metadata on every memnode since they will increase the version number of one or more nodes. Additionally, the transaction mechanism is based on a read and write set, and if an internal node in the read set is modified during a lookup transaction, the transaction must abort even if the internal node modification had no impact on the lookup transaction. Sowell, et al., provide two optimizations [53]. The first optimization is a proxy layer for client requests which cache all nodes that they access. Additionally, minitransactions are extended with a "dirty read" set for internal nodes read during a lookup transaction. Concurrent modifications to these nodes do not fail the transaction. However, this can cause consistency issues if a node along the lookup path of a transaction is split or merged. Thus, each node includes fence keys indicating the entire range of values which the subtree at each node may hold. This allows the lookup transaction to abort if the lookup reaches a node whose subtree cannot contain the element in question.

Mitchell, et al., implement Cell [54], a distributed B-tree following a different philosophy than Aguilera. Cell allows clients to directly access nodes of the tree using RDMA READ operations, although tree modification must be delegated to the server; clients are not allowed to RDMA WRITE directly to tree nodes. To optimize lookups to the tree, Cell structures the B-tree using *meganodes*, each of which is its own B-tree. This lowers the number of remote objects which must be read to look up an object in the B-tree, as a client can access many B-tree nodes with a single sufficiently large RDMA READ operation. Cell's B-tree uses a variant called a B-link tree [55], in which each B-tree node contains a *right-link* pointer to the next node at the same level, along with a marker indicating the maximum value that the node could hypothetically hold. This allows a client to read a node as part of a lookup while it is being simultaneously split due to an insertion, as the client can follow the right-link pointer to access the values that were previously part of the split node. Additionally, the insertion algorithm used by Cell requires locking only a single node at a time.

2.3 Non-Volatile Memory

2.3.1 NAND flash memory

Traditionally, enterprise, cloud, and high performance computing have all used similar storage systems. A storage cluster would consist of a file system distributed over many hard disk drives (HDDs). However, because HDDs consist of mechanical parts which must move in order to access data, HDDs have high access latencies, on the order of milliseconds, for both read and write operations. This is the single largest bottleneck for any consumer, enterprise, cloud, or HPC workload. Recently, solid-state drives (SSDs) based on NAND Flash have become popular as a replacement for HDDs. SSDs have read and write latencies on the order of microseconds. SSDs are laid out in a number of large *blocks* which are typically 16 KiB in size; each block is divided into 512 byte, 2 KiB, or 4 KiB *pages*. However, NAND flash has a peculiar property: while individual 1 bits may be cleared to 0 bits, the only way to change a 0 back to a 1 is to erase the entire block which resets it back to all 1 bits. Furthermore, NAND flash can only survive a low number of erase cycles before wearing out and becoming unreliable.

To deal with these problems, modern SSDs use a Flash Translation Layer (FTL) [31], analogous to a virtual memory page table. The FTL maps logical pages to physical pages on the storage medium, and this in turn is used to transparently replace a data modification with a copy-on-write operation. Wear-leveling algorithms are used by SSDs to spread the writes out across all pages on the storage medium such that the number of erase operations is approximately equal for each memory page. Due to these algorithms, there is no actual guarantee that consecutive logical blocks will be physically located together on the storage medium.

Database systems using the same methods as they did on HDDs will perform worse and will wear out the SSDs media faster. This is because changing a single bit within a page requires reading the previous contents of the page, locating a free (i.e., erased) page on another block, writing the modified block to the new location, updating the FTL to point to the new location, and marking the previous location as invalid. This previous location is unusable until the SSD performs *garbage collection*, during which valid pages are consolidated and blocks consisting only of invalid pages are

erased and thus made available again. This inefficiency causes many applications to perform worse than expected when ported from HDD-based storage to SSDs.

Thus, efficient use of SSDs requires the use of log-based data structures [56], which append new data rather than replacing existing data.

2.3.2 Storage Class Memory

Both HDDs and SSDs are block-oriented: the SCSI, ATA, and NVMe commands which access the disks operate on logical blocks of 512 or 4096 bytes in size, due to the physical properties of disks and NAND flash media. Newer byte-addressable storage technologies are emerging, including phase-change memory (PCM), spin-transfer-torque memory (STT-RAM), and resistive RAM (ReRAM). These technologies have read latencies measured in nanoseconds and write latencies only slightly slower than volatile memory. This makes it possible for these technologies to be the target of individual CPU load and store instructions. These byte-addressable persistent storage technologies are collectively referred to as Storage-class Memory (SCM). In turn, SCM and NAND Flash are collectively referred to as non-volatile memory (NVM).

As previously stated, HPC clusters have used distributed file systems (DFSs) to store data on a number of disks. The storage layouts for many distributed file systems are based on existing local file systems which assume that they are backed by a block device on spinning media [16, 13]. Contemporary file systems attempt to minimize expensive random access by placing file metadata close on disk to the actual data and rewriting data in place. Because SCM has generally uniform access latencies, excluding NUMA effects, this layout is no longer needed, and may even be harmful—NVM has much lower write endurance than spinning hard drives. As a result, modern wear leveling algorithms for NVM replace data rewrite operations with a fresh write onto a new disk block, which can cause significant overhead.

FlashNet [57] is a software stack which intends to unify all components of accessing local and remote NAND flash storage. FlashNet consists of a software Flash controller which maps the storage of a number of solid-state disks onto a single virtual address space, the ContigFS file system [57] which stores files contiguously in that virtual address space, and a software RDMA controller which

allows remote RDMA READ and RDMA WRITE operations on the Flash storage. The goals of FlashNet are to make I/O operations look to the application like any other memory access, keep overhead to a minimum, and present as simple an interface to the application as possible. This is accomplished by simplifying the file system and leveraging `mmap` as the main I/O mechanism.

Because SCM is meant to be part of the memory hierarchy, data destined for SCM may be cached at other volatile layers of the memory hierarchy, or the SCM medium may take time to actually record data that has been written to it. Applications must ensure that data destined for SCM is actually committed to the durable medium before the user is informed that the data is durable. For applications which allow access to remote SCM, this currently requires that clients send an explicit message to the application at the target to force a commit of the data to SCM. Tom Talpey submitted a proposal to the IETF to add an RDMA COMMIT operation to iWARP [58]. This operation will tell the remote NIC to flush specified regions of non-volatile memory to ensure that they are made durable, without waking up the user application or host CPU. This can significantly improve performance of storage applications.

2.3.3 Distributed File Systems

We now turn our attention to distributed file systems, but with a specific focus on storage class memory. The Hadoop Distributed File System (HDFS) [13] is a distributed file system intended for use with MapReduce and other applications which use the Hadoop stack. As such, HDFS does not implement standard POSIX semantics; rather, once an HDFS application closes a file that it has created, the file's contents becomes immutable. This works well with MapReduce applications which are implemented as simple filters which read a set of files and produce a single file as output. As a result, a number of concurrency issues can be eliminated.

HDFS borrows its fault tolerance model from the earlier Google File System [12]. In particular, HDFS keeps its metadata store in memory on a single node (with snapshots and a journal stored to disk), and defaults to creating three replicas for each file. Three replicas allows for a balance between fault tolerance and performance, as one replica can be stored on-the node producing the file, another replica on a different node on the same rack, and a third on a different rack. This

means that the filesystem can survive a single node failure as well as the failure of a single rack, the latter of which could be caused by a failure of a single switch or power distribution unit.

The crail framework [59] provides a multi-tier distributed file system based on the HDFS model. Unlike HDFS, fault tolerance is not a direct goal; rather, the focus is on providing a high performance temporary data store for the Spark Shuffle engine [60]. As such, crail provides a DRAM tier for these types of applications, while providing NVMe over Fabrics and block storage tiers for applications which require durable data storage. Additionally, crail does not attempt to preserve file locality by default; crail makes the assumption that the performance of remote Flash access is similar to that of local Flash access. However, applications may indicate a preference as to which nodes are used for data storage. Finally, while both HDFS and crail provide the synchronous HDFS API which aligns with TCP semantics, crail additionally provides an asynchronous API which exposes the underlying RDMA semantics.

2.3.4 Architectural Support

The literature consists of designs that rely only on existing hardware primitives as well as designs which require processor or system bus support not yet available in commodity hardware. The most common requirement is epoch barriers [36, 61]. An epoch barrier allows writes to be grouped into an epoch which are ordered with respect to other epochs, without affecting ordering of writes within the epoch. This is different from existing mechanisms such as mfence [62], which affect memory visibility by other CPU cores but do not necessarily enforce write ordering to main memory.

Ouyang, et. al., [63] implement a new Flash storage primitive called atomic-write by making a small modification to the flash translation layer (FTL) of a storage device. The FTL is typically implemented as a log-based file system, which allows entries to be atomically appended. Atomic-write adds a single bit flag to each entry indicating whether or not this block was the final block of a transaction; this bit is always 1 for normal single-block writes. This allows atomic writes of multiple blocks at the FTL level by setting the intermediate log entries to 0. On crash recovery, if the final log entry's flag is not 1, all entries after the final 1 are discarded since they were part of an incomplete transaction. Ouyang, et. al. demonstrate this scheme by replacing the existing

double-write scheme used by the MySQL InnoDB engine's transaction log with a simple atomic-write transaction. In this case, the number of writes are effectively halved, taking advantage of the atomic nature of the FTL and avoiding unnecessary wear on the Flash storage device due to duplicate writes.

2.3.5 Programming models

Multiple authors have proposed programming models for non-volatile memory [64, 61, 65]. These all rely on some form of transactional memory support, in which changes to data structures are isolated and applied atomically as a single unit, to avoid conflicts with other transactions. In general, existing transactional memory implementations provide atomicity, consistency, and isolation, three of the four ACID requirements for relational database applications [66]. Most NVM programming models seek to add durability to existing transactional memory semantics. These projects vary in the level and degree of safety.

The Mnemosyne project has three goals: to make it easy to create persistent data structures, to provide consistent updates via a transaction system, and to work with commodity processors. To accomplish this, Mnemosyne implements low-level constructs, including store operations which optionally bypass cache, and operations to map and unmap persistent memory regions into virtual memory, and transaction support using a transaction log. The transaction log used by Mnemosyne uses a circular buffer with a torn-bit, which is flipped on each pass through the buffer and used to detect an incomplete write to the transaction log. The torn-bit is inexpensive for small transactions but becomes expensive for transactions larger than 2 kibibytes, because the 64-bit words in the transaction must be divided among 63-bit buckets. Mnemosyne targets C and provides minimal memory safety checking using annotations understood by the Sparse semantic checker [67] to ensure that pointers into volatile memory are not stored in non-volatile memory.

The NV-heaps project has similar goals, but targets object-oriented C++ programs. While both systems provide ACID transactions, they differ in how they provide them. The goals of NV-heaps are to provide pointer safety, ACID transactions, a familiar API, high performance, and scalability. Memory safety is a primary goal of NV-heaps, and thus the framework leverages C++'s type system

to prevent pointers from volatile memory being stored in non-volatile memory, and additionally to prevent pointers from one non-volatile memory region from being stored in another. However, this extra checking incurs massive overhead, measured by the authors as an $11\times$ performance loss compared to a version of the system with no memory safety. The authors justify this by the complexity of manually verifying memory safety and the large cost of a corrupt data structure in persistent memory caused by an invalid pointer. NV-heaps also rely on processor support for epoch barriers to fence transactions, which are not available on commodity processors.

Transactional memory requires one of two models, undo or redo logging [68]. The first method uses an undo log. In this method, as a value is updated in persistent memory, the previous value must first be committed to the undo log. Each updated value must be committed in order in case the transaction is aborted, in which case the previous values must be restored in reverse order from the undo log. Then, the new value is written onto the persistent storage but not committed. When the application commits the transaction, the transaction memory implementation commits the pending new values to the persistent medium. If there is a failure at any time during this transaction process, the previous state may be restored by writing the values stored in the undo log. Once the new data has been completely committed, the application is notified that the write has completed.

A redo log contains a single compact log of all updates within each transaction, instead of storing previous values and updating the live data. As each value is updated, it is placed into the redo log instead of into the persistent memory region. Unlike an undo log, each individual update to the redo log need not be committed to the backend storage in order, because aborting the transaction merely requires throwing away the redo log. When the application commits the transaction, the existing data is overwritten asynchronously. This is because in the event of a failure, the redo log contains enough information to redo the transaction and re-start the write of the new data. However, throughout a transaction, any reads to the data must be redirected to the redo log, which requires redirection at the paging level and removes the direct access benefit of storage class memory.

Wan, et al., observe that redo logging performs better when a transaction affects many objects,

while undo logging performs better in read-intensive workloads [68]. Liu, et al., observe that redo logging performs better when memory redirection occurs at page granularity since the page tables can be smaller and leverage existing hardware to perform the mapping [65]. These authors produced DudeTM (DUrable DEcoupled Transactional Memory), an attempt to leverage this observation to build a durable transaction system which leverages existing volatile transactional memory systems. It decouples the transactional memory from the persistence problem by performing the transaction in memory and then persisting only the redo log before returning to the application. DudeTM then flushes the transaction asynchronously, as the redo log is all that is needed to completely persist the memory transaction.

FaRM

Fast Remote Memory (FaRM) [69, 70] is a distributed transactional object store. Objects are stored in memory regions; these regions are addressed via a distributed hash table. The distributed hash table consists of k rings each with its own unique hash function; each machine is inserted into each ring, using its IP address as input to each ring's hash function. Objects are stored in one of a number of memory regions; each region identifier encodes the ring number and position within the ring.

FaRM's programming model is transaction-based and uses continuation callback functions in a manner similar to Scheme or JavaScript. Transactions can create, read, write, and free objects. Writing an object requires creating a local copy and modifying the copy. Changes only take effect when the transaction is committed. The commit process is a two-phase commit. Locks are acquired during the *prepare* phase. During the subsequent *validation* phase, the versions of each object read during the transaction are checked against the stored version; if any version is out of date, the transaction is aborted. If validation passes, the new object data is sent to the memory servers and the new object versions are committed.

For routines which only need to read objects, a lock-free programming model can be used instead, which does not require the locking overhead of a transaction. However, lock-free read operations must be bookended by a start and stop function; any object data read becomes invalid

when the stop function is called. This bookending is used to determine the lifetime of old versions of objects in the system.

FaRM uses RDMA WRITE and RDMA READ operations for remote object access. The threads on a single system each contain a queue pair connected to a remote machine; this keeps queue pair utilization lower than connecting each local thread to every remote thread. FaRM also takes advantage of the behavior of specific InfiniBand HCAs to improve performance. For an application to be portable across all RDMA hardware and system bus architectures, the application should use immediate data or a subsequent SEND to trigger a completion for RDMA WRITE operations. FaRM instead takes advantage of two behaviors of Mellanox HCAs: the HCAs always place data in memory in increasing byte order, and they always place entire machine words at once. Thus, FaRM uses a circular buffer which is initially zeroed. Each "message" in the buffer starts with a length. The receiver polls the length until an RDMA WRITE operation changes the length to non-zero. Then, using the length value, the receiver polls the last word of the message trailer until the RDMA WRITE operation changes it to non-zero. This indicates the end of the message because the message format used by FaRM guarantees that the last word of the message trailer will never consist of all zeros. Once the receiver has processed the message, the receiver sets the memory used by the message to zero again so that it may be used for a subsequent request. This avoids the need to poll for completions, at the cost of increased CPU utilization and portability.

Each object has a header whose contents stay valid even when the object is freed; an *incarnation* number stored in the header is used to tell whether a different object has taken the place of an object formerly stored in that location. If object sizes must change to satisfy an allocation request, a barrier is set up for all active transactions, and only after all active transactions have ended can existing object headers be destroyed to allocate the new object. Each object is addressed via a 128-bit fat pointer consisting of the object address, size, and expected incarnation. This allows a memory server to ensure that the object contains all metadata necessary to verify that the stored object matches the object referenced by the fat pointer.

To demonstrate the capabilities of FaRM, the developers created a distributed hashtable on top of it, distinct from the distributed hashtable used internally for addressing memory regions. The

hashtable was designed to allow most lookups to be performed via a single RDMA READ, which is accomplished using an algorithm called *chained associative hopscotch hashing*. Hopscotch hashing has an invariant that each value must be stored within H buckets of the location given by the hash function. The chained associative variation includes an overflow chain per bucket, so that H can be smaller and thus RDMA READ requests corresponding to a lookup can be smaller. Buckets are kept as distinct FaRM objects stored contiguously with $H/2$ key/value pairs per bucket, so a lookup performs an RDMA READ of the buckets k and $k+1$ for a key with hash k; if the object is not found, the overflow chain is checked using lock-free reads.

The system requires *precise membership*: that is, all nodes in the cluster keep a list of all other nodes that are alive in the cluster and do not accept any requests from nodes outside this set. A central configuration manager (CM) verifies nodes are online using a heartbeat of a small number of milliseconds; this is possible because heartbeats are sent and received via a dedicated background thread bound to a dedicated CPU. On a heartbeat failure, the CM issues an RDMA READ probe to all nodes in the cluster; any that do not respond are removed from the cluster configuration. Memory regions are remapped from failed nodes to their replicas, and then the new configuration is propagated to all of the nodes. If the CM itself fails, nodes request reconfiguration from a backup CM. A CM only completes reconfiguration if a majority of nodes in the cluster respond to the RDMA READ probe; this ensures only a single CM succeeds in the event of a cluster partition due to a failed switch.

NAM-DB

NAM-DB (Network-attached-memory database) [37] focuses on scalable distributed transactions using snapshot isolation, which allows transactions to operate on previously committed data from the latest valid snapshot of each object at the time the transaction was started. To make global timestamps scalable to hundreds of nodes and multiple threads per node, NAM-DB uses a *timestamp vector* inspired by *vector clocks* [71]. However, unlike vector clocks, each record only includes the global thread identifier and timestamp of the latest commit, as opposed to storing the entire timestamp vector, greatly reducing the storage cost of timestamp vectors. The global *read times-*

tamp vector is copied at the start of each transaction. For a given transaction, a committed object is valid if the commit timestamp (i, t) is less than or equal to the corresponding field t_i in the read timestamp vector. The remaining scalability issue is that for thousands of nodes (or even higher orders of magnitude) the read timestamps become unreasonably huge and it places a large load onto the memory node which stores the latest version of the timestamp vector. The authors suggest compressing the timestamp by only including the most recent thread for each compute node, or partitioning the timestamp vector across multiple memory nodes.

Chapter 3

USERSPACE SOFTWARE RDMA

This chapter summarizes my work done on urdma [49], a software implementation of RDMA verbs which performs data transfers in userspace.

3.1 Introduction

As previously discussed, RDMA allows user applications to access remote virtual memory without kernel intervention. However, it requires expensive, specialized hardware on each end node. For high-performance computing environments, this expense is justified due to low latency requirements and the need for as many CPU cycles as possible to be dedicated to computation instead of network usage. However, RDMA as an abstraction can be useful outside of HPC environments. Existing software implementations of RDMA include softiwarp [29] and softroce [30]. Software implementations have two major uses: (i) research and experimentation for new RDMA features, (ii) as an inexpensive client endpoint for an RDMA server that uses real hardware. The latter use case requires the implementation to interoperate with existing implementations, but the former use case does not.

Existing software implementations are implemented in the kernel, matching the design of existing RDMA drivers. Implementing RDMA data transfer in the kernel, as opposed to userspace, has several advantages. Most importantly, the verbs stack is designed assuming that all resources will be allocated and connection management performed in the kernel. This means that the path of least resistance is a kernel implementation. Also, RDMA READ and RDMA WRITE operations can be implemented without involving the userspace process. Additionally, with kernel sockets zero-copy TCP data transfers are possible because the code has direct control over the socket buffer data structures.

However, there are many reasons why a software RDMA verbs emulation in userspace would be convenient. First, when a transfer operation is posted to an empty work queue, the userspace process must use a doorbell call to inform the kernel so that the kernel will begin processing the queue. For applications which send infrequent small messages, this can be very expensive. Additionally, kernel code, although easier to manage than hardware, is much harder to write and debug than userspace code. Finally, direct userspace access to other devices, such as GPUs and NVM devices, is becoming increasingly common. When writing a storage application that uses such a userspace interface for accessing a local NVM device, entering kernel space in order to perform network transfers of that data defeats the purpose of having userspace access to the storage device. Thus, we would like an efficient way to implement RDMA verbs in userspace. In doing so, we could intelligently reuse the same memory buffers that were used by the storage API with the RDMA verbs API.

The Data Plane Development Kit (DPDK)[1] provides an API for userspace applications to directly read and write Ethernet frames on commodity NICs. DPDK is targeted at applications that perform bulk packet transfer or forwarding, such as software routers, firewalls, and packet generators. The direct level of access from userspace that DPDK provides allows us an opportunity to rethink software RDMA.

While DPDK does not provide a TCP/IP stack, the ability to send and receive Ethernet frames is enough to implement a software RDMA implementation that performs data transfer in userspace. This chapter introduces *urdma*, which is a software RDMA emulation using DPDK that can run unmodified verbs applications. While urdma depends on hardware filter capability, it does not have a direct dependency on any specific NIC, putting it into a different class of userspace RDMA solution from hardware/software solutions such as Cisco's usNIC [72].

[1] https://dpdk.org

3.2 Implementation

DPDK gives a single user application complete control of NICs on the system. However, this means that there is no form of synchronization between multiple applications trying to use the same NIC. However, DPDK does support multiple processes within the same application sharing the NIC, by sending and receiving on independent queues on the NIC. This means that to support multiple verbs applications we must configure DPDK to treat them as processes within a single DPDK application.

To support this, urdma consists of three components, as illustrated in Fig. 3-1: (i) a kernel module (*urdma_kmod*) which provides RDMA connection management (CM) support, (ii) a userspace daemon (*urdmad*) which initializes DPDK and arbitrates which NIC queues each verbs application has access to, and (iii) the RDMA provider library (*urdma_prov*) which implements the verbs API and performs the data transfers. These depend on other system components, and these dependencies are shown in the figure. In particular, the use of RDMA verbs requires the verbs library and the connection management library, both of which have userspace (libibverbs, librdmacm) and kernel (ib_uverbs, rdma_cm) components. DPDK contains a kernel component called KNI, and relies on the VFIO kernel module which is part of the upstream Linux kernel. Finally, urdma relies on the *libnl* library in order to manipulate network interfaces on the system, including setting the MAC address of the interface and adding IP addresses.

3.2.1 Kernel module

As previously mentioned, the Linux RDMA stack, based on the OFA verbs API, requires a kernel driver to perform device and resource initialization as well as connection management. This is done for security reasons, since multiple independent applications are accessing the shared NIC. However, this is not ideal for an RDMA driver based on DPDK, because DPDK gives userspace complete control over the hardware, removing control from the normal NIC driver that resides in the kernel. Thus, urdma provides a "stub" kernel verbs driver, *urdma_kmod* in Fig. 3-1, that does the minimum to satisfy the kernel verbs API. This means that for each urdma protection domain, completion queue, and queue pair allocated by a userspace application, there is a corresponding

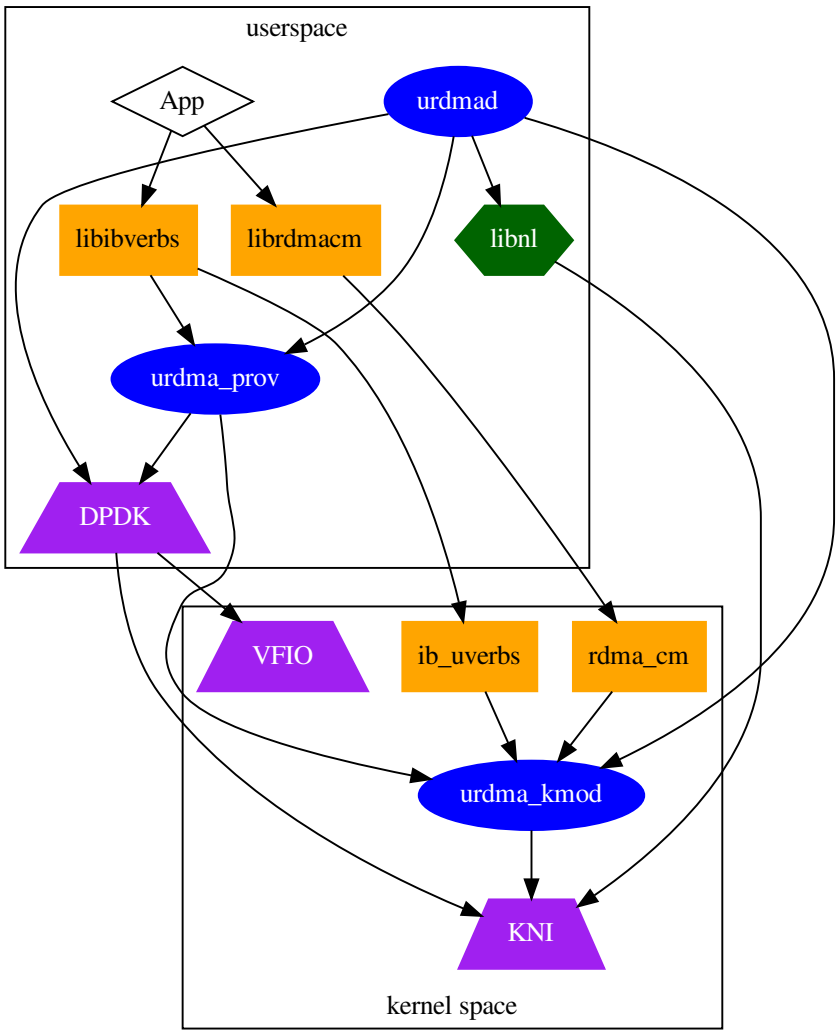

Figure 3-1: The components of urdma and their relationships. The blue components are the three parts of urdma: urdmad, the user daemon; urdma_kmod, the kernel module; and urdma_prov, the userspace verbs provider library. The purple components are part of DPDK. The orange components are part of the Linux RDMA verbs stack. The green components are other external libraries.

stub object in the kernel.

The most interesting part of urdma_kmod is connection management, because the RDMA connection management state machine is driven from the kernel. The urdma kernel module leverages a DPDK feature called kernel network interface (KNI) to perform the connection establishment in the kernel before handing control over to userspace. This happens just before the kernel sends the connection accept message, and on the client side just after the kernel receives the connection accept message. This ensures that userspace can set up hardware filters to direct the packets corresponding to that connection to a queue owned by the process before the first data packet arrives on the connection. In order to accomplish this, urdma_kmod provides a character device which is used to tell userspace when to enable the hardware filter for the queue pairs. This relies on the iWARP assumption that the client will send the first message.

3.2.2 User daemon

The urdma user daemon, *urdmad*, is responsible for initializing DPDK and arbitrating which queues on the NIC each verbs application has access to. When urdmad starts, it sets up KNI to create a mirror of each DPDK NIC in the kernel, so that urdma_kmod can send connection management packets. Additionally, urdmad creates a UNIX domain socket to which the verbs provider library connects in order to request access to NIC queues for each queue pair that the application creates. urdmad consists of a single thread event loop which forwards packets between the actual NIC under DPDK control and the virtual NIC under kernel control in addition to monitoring the character device signaling incoming connections and the UNIX domain socket used to communicate with the verbs provider, urdma_prov.

3.2.3 Provider library

We next discuss the implementation of our provider library for libibverbs (*urdma_prov* in Fig. 3-1), the userspace portion of the Linux RDMA stack which applications use to access RDMA queue pairs.

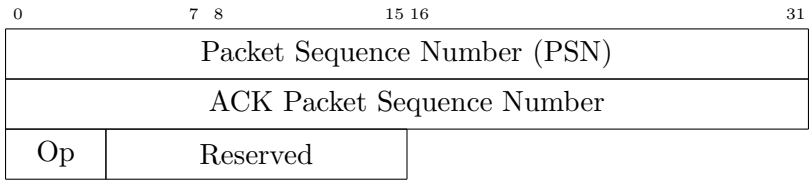

Figure 3-2: TRP protocol header.

Trivial Reliability Protocol (TRP)

In the urdma provider library, we implement the upper two layers of the iWARP protocol (DDP and RDMAP). However, for implementation simplicity, urdma uses UDP instead of TCP, with a thin shim protocol to provide reliability. While this sacrifices interoperability with existing implementations, it simplified the implementation: urdma does not need to implement the full TCP state machine and can implement simplified connection setup and teardown. We can assume more limited failure cases because the urdma nodes are likely to be on the same subnet. Additionally, a TCP implementation must cope with the byte stream nature of TCP; even if an endpoint never sends DDP segments that cross a TCP segment boundary, it must be prepared to reconstruct DDP segments from an arbitrary TCP stream. Using a UDP-based implementation sidesteps this concern. The final reason for using UDP is to keep urdma in control of all connection-related state, so that we can transition the connection state from kernel space to userspace, which would be difficult using TCP. While the iWARP specification allows for implementation over SCTP [48], a reliable message-based protocol defined by the IETF, no known iWARP implementations support SCTP.

The reliability shim protocol that urdma uses is referred to as the Trivial Reliability Protocol (TRP), and is implemented in both urdma_kmod and urdma_prov. The protocol format is shown in Fig. 3-2. TRP has opcodes for Connection Request, Connection Response, Connection Shutdown, Data, and Selective Acknowledgement. Selective acknowledgements are supported but are separate messages instead of being in an option header as in TCP. This was done for implementation simplicity. The header ends at 10 bytes as opposed to being a multiple of 4 bytes because the DDP header is intended to begin at a 2 byte offset due to the MPA length header. Unlike

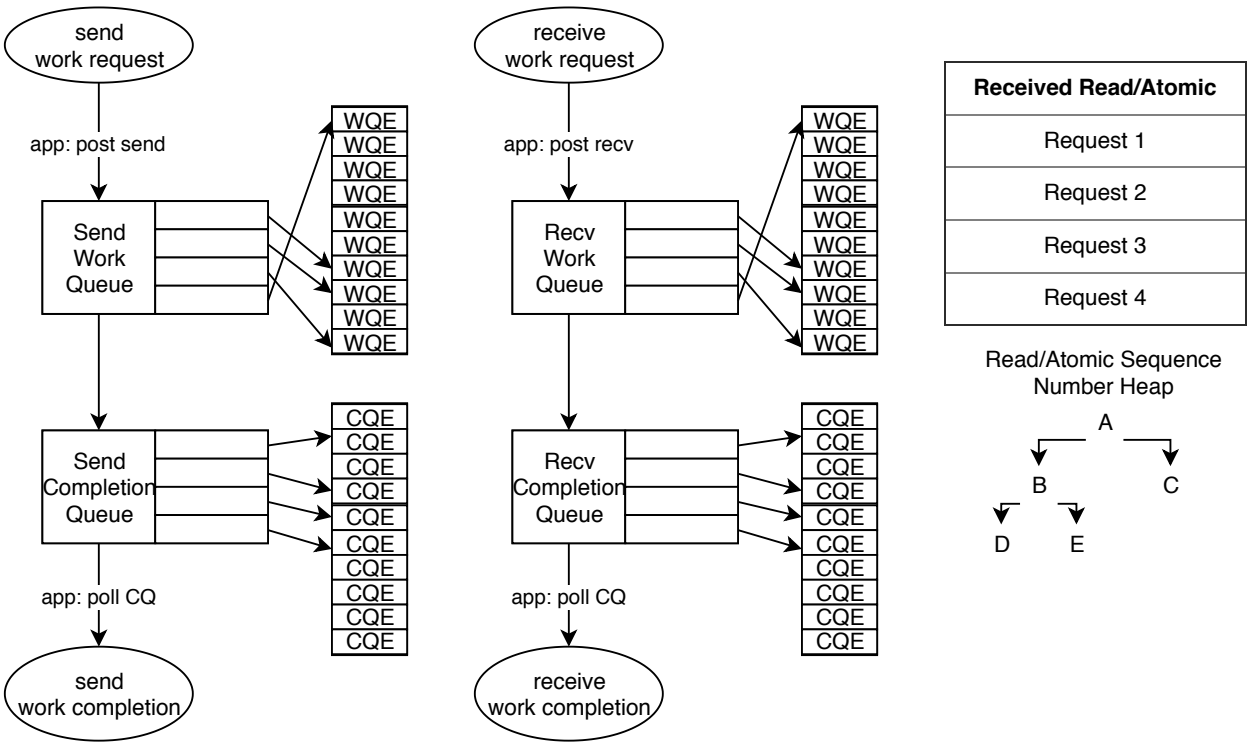

Figure 3-3: Data structures used by urdma_prov, the verbs provider library for urdma.

MPA, TRP does not need a length field because it can be trivially derived from the UDP length, and a TRP datagram will always contain exactly one undivided DDP segment.

Data Transfer

We now discuss the data transfer portion of the urdma verbs provider (urdma_prov). The urdma verbs provider (urdma_prov) performs all data transfer in a background progress thread. This design decision was made for two reasons. First, at the time that urdma was designed, DPDK API calls needed for data transfers were only able to be called from threads created by DPDK, referred to as logical cores or lcores. To run unmodified verbs applications, we allow the application to control its own threads and the affinity of its own threads. Second, we want RDMA READ and RDMA WRITE operations to occur asynchronously with respect to the application, without forcing the application to make verbs API calls to allow the data transfers associated with these operations to make progress. The data structures used by our verbs provider library are shown in Fig. 3-3.

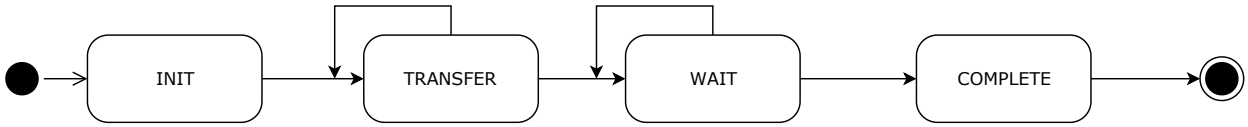

Figure 3-4: State diagram for send work requests in urdma.

The queues are implemented as pointers to a fixed-size array of work queue and completion queue elements. There is also a separate table of received RDMA READ and Atomic requests, as these do not correspond to local application work requests.

Our verbs provider uses a 4-state machine for send work queue entries (WQEs): INIT, TRANSFER, WAIT, and COMPLETE, as illustrated in Fig. 3-4. This state machine has two goals: (1) ensuring that operations cannot overlap, and (2) ensuring that the application will not receive a completion until the data buffer is ready to be reused. When the user issues uses the POST SEND verb to issue a send work request, an RDMA verbs driver places a work queue entry (WQE) onto the send queue. WQEs transition from the INIT state to the TRANSFER state when the background progress thread pulls them off the send queue. Once all segments for the operation have finished sending, the WQE transitions to the WAIT state to wait for the last segment of the request message to be acknowledged, and all segments of the response message to be received, if applicable. Once the data transfer associated with the WQE has finished, the WQE transitions to the COMPLETE state. In the COMPLETE state, urdma_prov must wait for all prior operations on the send queue to complete before it may post a completion queue entry (CQE) to the completion queue for the WQE, informing the application that the operation has finished.

The iWARP message header contains two bit fields: Tagged and Last. The last bit is self-explanatory: it is set on the last segment of an iWARP message and clear otherwise. However, the tagged bit requires more explanation. The iWARP protocol stack defines two types of transfer operations: tagged and untagged. Tagged messages are used for RDMA READ and RDMA WRITE messages, which place data into a specific virtual memory location without requiring involvement from the application at the target endpoint. Tagged messages in iWARP are identified by a steering tag (STag) and a tagged offset which determine the target memory region. Untagged messages are

used for messages that require processing by the target endpoint beyond simply placing the data. The iWARP protocol defines queues for each type of untagged message, which are independent of the queue pair concept from the verbs API. These are intended to be low-level hardware queues intended to process a single type of packet, in order to provide separation between the DDP and RDMAP protocols. Using this concept, per the iWARP RDMAP specification [26], untagged messages are identified by a queue number and a message sequence number. However urdma does not use this queue number concept at all, but rather uses the opcode field to decide how to process the packet, since the opcode field is global and not specific to a given queue number.

As mentioned, iWARP does not define any kind of sequence number for tagged messages. This complicates the implementation, because an application can issue multiple independent overlapping RDMA requests targeting the same STag at the same time. We only have the last bit in the header as a way of separating one message from another in the stream of segments Ideally, we want to place the data in an RDMA data segment immediately, even if the segment was received out of order. However, if a packet is received out of order, we do not necessarily know if any of the intermediate packets had the last bit set, and thus we do not know whether or not this belongs to the same message as the previous segment or another message. To determine the actual packet ordering, urdma uses the sequence numbers delivered by TRP. Once the last segment of the message is received and confirmed in order (no holes in the TRP sequence numbers), the message is considered "delivered". In the case of RDMA WRITE operations, nothing special needs to happen at the responder, since the requester will receive the TRP acknowledgement of the last segment and issue the appropriate completion.

For RDMA READ operations, on the other hand, the data flows in the opposite direction, from the responder to the requester. In this case, the requester must wait to issue the completion until the last RDMA READ response segment is delivered. To accomplish this, urdma_prov keeps the last segment number of every RDMA READ message in a binary heap, as shown in Fig. 3-3. Every time a segment is received, if there is a entry in the binary heap that is less than the received segment number and there are no holes in the sequence numbers, the minimum entry is popped and the earliest RDMA READ request is considered complete and put into the completion queue.

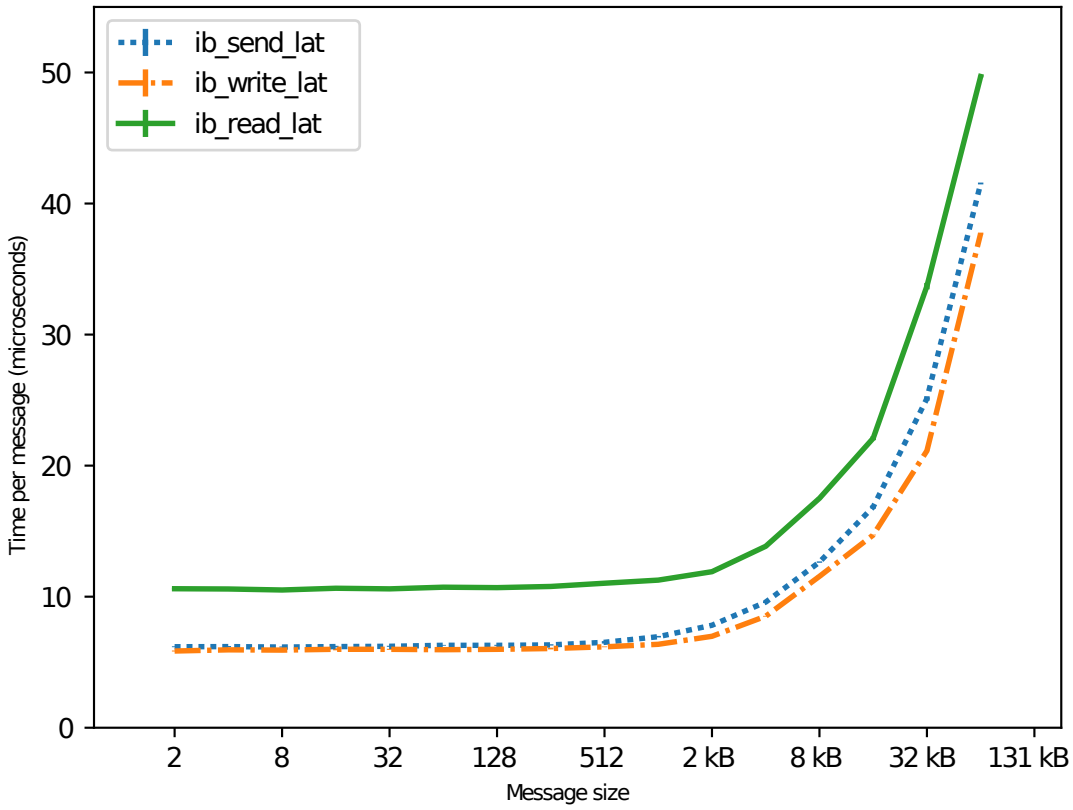

Figure 3-5: Latency vs. message size for RDMA perftest microbenchmarks for urdma. Latency is measured as one-half of the round trip time for SEND and RDMA WRITE and the full round trip time for RDMA READ.

3.3 Performance Evaluation

The performance results in this section reflect the performance of urdma as of June 2017 [49].

We compare performance for urdma, softiwarp, and a reference iWARP HCA. The performance tests run on pairs of identical systems. Our urdma and softiwarp tests ran on a pair of Supermicro SYS-6028R-T systems. Each system has 2 Intel Xeon E5-2630 v4 CPUs, 64 GB of DDR4 RAM, and a PCIe generation 3 bus. We use Intel XL710 40GbE NICs for our software RDMA devices. The NICs are running firmware version 5.05.

The reference iWARP HCA that we used for these tests is a Chelsio T580-LP-CR Unified

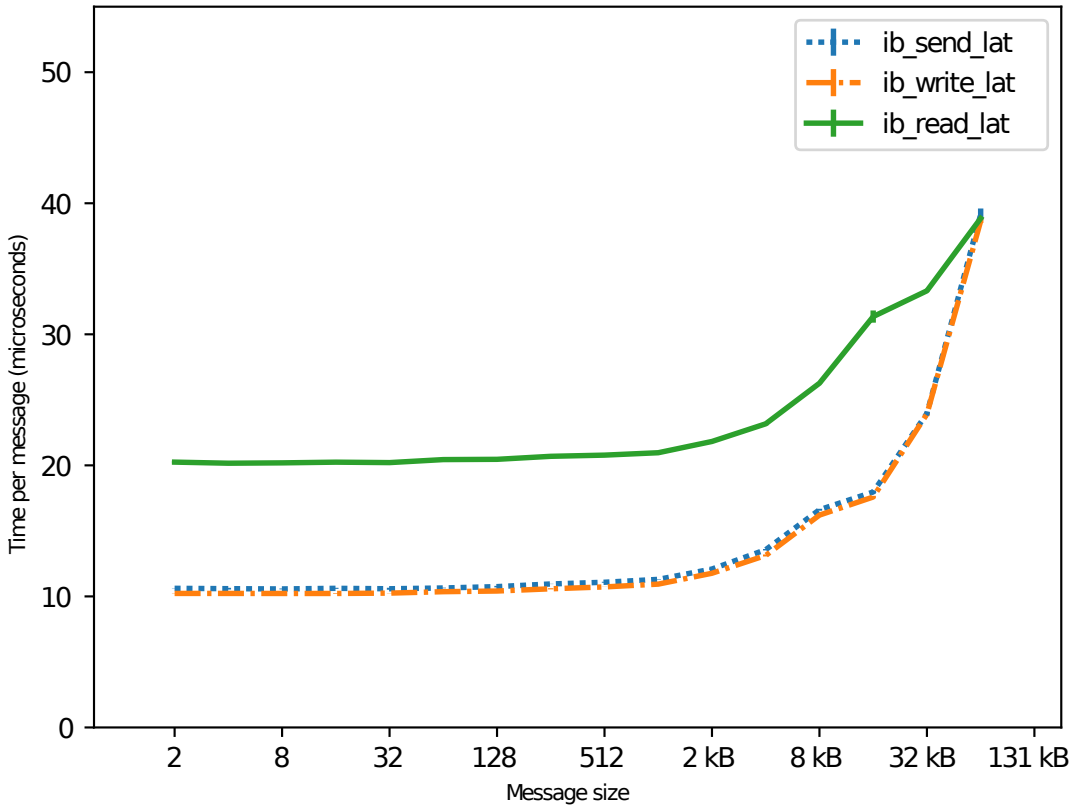

Figure 3-6: Latency vs. message size for RDMA perftest microbenchmarks for softiwarp. Latency is measured as one-half of the round trip time for SEND and RDMA WRITE and the full round trip time for RDMA READ.

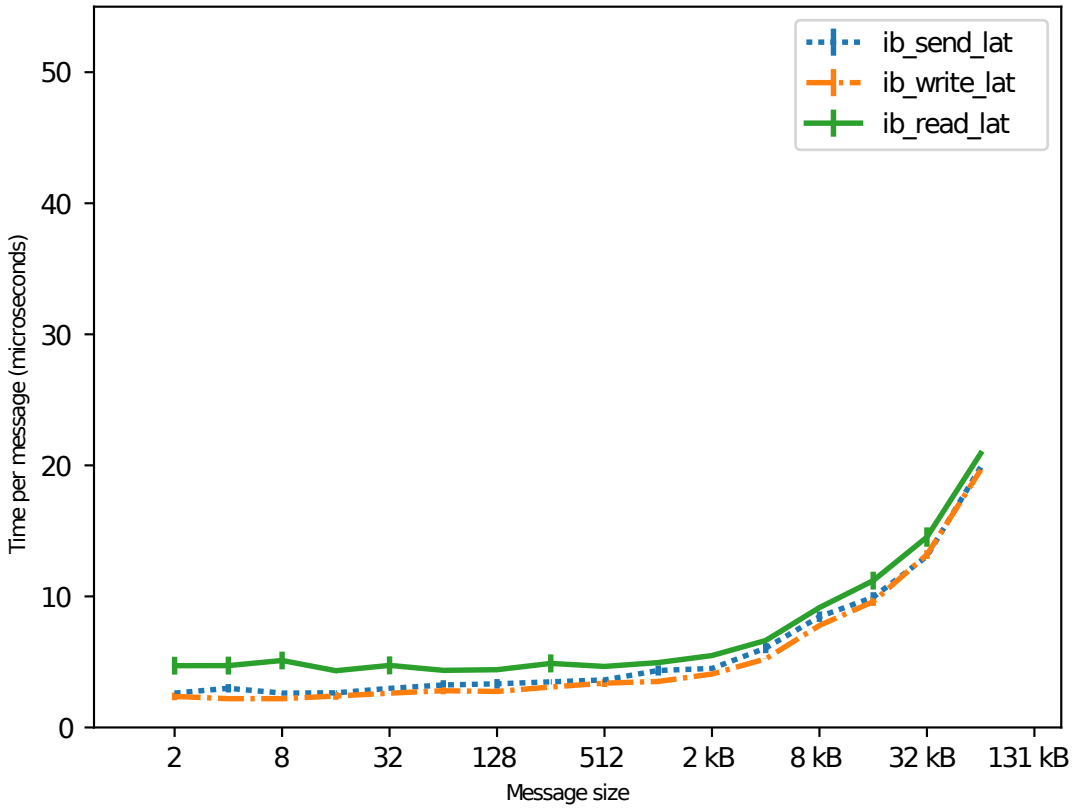

Figure 3-7: Latency vs. message size for RDMA perftest microbenchmarks for the reference HCA. Latency is measured as one-half of the round trip time for SEND and RDMA WRITE and the full round trip time for RDMA READ.

Wire Ethernet iWARP controller, with firmware version 0.271.9472 and userspace verbs driver version 1.4.0. The Chelsio HCAs are in a second pair of Supermicro servers, with 2 Intel Xeon E5-2609 CPUs, 64 GB of DDR3 RAM, and a PCIe generation 3 bus. Although the specification for these systems differ, we have put the reference HCA into the lower-spec system, as we expect that the reference HCA will offload all of the network data transfer calls and the lower specs will have minimal impact on the results.

All systems in this performance evaluation used Ubuntu 16.10 with the Linux 4.8.0-46-generic kernel as provided with the distribution, DPDK 16.07.2, and the provided libibverbs and librdmacm.

We first examine the latency of our three devices, measured as one-half of the round trip time for SEND and RDMA WRITE and the full round trip time for RDMA READ[2]. The results are shown in Fig. 3-5 for urdma, Fig. 3-6 for softiwarp, and Fig. 3-7 for the reference HCA. For reference, the means and standard deviations used to generate these graphs are shown in the tables in Appendix A. Overall, the reference HCA has the lowest latency, as expected, because the data transfer is completely offloaded with no dependency on the kernel or thread scheduler. For messages smaller than 16 KiB[3], urdma has a latency approximately 4 microseconds faster than softiwarp. This is the cost of the kernel context switch required by softiwarp when a send work request is added to an empty send queue, because the latency test sends a single message at a time and waits for the response.

We next show the results for the throughput microbenchmarks in Fig. 3-8, 3-9, and 3-10, for urdma, softiwarp, and the reference HCA respectively. The means and standard deviations used to generate these graphs are shown in the tables in Appendix A. The reference HCA provides very consistent throughput within 4 Gbps of the theoretical limit of the hardware for all message sizes 2048 bytes and above. For urdma, the RDMA WRITE throughput achieves at least 34 Gbps for message sizes between 32768 bytes and 2 Mebibytes[4], while the SEND throughput only achieves

[2]Recall that RDMA READ is fundamentally a round trip operation, because the data travels in the opposite direction of the request.

[3]1 KiB (kibibyte) = 1024 bytes

[4]1 Mebibyte = 1048576 bytes

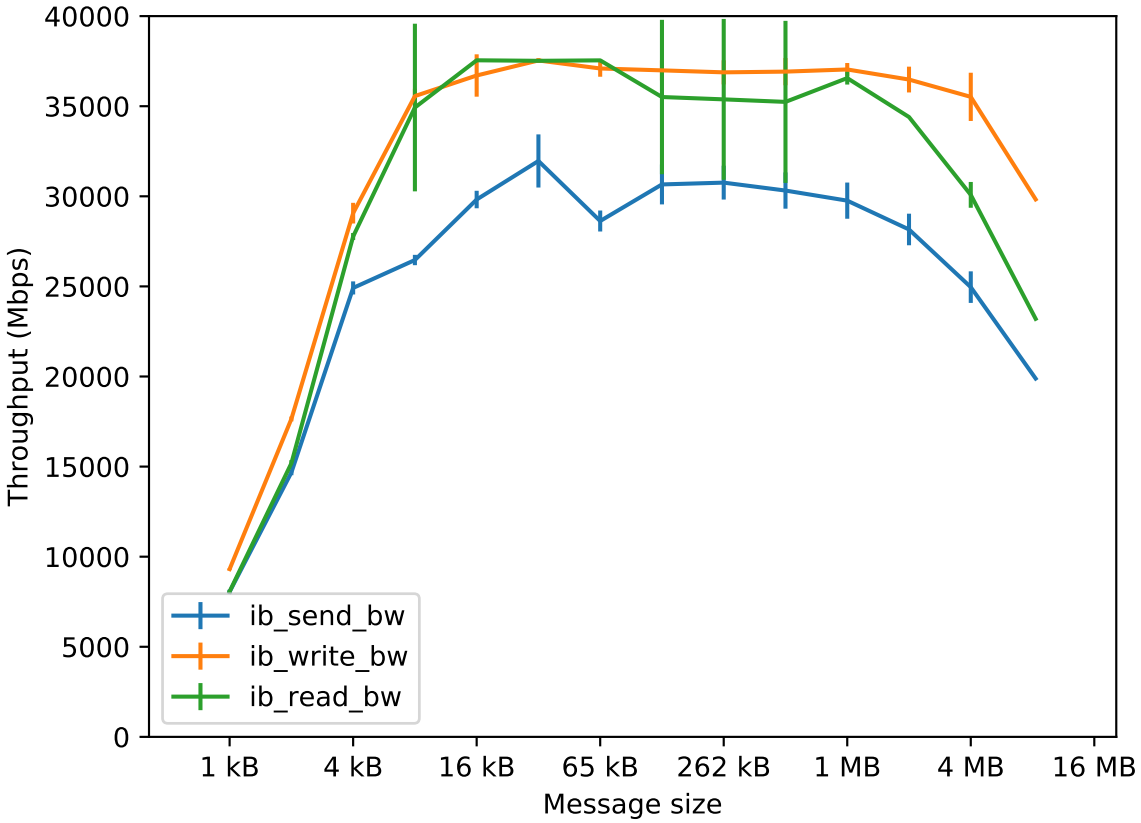

Figure 3-8: Throughput vs. message size for RDMA perftest microbenchmarks for urdma.

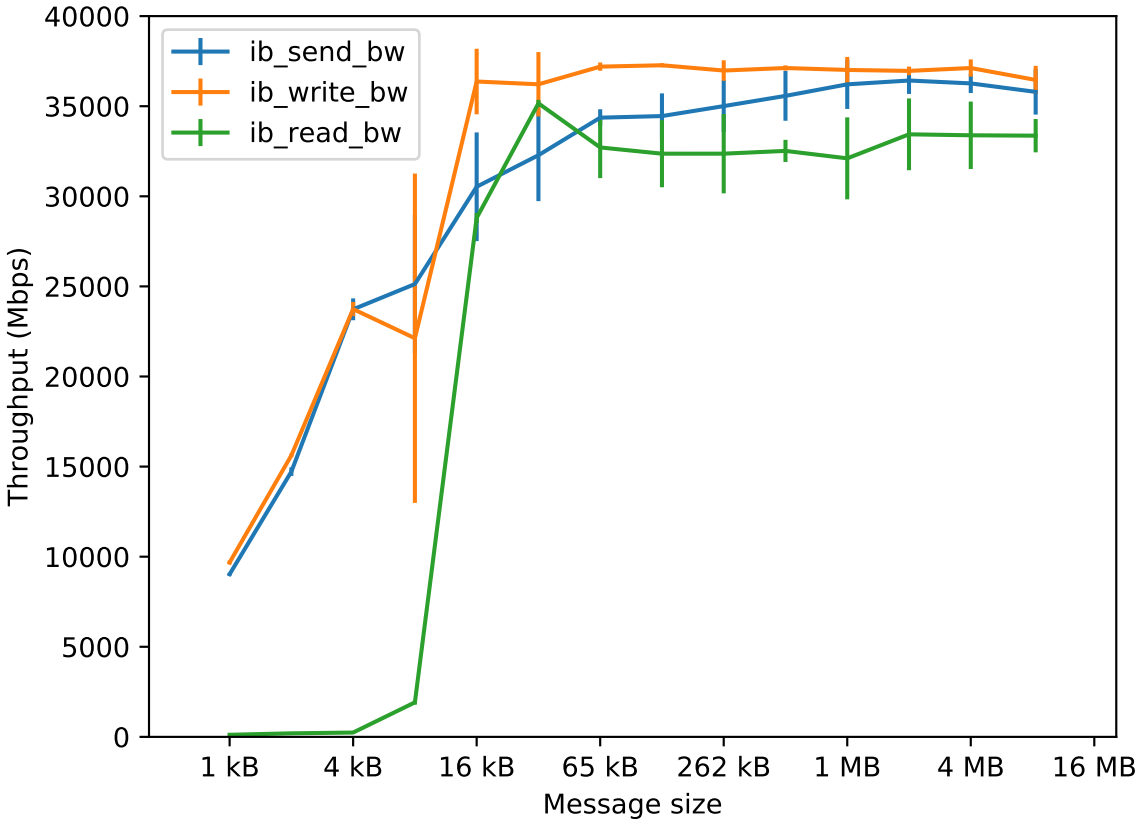

Figure 3-9: Throughput vs. message size for RDMA perftest microbenchmarks for softiwarp.

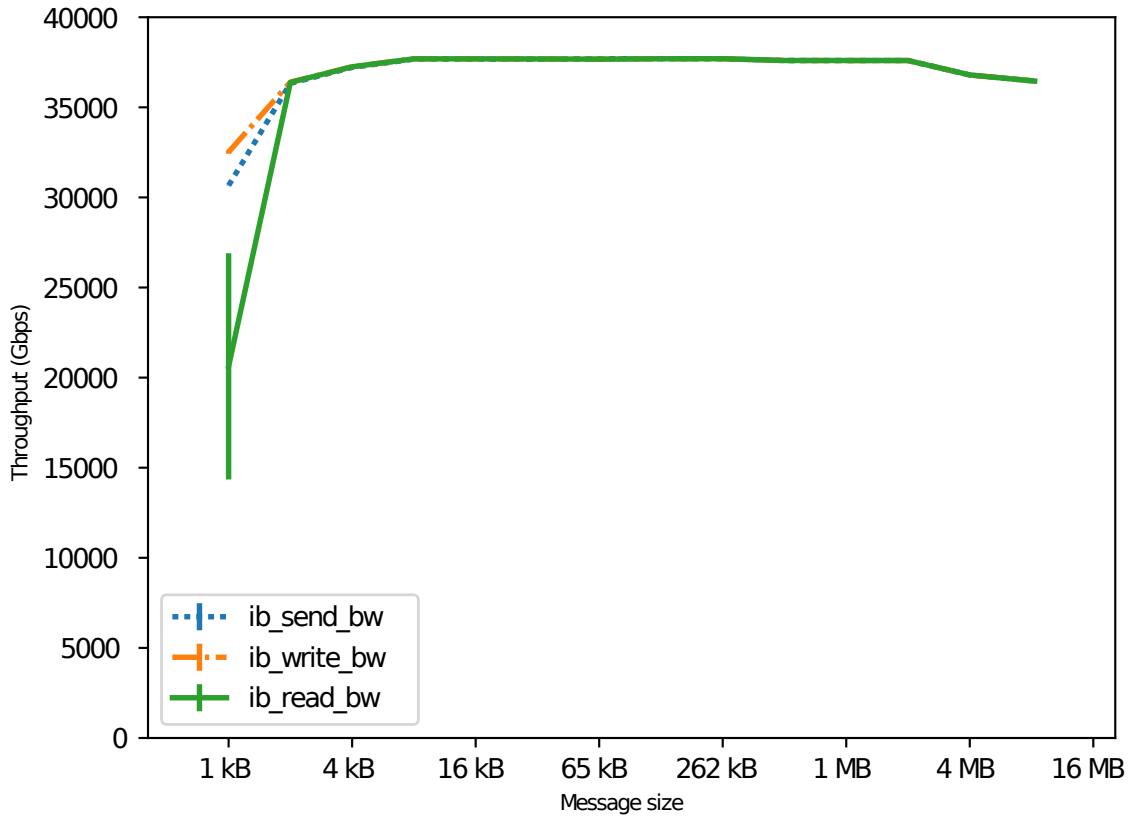

Figure 3-10: Throughput vs. message size for RDMA perftest microbenchmarks for the reference HCA.

a maximum of 27 Gpbs. This is because SEND requires the application at the remote endpoint to dequeue entries from the receive completion queue to free up credits for the sender to resend, and urdma additionally does not begin sending the next SEND message until the last segment of the previous SEND message has been acknowledged. We expect that most applications will use RDMA READ and RDMA WRITE for bulk data transfer, so the throughput of the SEND operation is not critical. The softiwarp implementation achieves greater than 30 Gbps for all message sizes greater than 16384 bytes. This shows that the kernel context switch overhead has much less impact on throughput than it does on latency. This is because the cost of the kernel context switches are amortized by the number of messages being transferred and the large size of those messages.

An additional concern for urdma is that throughput decreases for message sizes greater than 1 MiB. This is because we are using the minimum size for the NIC descriptor queues. When using the maximum size for the NIC descriptor queues, the throughput suffers massively, which is likely due to the number of descriptors overflowing the cache and causing many cache misses. Future work will identify optimal descriptor queue sizes for small and large messages.

We next show the throughput of crail [59] over urdma, using the iobench benchmark and using the DRAM tier to simulate storage class memory, in order to demonstrate the potential of urdma for use with userspace storage applications. For this experiment, we used 5 nodes with 512 GB RAM, 2 CPU sockets with 11 cores each, and a 10G Ethernet interconnect, Ubuntu 18.04 and crail commit ec2179e8d85fd36ca0572a3178454b581e67d057. Four of the nodes were configured as datanodes and the remaining node was configured as a namenode. For each test case, records of a given size were read or written in batches of sizes 1–8, and we measure the throughput. The results are shown in Fig. 3-11 and Fig. 3-12. For the writeAsync benchmark, the maximum throughput is achievable with 512 KiB and a batch size of 2 or higher. For the readSequentialAsync benchmark, the maximum throughput can be achieved with any message size from 128 kibibytes to 4 mebibytes. The read benchmark likely performs slightly better because crail must allocate blocks before actually doing the write. However, this demonstrates that crail running over urdma is able to maximize the throughput of a 10G network.

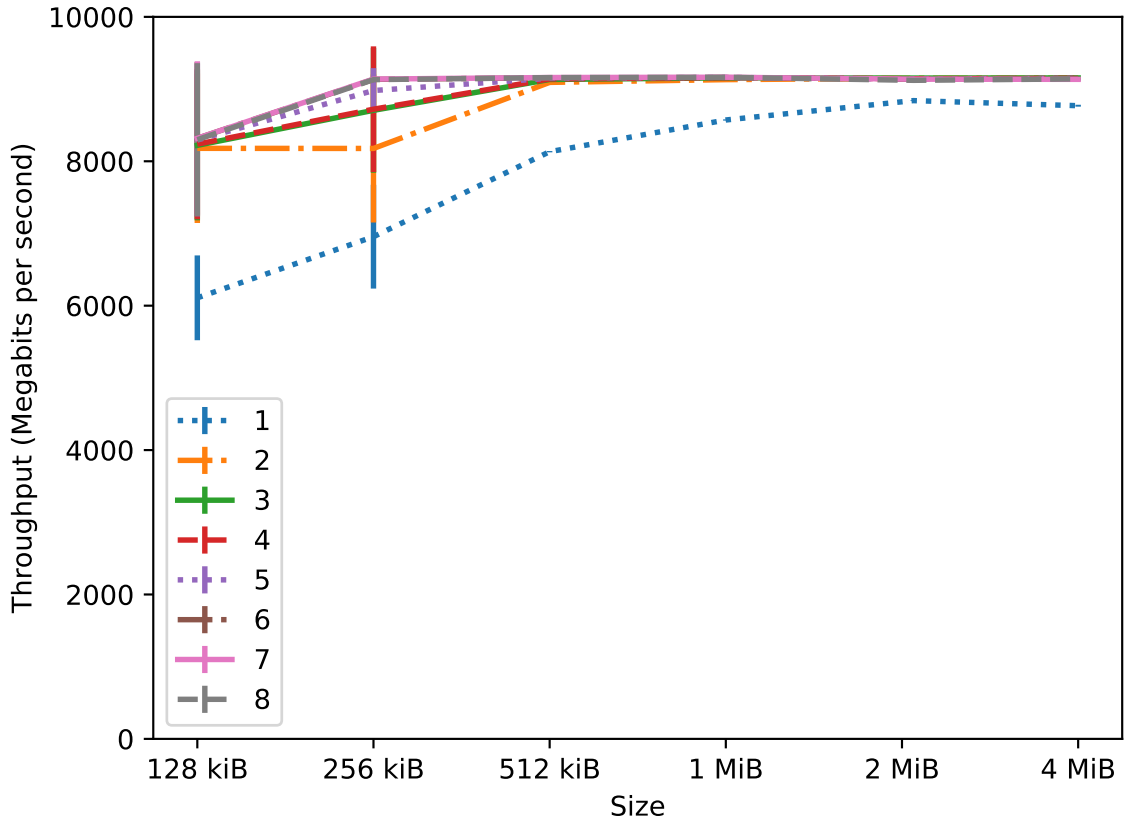

Figure 3-11: Throughput vs. record size for crail iobench tool for different batch sizes, for the writeAsync operation.

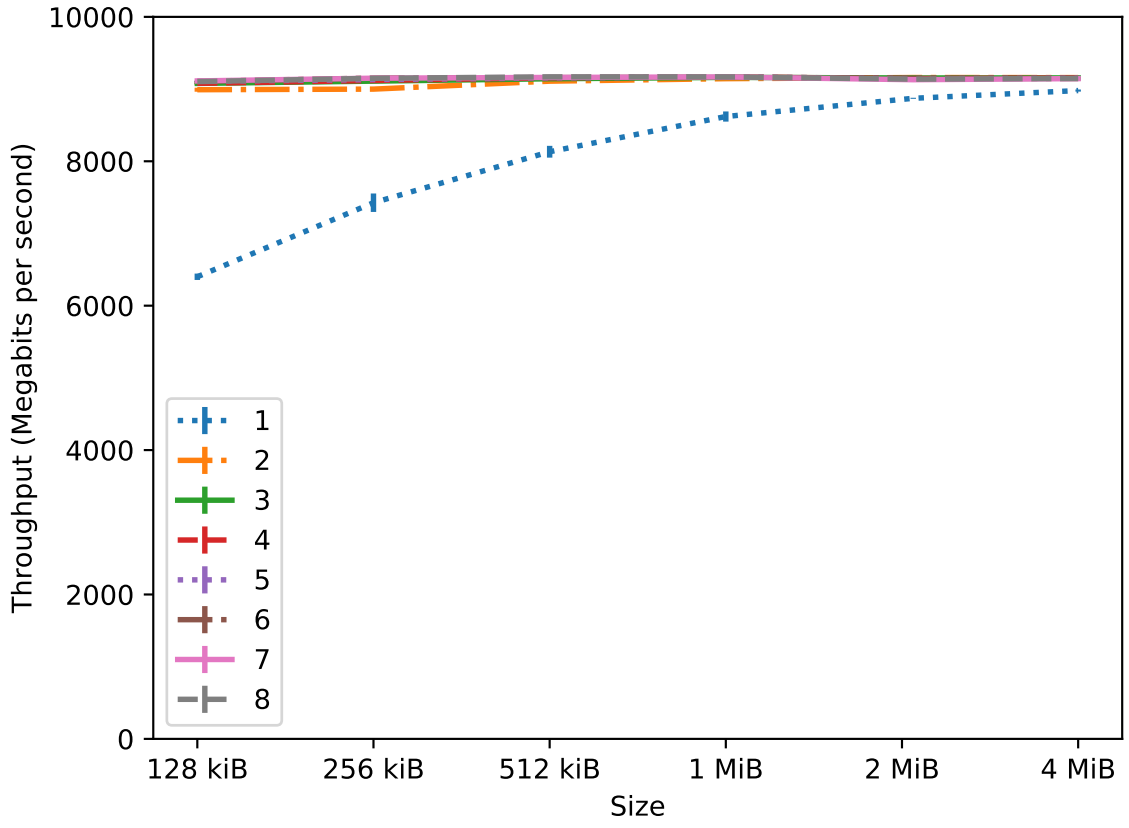

Figure 3-12: Throughput vs. record size for crail iobench tool for different batch sizes, for the readSequentialAsync operation.

3.4 Conclusion

This chapter has introduced *urdma*, a software RDMA implementation using DPDK to perform data transfers in userspace. Unlike prior solutions, urdma does not require any code to execute in the kernel to perform data transfers once the connection is established. Our kernel module, urdma_kmod, is only used for connection management and to provide the stub kernel objects required by the Linux RDMA verbs library to bring up a queue pair. The urdma driver can run unmodified verbs programs and provide performance comparable to or better than softiwarp. We then additionally showed that urdma can be used as the underlying transport for the crail DRAM storage tier, showing that urdma is useful for storage applications.

3.5 Future Work

As previously mentioned, one of the major motivations for urdma is to allow efficient implementation of network storage protocols using RDMA software emulation. As future work, we would like to adapt the NVMf target implementation included with SPDK for urdma. This requires that urdma gains some level of SPDK integration, because the NVMf target uses the verbs library and needs the `ibv_get_device_list` function to include the urdma device(s) in the list, but urdma, by its current design, also needs to trigger DPDK initialization to remain transparent for most users. The simplest level of integration would simply require urdma to perform SPDK initialization itself. However, this would not actually make the implementation more efficient, because urdma performs data copies internally.

A more complex but more efficient integration would require the NVMf target implementation to bypass the verbs API, in order to reuse the buffers that are used to access the local NVMf device. Additionally, the data transfers with the local NVMe device would be done using the same thread that drives RDMA transfers. This provides better CPU cache utilization because metadata for the transfers would not have to be loaded into the CPU caches for multiple CPU cores.

Chapter 4

VERBS OFFLOADED LOCKING TECHNOLOGY

In this chapter, we discuss Verbs Offloaded Locking Technology (VOLT), a novel remote locking solution that is implemented in terms of RDMA verbs operations. In Section 4.1, we discuss the problems with existing remote locking solutions and the motivation for VOLT. In Section 4.2, we discuss the implementation of VOLT, the design of a bytecode extension mechanism for RDMA, and how VOLT would be implemented in terms of this bytecode extension mechanism. We evaluate the performance, safety, and liveness properties of VOLT in Section 4.3, compare to related work in Section 4.4, and discuss our conclusions in Section 4.5.

4.1 Introduction

Distributed applications and middleware require a locking mechanism for synchronization. Although this can be implemented via the atomic compare and swap operation defined by the InfiniBand [22] and iWARP [39] specifications, such an implementation requires that remote applications retry the operation if their initial attempt to take the lock fails. This in turn increases network traffic and host CPU load on the host requesting the lock. Currently, the only other alternative is to implement the locking mechanism purely at the application level, which requires involvement at the host CPU at both the requester and responder. Future high-performance computing systems will use disaggregated memory [37], in which memory is directly attached to the network using a controller with little onboard processing. This will be facilitated by newer system architectures such as Gen-Z[1] which merge the network and system bus together, allowing more direct access to hardware resources on remote nodes.

[1] `http://genzconsortium.org`

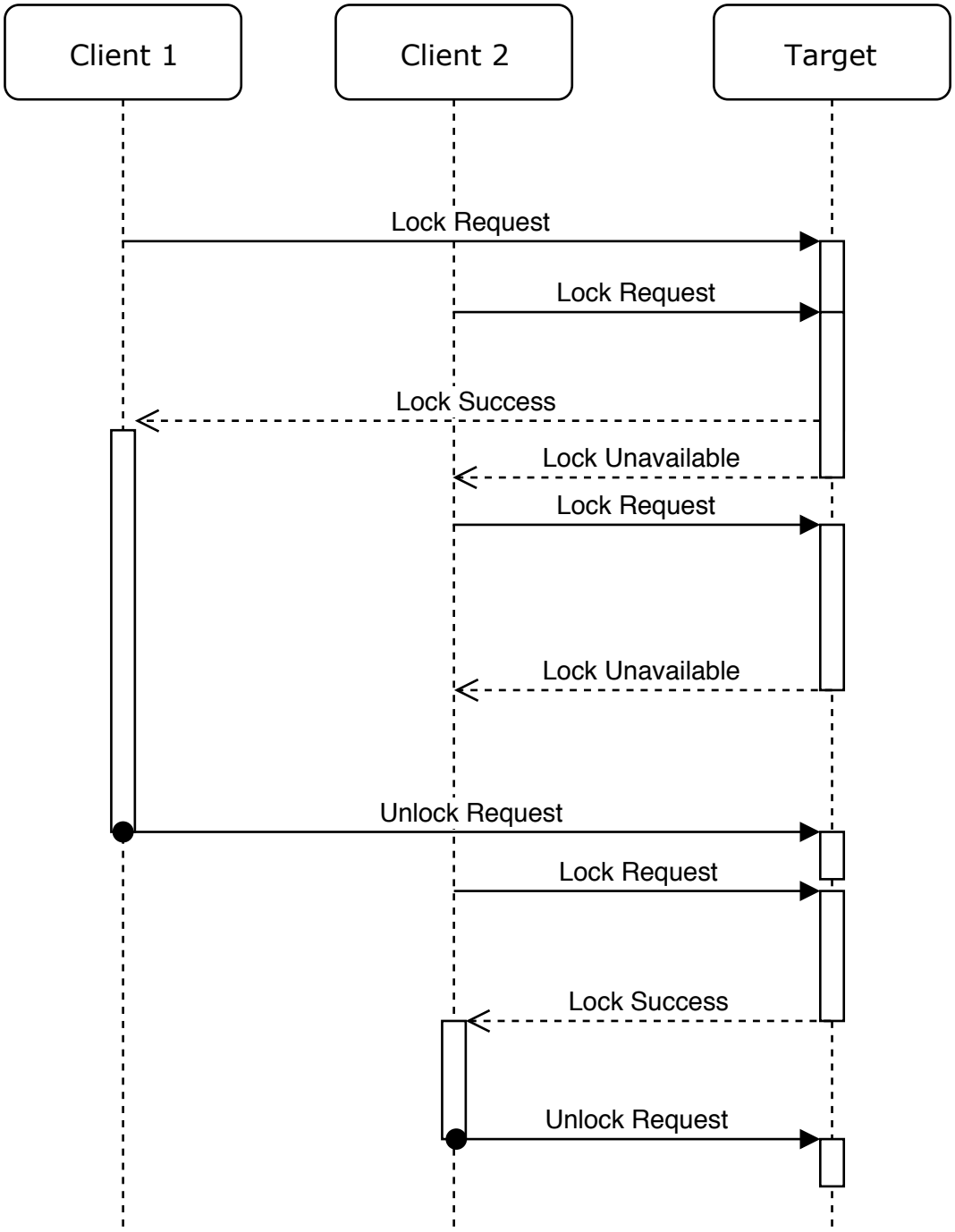

Figure 4-1: An RPC lock mechanism requiring clients to poll for the lock availability.

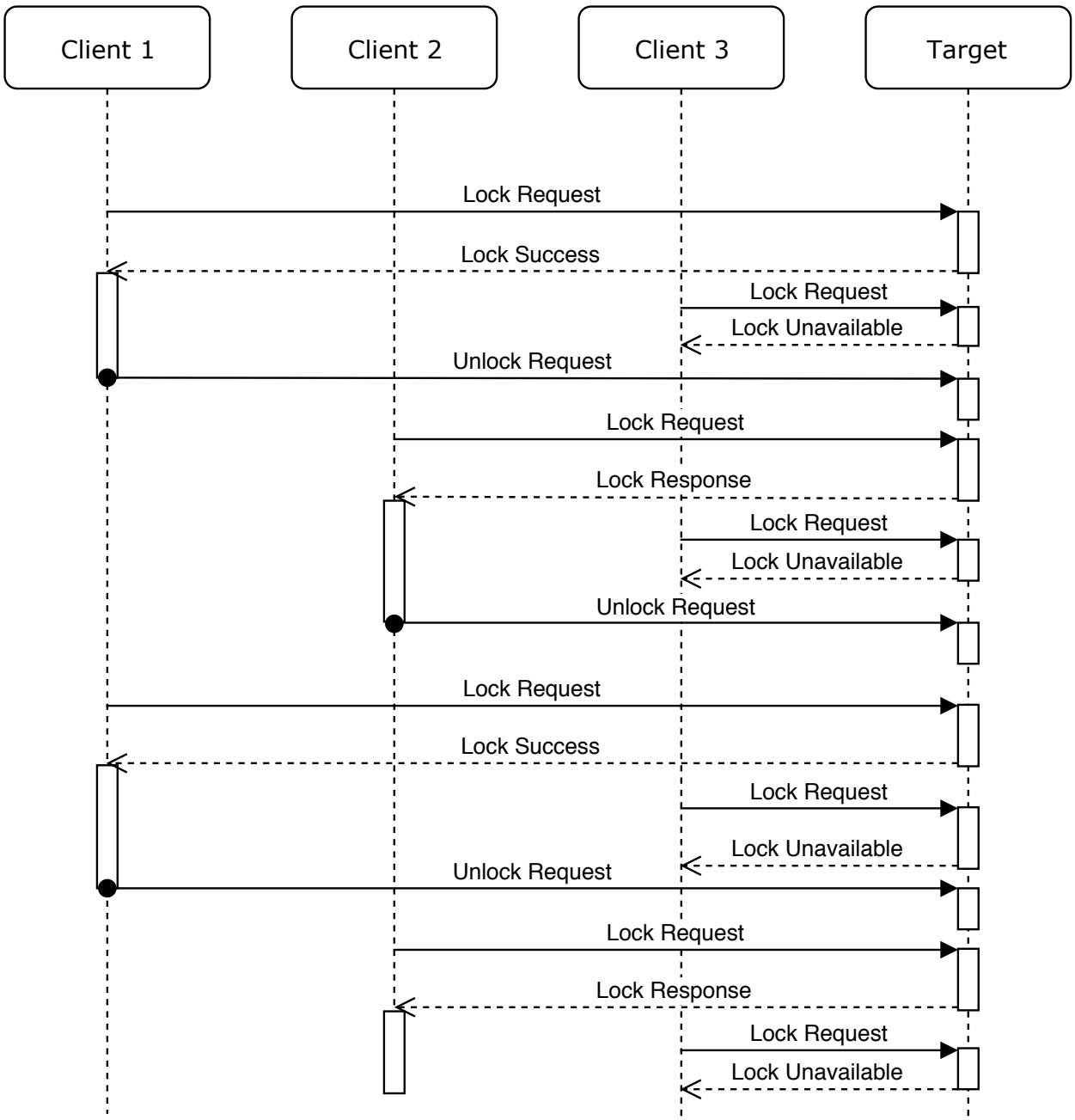

Figure 4-2: Starvation scenario in polling RPC lock mechanism.

RDMA currently provides two ways for an application to implement the messaging for a locking mechanism. In the first approach, the application uses SEND work requests to send a lock request message to a central authority. The central authority will reply with a message indicating whether or not the lock was available, and acquire the lock on behalf of the client application if the lock was available. We will refer to this as the remote procedure call (RPC) approach, because it is based on the design of a typical request-and-response RPC system [38]. This mirrors how remote locking would be implemented using the sockets API with TCP/IP. This locking design requires full participation by the application (and the host CPU) of the client and the server endpoint. However, it can be accomplished with a minimum of 2 messages per lock request and 1 message per unlock request transiting the network in the absence of contention, as we will illustrate.

Fig. 4-1 illustrates the simplest example of the RPC approach. Note that when Client 2 attempts the lock request and it is unavailable, the client must poll the server until it obtains the lock. This wastes network resources and also introduces possible *starvation*—if at least 3 clients are trying to access the lock, it is possible that Client 3 may never succeed in obtaining the lock because Clients 1 and 2 keep trading it back and forth, and Client 3 never polls at the correct time to obtain the lock. We illustrate the possible starvation in Fig. 4-2.

Fig. 4-3 illustrates a design in which the lock target maintains a queue of clients trying to access the lock. In this case, clients do not poll to re-attempt to acquire the lock when it is unavailable; rather, the server queues the request and then sends the Lock Success message when the lock becomes available. This has two advantages over the simple mechanism: (i) fewer messages must traverse the network when the lock is under contention, and (ii) the server may introduce queueing policies, such as first-in-first-out (FIFO), in order to prevent starvation. Fig. 4-4 shows the same scenario as Fig. 4-2 except with server-side queueing, showing how a fair queueing system prevents starvation of a single client. Note, however, that fair queueing depends on the server implementation and that client applications have no control over the server's queueing design. Additionally, the software application at the target now must maintain this queue of clients waiting for the lock, putting additional load onto the target system.

The second method is to use the atomic compare and swap operation that is built into RDMA [22,

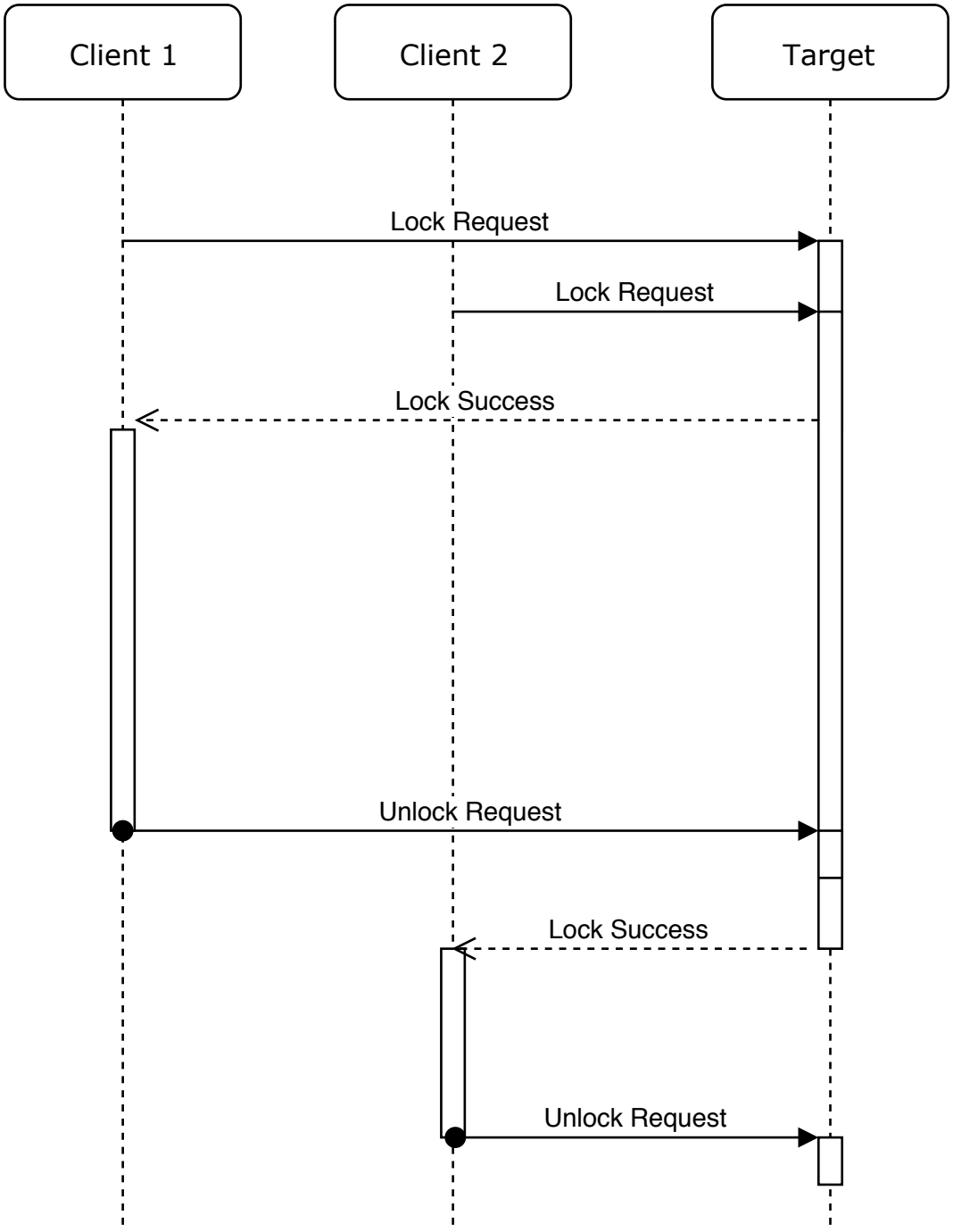

Figure 4-3: An RPC lock mechanism with queuing at the target.

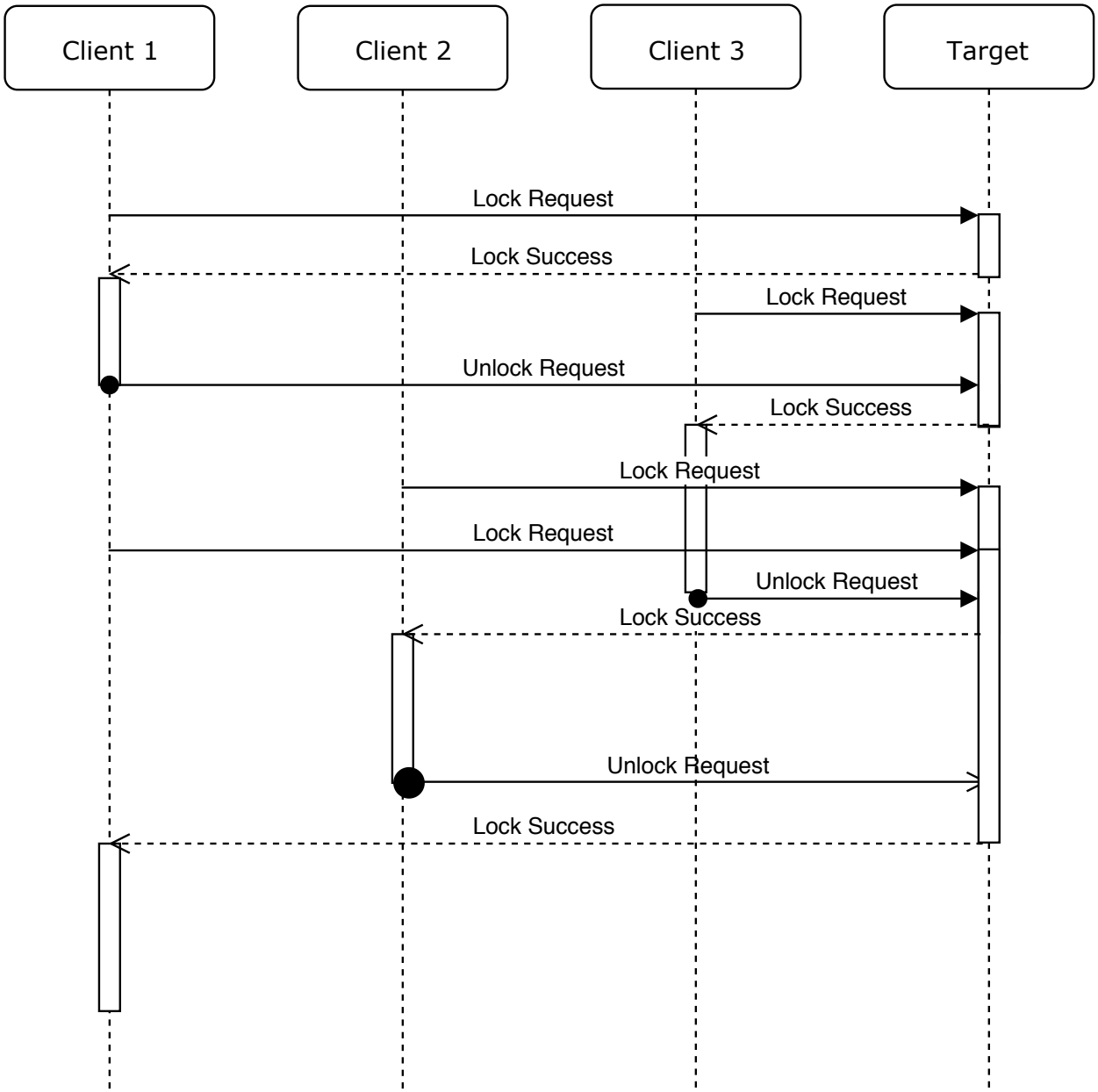

Figure 4-4: Server-side queueing preventing starvation at the target endpoint.

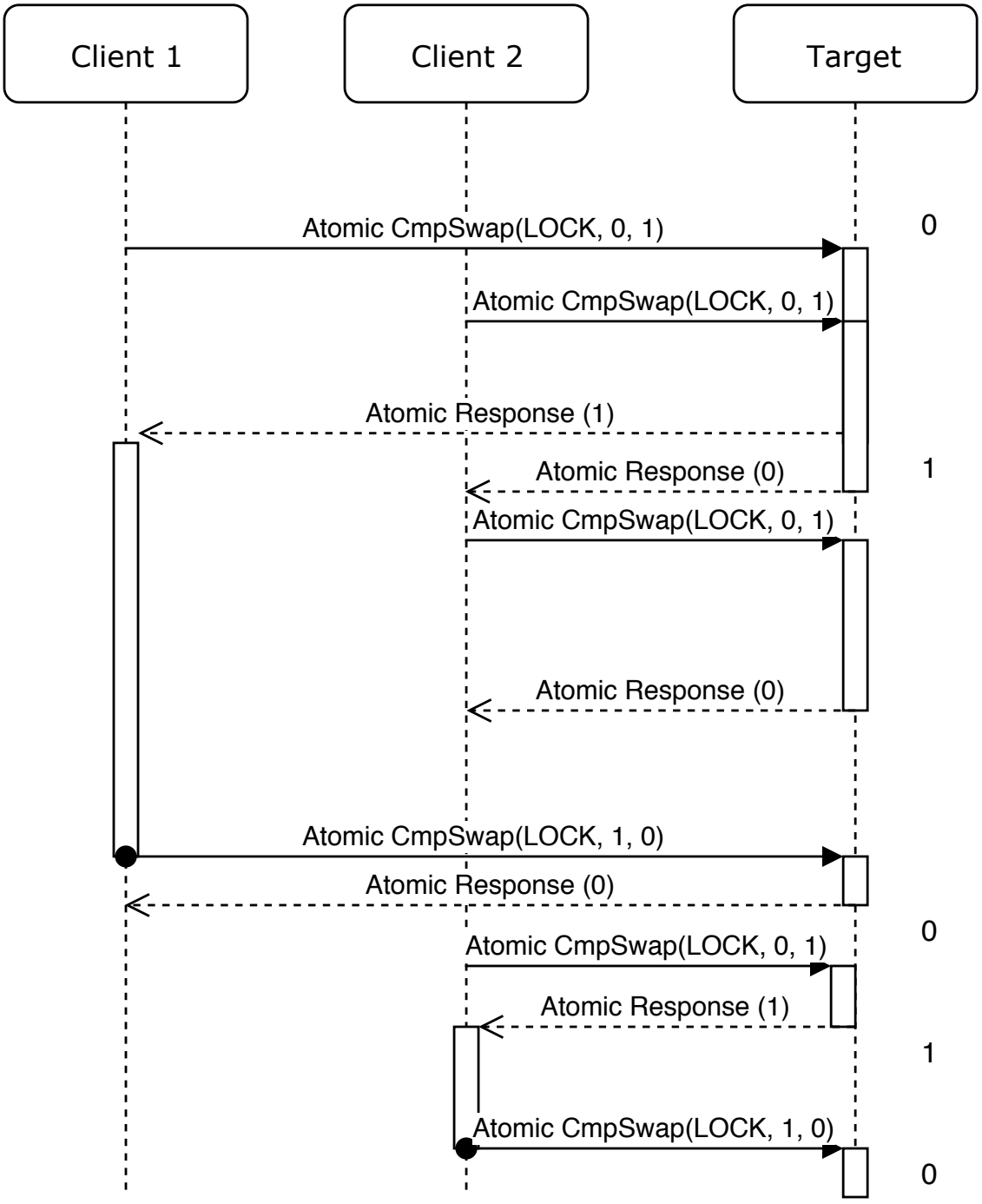

Figure 4-5: A lock mechanism using RDMA atomic operations.

39], which we will henceforth refer to as the Atomic approach and is illustrated in Fig. 4-5. Atomic operations are performed by the HCA with no involvement from the target host CPU. This makes this approach useful when the target is a disaggregated memory node, in which case executing application code at the target is undesirable. At the network level, this requires 2 packets per lock request and 2 packets per unlock request (an atomic request and atomic response for each). However, there is no ability to implement a queueing mechanism—if the lock has already been acquired, the atomic operation will fail and the requesting application must try again. As with the first RPC design, this not only increases both initiator application CPU usage and network usage, but also has no mechanism to avoid starvation. Any queueing policy would have to be implemented as part of the application logic and would be fully reliant on the requesting applications cooperating.

This classification is similar to the local (within a node) locking mechanism classification scheme described by Kagi et al. [73], which describes 4 types of local synchronization mechanisms: local spinning, queue-based locking, collocation, and synchronous prefetch. Local spinning is analogous to the RPC poll and atomic approaches, and queue-based locking is analogous to the RPC queue approach. The remaining two have no current analogous operation in RDMA. Collocation refers to transferring the lock with the data it protects. Synchronous prefetch refers to a CPU prefetching a lock value in such a way that when the current lock holder releases the lock, the requester immediately obtains it.

In this chapter, we define a RDMA LOCK operation, which combines the best aspects of the RPC and Atomic approaches by building a lock operation into RDMA itself. Like the RPC approach, this does not require the application to explicitly retry lock requests. Additionally, like the Atomic approach, this operation will be offloaded from the target host CPU if implemented in hardware, making it useful for "dumb" disaggregated memory nodes. Our RDMA LOCK operation takes advantage of the ordering guarantees that RDMA provides to block queue pair processing until a lock is acquired on the remote node. Since the target node determines the queue pair processing, the target node is free to adjust the order of queue pair processing in order to implement a fairness policy to avoid starvation. We henceforth refer to our RDMA LOCK implementation as verbs offloaded locking technology (VOLT).

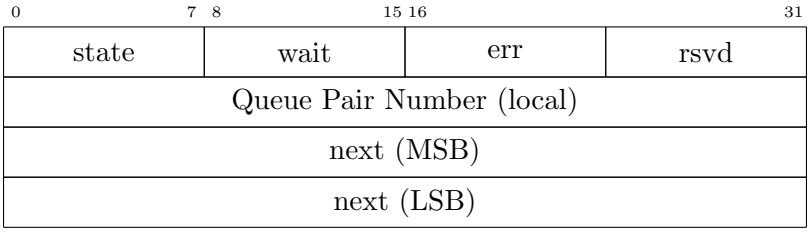

Figure 4-6: RDMA Lock in-memory layout for VOLT.

This RDMA LOCK operation is implemented in urdma using C code. We also design a bytecode extension mechanism for RDMA using enhanced Berkeley Packet Filter (eBPF) [74] that would allow this operation to be implemented in any hardware that supported running eBPF bytecode.

4.2 Implementation

4.2.1 Locking

For each lock object, VOLT uses a small 16 byte region of remote application virtual memory whose contents are controlled by the target HCA in response to RDMA LOCK and RDMA UNLOCK requests from a remote application. The layout of these 16 bytes is shown in Fig. 4-6, with one byte controlling the state of the lock, one byte being used to track the number of queue pairs that are waiting for the lock, an error flag byte, a reserved byte, four bytes for the queue pair number that is currently holding the lock, and eight bytes used for a pointer to thread the list of locks that are held by the queue pair. The list is singly-linked because it is expected that the number of locks held by the queue pair at any given time will be small. The lock's value is set positive when the lock is taken, and zero when the lock is free.

The RDMA LOCK request message format is shown in Fig. 4-7, and the response message format in Fig. 4-8. Both of these messages are untagged, intentionally mirror the iWARP atomic message formats[39], and are targeted at the same iWARP message queues. The first 20 bytes of these messages are the required fields for iWARP untagged messages. We use the same message opcode for lock and unlock requests because there is not much room remaining in the RDMAP opcode space. Future extensions to the locking mechanism could use additional values for the

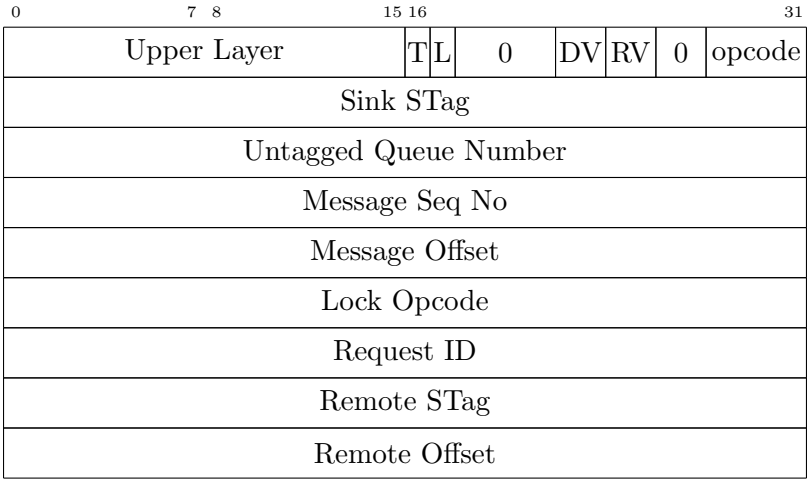

Figure 4-7: RDMA Lock Request iWARP message format. The top fields through Message Offset are iWARP protocol fields; the last four fields starting with Lock Opcode are defined as part of VOLT.

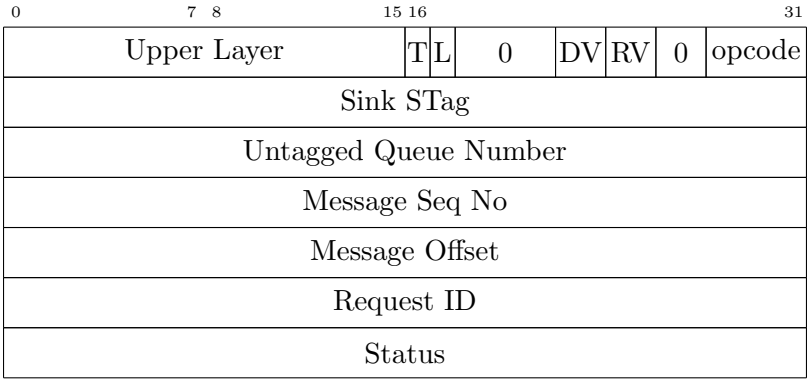

Figure 4-8: RDMA Lock Response iWARP message format. The top fields through Message Offset are iWARP protocol fields; the last two fields starting with Request ID are defined as part of VOLT.

Lock Opcode field. Unlike the RDMA READ Response message but like the Atomic Response message, the Lock Response message is untagged to allow us to pass additional metadata and avoid the need to unnecessarily register memory on the requesting endpoint. Tagged messages in iWARP are limited to pushing data into registered memory at the Sink STag location, and cannot provide any other kind of metadata. On the other hand, we can add as much metadata as needed to an untagged response message. In particular, the response messages contain a Request ID to allow the requester to find the correct request if the requester had issued multiple RDMA Lock requests at the same time. This is the same mechanism as used in Atomic Response messages.

To control waiting for the lock, we rely on the existing ordering guarantees of RDMA operations. All RDMA LOCK operations have an implicit fence indicator, so that any prior RDMA READ operations (or any other outstanding operations on the send queue) are guaranteed to have completed prior to the lock being acquired. Thus, when a lock request is at the top of the work queue, all further operations on the send queue are blocked until the lock is acquired or the queue pair is moved to the error state. This can be used to implement collocation [73] by posting a RDMA READ or RDMA WRITE operation immediately after the RDMA LOCK in the same work request list.

The remote endpoint which has acquired the lock thus enters its critical section and is able to use RDMA READ and RDMA WRITE requests to manipulate state controlled by the lock without interference from other endpoints. The lock is released when the controlling endpoint issues an RDMA UNLOCK request or the queue pair is transitioned to the error state. In the former case, the next endpoint to acquire the lock then controls the state as normal. In the latter case, when the controller of the lock is no longer reachable, the next endpoint to acquire the lock will receive a completion with error status indicating that the lock was acquired due to queue pair termination and that the critical section may not have been completed successfully. This endpoint must attempt to recover the state of any shared remote memory region for which the lock was controlling access. Such recovery mechanisms are highly application-specific and outside of the scope of the locking mechanism.

Handling failure of the server requires more complex logic. If the server does not persist its state, then bringing up a fresh instance of the server will result in locks being available again

and completely empty state. Some key-value store applications such as memcached [75] are used entirely as a cache mechanism, and this behavior is fine, since client applications will simply retry the operation from the actual datastore behind the cache. However, server applications which persist state into non-volatile memory may have locks in the held state when the server restarts, except that the queue pair that held the lock no longer exists. A server application may reset any such locks by clearing the backing memory of the lock to all zeros. However, recovery of the underlying data structure is again up to the application. One approach to recover such state would be to not allow normal client connections until a special "fsck" client runs and resets any state due to transactions in progress.

4.2.2 Bytecode Extension Mechanism

The enhanced Berkeley Packet Filter (eBPF) [74] was designed by the Linux kernel community based on the classic Berkeley Packet Filter (cBPF) developed by McCanne and Jacobson [76]. As its name suggests, the original purpose for cBPF was to provide a simple language for userspace to filter packets on a raw socket. eBPF provides a simple bytecode language with a small number of registers, the usual arithmetic, conditional, and branching operations, and the ability to map values into a small memory region. One can also compile C code that meets the appropriate constraints into eBPF bytecode, giving flexibility to developers. eBPF is also designed to be easy to verify for safety, i.e., a short verifier program can ensure that an arbitrary eBPF program cannot jump beyond the length of the program or access arbitrary system memory. Additionally, one limitation is that eBPF programs are not allowed to contain loops, to ensure that they always terminate without having to solve the halting problem. These properties make eBPF ideal for allowing userspace applications to supply small programs to execute in the kernel to perform packet filtering and debug logging control.

Using eBPF, we can provide a way for RDMA applications to supply custom queue pair operations to run on an HCA which supports running eBPF bytecode. This requires an API to solve several problems. First, the eBPF programs must be able to maintain state between invocations, which is unusual for eBPF programs as they are usually used as passive filters. This state must

```
1  struct verbs_ebpf_ops {
2    size_t qp_state_size, wqe_state_size, size_t pkt_size;
3    int opcode;
4    void (*send_wqe_post)(struct pt_regs *ctx, struct ibv_send_wr *wr);
5    int (*send_wqe_transfer)(struct pt_regs *ctx, void *wqe);
6    int (*send_wqe_wait)(struct pt_regs *ctx, void *wqe);
7    void (*send_wqe_complete)(struct pt_regs *ctx, void *wqe);
8    bool (*pkt_match)(struct pt_regs *ctx, uint8_t *pkt);
9    bool (*pkt_place)(struct pt_regs *ctx, uint8_t *pkt);
10   void (*msg_deliver)(struct pt_regs *ctx, void *pktctx);
11   void (*qp_setup)(struct pt_regs *ctx, void *qp);
12   void (*qp_teardown)(struct pt_regs *ctx, void *qp);
13 };
14
15 int verbs_attach_epbf_op(struct verbs_epbf_ops *ops);
```

Figure 4-9: Functions required for verbs operations implemented in eBPF.

be reliably torn down if the queue pair goes into the error state for any reason. We show a C struct defining the methods that such an operation would need to define in Fig. 4.2.2. We use a pair of methods, `qp_setup` and `qp_teardown`, for this. The operations receive a buffer of size `qp_state_size` which is kept as part of the queue pair state.

Second, the operation must be able to accept send work requests. These operations assume the same send work request state machine as shown in Fig. 3-4 in Sec. 3.2.3. The RDMA implementation is responsible for maintaining the state machine and calling the correct eBPF methods in the correct states. The `send_wqe_post` method records the initial state of the operation into a WQE, which will start in the INIT state. When the WQE moves to the head of the queue, the RDMA implementation transitions it into the TRANSFER state and calls `send_wqe_transfer`. This method returns the next state for the WQE, and the implementation will continue to call the method until it returns a different state, which will usually be WAIT. The WQE can only transition from WAIT to COMPLETE when all segments of all messages sent by the requester as part of the operation have been acknowledged by the lower-level transport protocol at the receiver. However, this is a necessary but insufficient condition for many RDMA operations, including RDMA READ and atomic operations. Thus, the `send_wqe_wait` method will be called to determine whether to stay in the WAIT state or transition to the COMPLETE state. The `send_wqe_complete` method

```
1  #define MAX_LOCKS 4
2  struct volt_qp_info {
3    struct rdma_qp_lock *held_locks[MAX_LOCKS];
4  };
5
6  void volt_qp_setup(struct pt_regs *ctx, struct volt_qp_info *qpinfo)
7  {
8     qpinfo->lock_list_head = NULL;
9  }
10
11 void volt_qp_teardown(struct pt_regs *ctx, struct volt_qp_info *qpinfo)
12 {
13   struct rdma_qp_lock *ptr;
14   for (x = 0; x < MAX_LOCKS; ++x) {
15     if ((ptr = qpinfo->locks_held[x])) {
16       ptr->err = 1;
17       ptr->lock = 0;
18     }
19   }
20 }
```

Figure 4-10: Function implementations required for verbs operations implemented in eBPF, written in C pseudocode.

simply builds a CQE from the WQE, and tears down the state associated with the WQE.

Finally, the operation must be able to receive packets corresponding to the operation. This requires three methods. First, the `pkt_match` method simply returns true if the packet should be processed by this particular operation. We then split the operation into a data placement phase and a delivery phase, as defined by the iWARP specification [26]. If the packet matches the operation, then the `pkt_place` method is called, which performs the data placement part of the operation and returns true if all data has been placed, or false otherwise. Once all data has been placed, and all prior messages have been delivered, then the `msg_deliver` is called. In the first version of this system, all custom operations are considered to be strictly ordered and have an implicit fence bit. Allowing different ordering requirements for custom operations is future work.

4.2.3 RDMA Locking in Bytecode

We use VOLT as an example. Fig. 4-10 illustrates QP setup and teardown methods as C pseudocode, which would be compiled down to eBPF bytecode. In this case, the only state kept for

the queue pair is the currently held locks. This must be limited to some arbitrary value because eBPF does not support loops, and the loop illustrated in our example would be unrolled by the compiler. We choose four in this case because we anticipate that most applications will not need to hold many locks at the same time, and application designs that require holding an unbounded number of locks would have too much lock contention to have good performance. For instance, the most efficient B-link-tree insertion algorithm only needs to hold one lock at a time [55]. Even a less efficient prior iteration of the B-link-tree algorithm could lock a maximum of three nodes at a time, reaching the maximum only in an infrequent worst case scenario [77][2]. An application could open multiple queue pairs in the event that it needed to hold more than four locks simultaneously.

Fig. 4-11 shows the implementation of send WQE operations. The `volt_send_wqe_post` operation simply uses a helper function to allocate a WQE and then fills in the appropriate metadata. The NIC is responsible for enqueueing the WQE once it is filled in correctly. When the operation reaches the front of the queue, the `volt_send_wqe_transfer` method is run. This operation is responsible for creating packets and placing them on the wire. In the case of VOLT, only a single packet needs to be transmitted, and when the packet has been enqueued, then the function returns the `SEND_WQE_WAIT` state from the diagram in Fig. 3-4. Finally the `volt_send_wqe_complete` operation is used to fill in a CQE (completion queue entry) to place in the completion queue when the operation completes.

Fig. 4-12 shows the packet retrieval implementation for VOLT as it would be implemented in terms of eBPF. First, the `volt_pkt_match` function is called for each incoming packet that the NIC cannot natively support, and is used to determine if the incoming packet corresponds to this operation. If the packet matches, the `volt_pkt_place` function is used to place any data into application memory; in the case of VOLT this is a no-op because lock and unlock requests are

[2]This case is when the insertion is being done into a node whose ancestor is in the process of being split. The locks held are the original node of the split, its parent, and a single node to the right of the parent, as the algorithm traverses the right links to find the correct insertion point. Sagiv's improved solution observes that it is unnecessary to prevent one update operation from overtaking another update operation, and thus locks are only necessary to ensure that each node update is atomic.

```
1  struct volt_wqe *volt_send_wqe_post(struct pt_regs *ctx,
2      struct ibv_send_wr *wr)
3  {
4    /* alloc_wqe fills in common fields from the WR */
5    struct volt_wr_data *wrdata = wr->custom;
6    struct volt_wqe_data *wqe = alloc_wqe(qp, wr);
7    wqe->opcode = wrdata->opcode;
8    wqe->remote_addr = wrdata->remote_addr;
9    wqe->rkey = wrdata->rkey;
10   return wqe;
11 }
12
13 int volt_send_wqe_transfer(struct pt_regs *ctx, struct send_wqe *wqe)
14 {
15   struct volt_wqe_data *wqedata = wqe->custom;
16   pkt = alloc_untagged_pkt(sizeof(struct rdmap_lockreq_pkt));
17   bpf_map_insert(cur_qp->wqe_map, pkt->untagged.msn, wqe);
18   pkt->volt_opcode = byteswap(wqedata->volt_opcode);
19   pkt->volt_req_id = pkt->untagged.msn;
20   pkt->remote_stag = byteswap(wqedata->rkey);
21   pkt->remote_offset = byteswap(wqedata->remote_addr);
22   send_pkt(wqe, pkt);
23   return SEND_WQE_WAIT;
24 }
25
26 struct volt_cqe *volt_send_wqe_complete(struct pt_regs *ctx,
27     struct volt_wqe *wqe)
28 {
29   struct volt_cqe *cqe = alloc_cqe(wqe);
30   *wqe->sge_list[0].addr = wqe->status;
31   return cqe;
32 }
```

Figure 4-11: Pseudocode for implementations of WQE operations in eBPF.

```
1   bool volt_pkt_match(struct pt_regs *ctx, uint8_t *pkt) {
2     switch (rdmap_get_opcode(pkt)) {
3     case RDMAP_VOLT_LOCK_REQUEST:
4     case RDMAP_VOLT_LOCK_RESPONSE:
5       return true;
6     default:
7       return false;
8     }
9   }
10
11  bool volt_pkt_place(struct pt_regs *ctx, uint8_t *pkt) {
12    /* no-op */
13    return true;
14  }
15
16  void volt_msg_deliver(struct pt_regs *ctx, uint8_t *pkt) {
17    struct volt_req *req = alloc_rdma_responder_resource(get_qp(ctx));
18    req->opcode = pkt->lock_opcode;
19    req->req_id = byteswap(pkt->volt_req_id);
20    req->addr = byteswap(pkt->lock_addr);
21    req->stag = byteswap(pkt->stag);
22  }
23
24  void volt_advance_rdma_req(struct pt_regs *ctx, struct volt_req *req) {
25    switch (req->opcode) {
26    global_mutex_lock(ctx);
27    case VOLT_OPCODE_LOCK:
28      if (*req->addr == 0) {
29        *req->addr = 1;
30        done = true;
31      }
32      break;
33    case VOLT_OPCODE_UNLOCK:
34      *req->addr = 0;
35      done = true;
36      break;
37    }
38    global_mutex_unlock(ctx);
39    if (done) {
40      struct volt_wqe_data *wqedata = wqe->custom;
41      pkt = alloc_untagged_pkt(sizeof(struct rdmap_lockresp_pkt));
42      bpf_map_insert(cur_qp->wqe_map, pkt->untagged.msn, wqe);
43      pkt->volt_opcode = byteswap(wqedata->volt_opcode);
44      pkt->volt_req_id = req->req_id;
45      pkt->status = byteswap(0);
46      send_pkt(wqe, pkt);
47      free_rdma_responser_resource(req->qp);
48    }
49  }
```

Figure 4-12: Pseudocode for implementations of packet retrieval operations in eBPF. For brevity, error cases are ignored.

strictly ordered. The `volt_msg_deliver` function creates an RDMA READ-style responder resource to handle the locking or unlocking of the mutex. Responder resources are used for operations that require processing beyond simply placing data in memory and delivering a local completion. In the case of VOLT, a responder resource is necessary because we need to handle the case where the lock is not available and retry.

Finally, the `volt_advance_rdma_req` is used to actually progress the operation. It is called as part of normal queue pair processing when the NIC is ready to process pending RDMA READ operations. This operation must do the real work of acquiring or releasing the lock. In this case, we illustrate taking a global lock, which would be required if a NIC may perform operations on multiple queue pairs concurrently. When the NIC is able to acquire or release the lock, the operation then sends a response packet and releases the responder resource.

4.3 Evaluation

As we currently only have a software implementation of this mechanism, we do not have the means to measure the effect of CPU offload that could be provided by a hardware implementation. However, we can measure the overhead of locks implemented using RDMA atomic compare and swap operations.

In Fig. 4-13, we show the performance of 1–4 client applications sending simultaneous lock/unlock requests for a remote object, the using RPC, Atomic, and VOLT approaches. We show the time taken for 100,000 lock/unlock cycles. We first examine the existing approaches. The RPC queue method vastly outperforms the RPC poll method, which is an unexpected result, as we expect them to perform closely. The RPC poll method also has the largest variance in performance. It is likely that the polling magnifies the contention and that threads are barely "missing" acquiring the lock and must resend the request, and that the number of "extra" requests magnifies the contention. However, this does not explain why the performance is significantly worse for the single thread case which has no contention. The RPC queue method, on the other hand, appears to exhibit the same performance for between 1 and 3 threads, and only slows down under contention at 4 threads. This indicates that the overhead of message processing by the application is the dominant component of

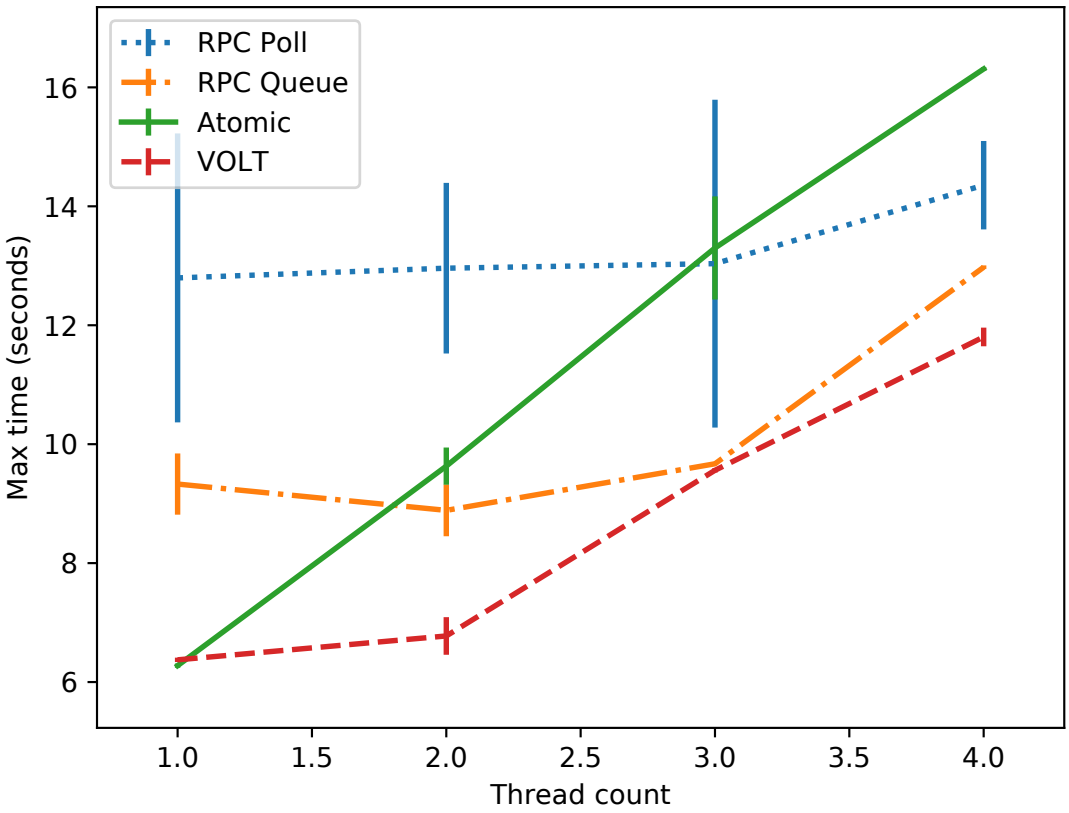

Figure 4-13: Time taken and lock cycles per second for 100,000 lock cycles for each locking scheme.

the time taken. The time taken for the Atomic method scales linearly with the number of threads, which makes sense given that it is simply flipping a single bit at the destination node. VOLT appears to have nearly identical performance for one or two threads, and scales linearly beyond 3 threads. This shows that having extra network traffic due to contention significantly impacts performance, and thus VOLT can significantly reduce overhead over the Atomic locking method by leveraging the queue pair mechanism to avoid extra message exchanges.

4.4 Related Work

Buntinas, et al., discuss the implementation of a locking primitive as part of the Aggregate Remote Memory Copy Interface (ARMCI) [78]. They discuss an existing implementation which uses a ticket-based mechanism, where each lock consists of a ticket and a counter. Every time a client requests the lock, the ticket number is incremented and the previous value is associated with the client request. The server increments the counter each time the lock is released, and the client request is satisfied when its ticket number is equal to the counter value. This simple mechanism ensures that each request can be uniquely identified and that requests are served in order. This solves the fairness problem by ensuring that requests are served in FIFO order. The authors note that this approach requires that a message be sent to the server every time that a lock is released.

The authors implement a software-based solution in which the lock is literally defined as a linked list of requests. Each request has a flag indicating whether or not the lock is available. Each process has a linked list pointer which starts out NULL. To request the lock, the process performs an atomic swap with the lock value. If the process acquires the lock, then the process will get a NULL pointer back. Otherwise, it has a pointer to the end of the lock request linked list, and adds itself to the list and begins polling on the lock flag in its entry. To release the lock, the process performs an atomic compare-and-swap to set the lock value to NULL if the lock value still points at its request (i.e., no other process has submitted a request for the lock). If there is a pending request, the releasing process will flip the lock flag on that request object, and the requesting process will poll the lock flag and acquire the lock.

This design used in ARMCI is essentially a variant on the Atomic implementation discussed

earlier, but requires more memory and bookkeeping than the simple design we presented. In particular, linked lists are less suited for a hardware implementation due to the need for dynamic memory allocation. Additionally, this implementation expects the whole distributed application to have a single global virtual address space. Finally, the design requires polling, although the effect of the polling is minimized due to the polling being done on a memory address local to the node requesting the lock, instead of a memory location on the server. However, this is still more overhead than using the queue pair itself as the lock mechanism, because with VOLT, an application can choose to leverage the CQ notification mechanism to avoid the CPU overhead of a busy poll loop.

4.5 Conclusion

We have developed a novel and unique remote locking solution using a new RDMA verbs operation to offload the lock management to an RDMA NIC. This solves problems with existing solutions. Unlike the RPC solution, this solution avoids loads on the CPU at the responder, and unlike a solution based on existing RDMA atomic operations, this does not require polling on the client and enables the remote RDMA NIC to implement a queueing policy for the locks. We showed a performance evaluation of the locking mechanism compared to the RPC and atomic locking approaches.

We additionally developed a novel scheme for allowing extensions to existing RDMA hardware by leveraging eBPF bytecode. This mechanism allows an application to inject a new operation into existing RDMA NICs via a handful of eBPF programs. We then illustrated how our locking implementation can be implemented using this bytecode mechanism.

Chapter 5

CONCLUSION

5.1 Conclusions

In this conclusion, we discuss the results of this dissertation and answer the questions from Section 1.4.

Kernel overhead has significant impact on software RDMA and storage access solutions. Current software RDMA emulation drivers are implemented in the kernel, requiring userspace applications to context switch into the kernel to perform data transfers. This produces significant overhead for RPC applications which exchange small messages relatively infrequently. On the other hand, data transfer applications which exchange frequent large messages are less affected because kernel software implementations perform context switches only when no operation is reading the queue and thus the cost of the small number of context switches is amortized by the number of data transfers. Additionally, kernel Ethernet drivers are interrupt-based, meaning that an interrupt is usually received for every packet. Modern Ethernet drivers will coalesce interrupts for multiple packets received together within a short interval of time, but this is a trade-off between the number of context switches vs. the delay before the kernel is informed of a received packet.

We have developed urdma, a software RDMA driver which performs data transfer in userspace implemented using DPDK. While urdma has a kernel component, this is only used for verbs and connection setup. DPDK provides userspace poll-mode drivers for controlling the Ethernet NIC, so there is no interrupt overhead. We have demonstrated that urdma can provide better latency and similar throughput to softiwarp. We then additionally showed that urdma can be used as the underlying transport for the crail DRAM storage tier, and can achieve close to the maximum throughput of the underlying network.

We have also designed an extension for urdma providing a remote locking protocol controlled via RDMA operations, which we refer to as VOLT. This provides an alternative to existing RPC and atomic-based methods for acquiring and releasing locks. Our method provides lower CPU utilization and network utilization compared to the existing solution, while being able to be used as a mutex for distributed applications.

VOLT only requires 8–12 bytes in memory per lock, meaning that it is easy to add the lock to objects within a data structure. While the eBPF implementation places a limit on the number of locks that can be held simultaneously by a single client, there is no limit on the number of locks that can exist within a system. This makes VOLT usable with large distributed data structures such as a B-tree, as each node can have its own independent lock.

Because VOLT controls the queue pair ordering, it ensures that an application may queue an RDMA READ or RDMA WRITE request that will execute only after the lock has been acquired. This is unlike using atomics for locking, where the application must check the result of the atomic operation to ensure that it succeeded before queueing an operation on that object, and repeat the atomic operation on failure. This means that VOLT can be used to more efficiently synchronize applications which use one-sided RDMA operations on shared objects.

VOLT is designed to survive the failure of client nodes, even if the client node is currently holding the lock. If a client node's connection fails while it is holding the lock, the server will automatically release the lock and the next client to acquire the lock will get an error indication. While it is up to the client application to correctly handle the error and reset the shared state of the locked object, this capability means that VOLT can withstand the failure of a single node.

The current design of VOLT can in theory be used to synchronize access to objects in a multiversioned data structure; however, it is not perfectly suited for this. In particular, distributed data structures tend to be modified via transactions, and in general a transaction is rolled back if locks cannot be acquired. As future work, a possible extension to VOLT that would fix this weakness would be to implement a conditional lock operation, that would only return the lock if another memory value has a given value. This could be used to check the version number of a data structure. However, a naïve implementation of this would defeat one of the advantages of VOLT:

an RDMA operation could no longer be queued after the lock, since there would no longer be a guarantee that the lock would have been acquired. A massive overhaul to the queue pair mechanism would be required in order to allow for general conditional queueing of verbs operations.

We provide an extension mechanism to add new RDMA operations to existing hardware, using a small set of programs compiled to eBPF bytecode. Any RDMA hardware which implemented an eBPF bytecode interpreter could then run these programs to provide custom operations. We demonstrate that VOLT can be implemented in terms of this extension mechanism by showing an implementation in C pseudocode.

5.2 Future Work

In order to evaluate the locking implementation, one could implement a simple object system for RDMA called the RDMA Object System (ROS). ROS is a simple versioned key-value object store that is implemented in terms of RDMA verbs operations. ROS will use redo logging [68] to ensure atomicity of transactions, and use VOLT (Chapter 4) for synchronization between client nodes during transaction commit. One of the central design decisions for ROS will be to make the system client-driven whenever possible, with as little work as possible being done in the server nodes. This will enable ROS to work in a disaggregated memory setup, where each disaggregated memory node can act as an independent server.

In the initial design for ROS, servers will only need to know how to allocate shared objects and transaction logs. Each client is assumed to require only a single transaction at a time, and will thus receive the virtual address and rkey corresponding to a redo log buffer when it connects to a server. Because the transactions are redo-based, the client may start a transaction at any time without needing to notify any servers. The client is then free to read objects and write updates to its redo log until it is ready to commit. The client will record the version number associated with each object it is reading or writing. The updates in the redo log are not made available to other clients until the client is ready to commit. The commit process requires the client to use VOLT to lock each object that it plans to update, and verify that the version number matches. If the version number of any object does not match the originally recorded version number, the client will release

all locks and retry the transaction. Once all objects are locked and version numbers verified, the client issues a COMMIT operation [79, 58] to commit the redo log to non-volatile storage. Finally, the client will update the contents of the live objects based on the contents of the redo log before releasing the locks. This two-phase commit procedure is loosely inspired by DudeTM [65], with the exception that other clients will not see the updated version until it is fully committed, so that clients can read an object with a single RDMA READ request.

Using this object system, one could implement a B-tree structure which is distributed among multiple nodes such that each node may be stored in multiple locations for redundancy. Previous work [50, 51, 53, 54, 37] demonstrates variations on B-tree which are either durable but not distributed, distributed but not durable, and/or do not have full RDMA support. One could synthesize these approaches to create a B-tree which satisfies all of the ACID guarantees. My implementation will allow clients to not only perform lookups using RDMA READ operations, but also to perform modifications using RDMA WRITE operations, taking advantage of the remote locking primitive defined in the previous section. Like the previous cited work, each B-tree entry will be versioned, and for simplicity only the leaves will contain pointers to data values. The tree's current version will be stored with the root of the tree and used in lookups. The distributed nodes will be replicated permanently on multiple nodes; existing distributed B-trees cache remote nodes for efficiency but do not use these replicas for fault tolerance. This data structure will provide a high-level C++ or Java interface to lookup, insert, update, and remove elements.

This would allow an application to use RDMA READ and RDMA WRITE operations to manipulate shared persistent data stored in the B-tree. However, the application must explicitly flush the memory region to make the data durable, in order to ensure that the data made durable is in fact consistent, and that restarting the node after a power failure will result in a valid application state.

Under this scheme, synchronization of replicas is feasible as each node consists of entries whose pointers are immutable between when they are inserted and when the last reference to their deleted version has disappeared. Unlike a traditional B-tree, deletion is accomplished by setting the deleted version field, which can be done using a single atomic operation on the primary replica in the

majority of cases. The replicas can then be updated with a single RDMA WRITE operation, since the success of the atomic operation guarantees that the update has succeeded.

BIBLIOGRAPHY

[1] D. Nowitz, *UUCP implementation description*, 1978, vol. 2.

[2] J. Postel and J. Reynolds, "File Transfer Protocol," Internet Request for Comments (INTERNET STANDARD), RFC Editor, RFC 959, Oct. 1985, updated by RFCs 2228, 2640, 2773, 3659, 5797, 7151. [Online]. Available: http://www.rfc-editor.org/rfc/rfc959.txt

[3] S. Shepler, M. Eisler, and D. Noveck, "Network File System (NFS) Version 4 Minor Version 1 Protocol," Internet Request for Comments (Proposed Standard), RFC Editor, RFC 5661, Jan. 2010, updated by RFC 8178. [Online]. Available: http://www.rfc-editor.org/rfc/rfc5661.txt

[4] (2015) Netware control protocols. Novell. [Online]. Available: http://www.novell.com/developer/ndk/netware_core_protocols.html

[5] "[MS-SMB]: Server message block (SMB) protocol," Microsoft Corporation, Tech. Rep., Oct. 2013. [Online]. Available: http://msdn.microsoft.com/en-us/library/cc246231.aspx

[6] D. A. Patterson, G. Gibson, and R. H. Katz, "A case for redundant arrays of inexpensive disks (RAID)," in *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '88. New York, NY, USA: ACM, 1988, pp. 109–116. [Online]. Available: http://doi.acm.org/10.1145/50202.50214

[7] "Fibre Channel physical interfaces (FC-PI-5)," ANSI INCITS 479-2011, 2011.

[8] "Fibre Channel framing and signaling (FC-FS-2)," ANSI INCITS 470-2011, 2011.

[9] *INCITS 515: Information technology - SCSI Architecture Model - 5 (SAM-5)*, ANSI INCITS Std. 515-2017, Feb. 2017.

[10] M. Chadalapaka, J. Satran, K. Meth, and D. Black, "Internet Small Computer System Interface (iSCSI) Protocol (Consolidated)," Internet Request for Comments (Proposed Standard), RFC Editor, RFC 7143, Apr. 2014. [Online]. Available: http://www.rfc-editor.org/rfc/rfc7143.txt

[11] "Fibre Channel backbone 5 revision 2.00 (FC-BB-5)," American National Standard for Information Technology, International Committee for Information Technology Standards Technical Group T11, 4 Jun. 2009.

[12] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google file system," in *ACM SIGOPS Operating Systems Review*, vol. 37. ACM, 2003, pp. 29–43.

[13] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop distributed file system," in *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, 2010, pp. 1–10.

[14] P. Schwan, "Lustre: Building a file system for 1000-node clusters," in *Proceedings of the 2003 Linux Symposium*, vol. 2003, 2003. [Online]. Available: https://ols.fedoraproject.org/OLS/Reprints-2003/LinuxSymposium2003.pdf#page=380

[15] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn, "Ceph: A scalable, high-performance distributed file system," in *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, ser. OSDI '06. Berkeley, CA, USA: USENIX Association, 2006, pp. 307–320. [Online]. Available: http://dl.acm.org/citation.cfm?id=1298455.1298485

[16] F. B. Schmuck and R. L. Haskin, "GPFS: A shared-disk file system for large computing clusters," in *FAST*, vol. 2, 2002, p. 19. [Online]. Available: http://www.cse.buffalo.edu/faculty/tkosar/cse710_spring14/papers/gpfs.pdf

[17] *ISO/IEC 7498-1:1994–Information technology–Open Systems Interconnection–Basic Reference Model: The Basic Model*, Internation Organization for Standardization Std. 7498-1, 1994.

[18] *ISO/IEC/IEEE International Standard–Information technology–Local and metropolitan area networks–Part 5: Token ring access method and physical layer specifications*, Std. 802-5:1992, June 1992.

[19] *IEEE Standard for Ethernet*, IEEE Std. 802.3-2015, March 2016.

[20] B. L. Worthington, G. R. Ganger, and Y. N. Patt, "Scheduling algorithms for modern disk drives," in *ACM SIGMETRICS Performance Evaluation Review*, vol. 22, no. 1. ACM, 1994, pp. 241–251.

[21] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W.-K. Su, "Myrinet: A gigabit-per-second local area network," *IEEE micro*, vol. 15, no. 1, pp. 29–36, 1995.

[22] *InfiniBand Architecture Specification Volume 1*, InfiniBand Trade Association Std., Rev. 1.3, Mar. 2015.

[23] corbet. (2005, 22 Aug.) Linux and TCP offload engines. Linux Weekly News. [Online]. Available: https://lwn.net/Articles/148697/

[24] P. MacArthur, Q. Liu, R. D. Russell, F. Mizero, M. Veeraraghavan, and J. M. Dennis, "An integrated tutorial on InfiniBand, verbs, and MPI," *IEEE Communications Surveys Tutorials*, vol. 19, no. 4, pp. 2894–2926, Fourthquarter 2017.

[25] *Supplement to Infiniband Architecture Specification Volume 1, Release 1.2.1: Annex A17: RoCEv2*, Infiniband Trade Association Annex A16, Sep. 2014.

[26] R. Recio, B. Metzler, P. Culley, J. Hilland, and D. Garcia, "A Remote Direct Memory Access Protocol Specification," Internet Request for Comments (Proposed Standard), RFC Editor, RFC 5040, Oct. 2007, updated by RFC 7146. [Online]. Available: http://www.rfc-editor.org/rfc/rfc5040.txt

[27] H. Shah, J. Pinkerton, R. Recio, and P. Culley, "Direct Data Placement over Reliable Transports," Internet Request for Comments (Proposed Standard), RFC Editor, RFC 5041, Oct. 2007, updated by RFC 7146. [Online]. Available: http://www.rfc-editor.org/rfc/rfc5041.txt

[28] P. Culley, U. Elzur, R. Recio, S. Bailey, and J. Carrier, "Marker PDU Aligned Framing for TCP Specification," Internet Request for Comments (Proposed Standard),

RFC Editor, RFC 5044, Oct. 2007, updated by RFCs 6581, 7146. [Online]. Available: http://www.rfc-editor.org/rfc/rfc5044.txt

[29] (2009, Mar) SoftiWARP. [Online]. Available: http://downloads.openfabrics.org/Media/Sonoma2009/Sonoma_2009_Mon_softiwrp.pdf

[30] (2017, May) SoftRoCE Update. [Online]. Available: https://openfabrics.org/images/eventpresos/workshops2015/DevWorkshop/Tuesday/tuesday_14.pdf

[31] S. Mittal and J. S. Vetter, "A survey of software techniques for using non-volatile memories for storage and main memory systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 27, no. 5, pp. 1537–1550, May 2016, iD: 1. [Online]. Available: http://ieeexplore.ieee.org/document/7120149/

[32] *NVM Express*, NVM Express, Inc. Std., Rev. 1.3a, 2017.

[33] *NVM Express over Fabrics*, NVM Express, Inc. Std., Rev. 1.0, Jun 2016.

[34] R. Dennard, "Field-effect transitor memory," U.S. Patent 3 387 286, Jun., 1968.

[35] *Portable Operating System Interface (POSIX) Base Specifications, Issue 7*, IEEE Std. 1003.1-2016, Sep. 2009.

[36] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee, "Better I/O through byte-addressable, persistent memory," in *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, ser. SOSP '09. New York, NY, USA: ACM, 2009, pp. 133–146. [Online]. Available: http://doi.acm.org/10.1145/1629575.1629589

[37] E. Zamanian, C. Binnig, T. Harris, and T. Kraska, "The end of a myth: distributed transactions can scale," *Proc. VLDB Endow.*, vol. 10, no. 6, pp. 685–696, 2017. [Online]. Available: https://doi.org/10.14778/3055330.3055335

[38] B. H. Tay and A. L. Ananda, "A survey of remote procedure calls," Ph.D. dissertation, New York, NY, USA, Jul. 1990. [Online]. Available: http://doi.acm.org/10.1145/382244.382832

[39] H. Shah, F. Marti, W. Noureddine, A. Eiriksson, and R. Sharp, "Remote Direct Memory Access (RDMA) Protocol Extensions," Internet Request for Comments (Proposed Standard), RFC Editor, RFC 7306, Jun. 2014. [Online]. Available: http://www.rfc-editor.org/rfc/rfc7306.txt

[40] J. Hilland, P. Culley, J. Pinkerton, and R. Recio, "RDMA Protocol Verbs Specification," Internet Draft, IETF Secretariat, Internet Draft draft-hilland-rddp-verbs-00, Apr 2003. [Online]. Available: http://www.rdmaconsortium.org/home/draft-hilland-iwarp-verbs-v1.0-RDMAC.pdf

[41] (2017, Apr) DiSNI: Distributed Storage and Networking Interface. [Online]. Available: https://github.com/zrlio/disni

[42] J. Postel, "User Datagram Protocol," Internet Request for Comments (INTERNET STANDARD), RFC Editor, RFC 768, Aug. 1980. [Online]. Available: http://www.rfc-editor.org/rfc/rfc768.txt

[43] ——, "Internet Protocol," Internet Request for Comments (INTERNET STANDARD), RFC Editor, RFC 791, Sep. 1981, updated by RFCs 1349, 2474, 6864. [Online]. Available: http://www.rfc-editor.org/rfc/rfc791.txt

[44] S. Deering and R. Hinden, "Internet Protocol, Version 6 (IPv6) Specification," Internet Request for Comments (INTERNET STANDARD), RFC Editor, RFC 8200, Jul. 2017. [Online]. Available: http://www.rfc-editor.org/rfc/rfc8200.txt

[45] P. Grun, *Introduction to InfiniBand for End Users*. InfiniBand Trade Association, 2010.

[46] J. Postel, "Transmission Control Protocol," Internet Request for Comments (INTERNET STANDARD), RFC Editor, RFC 793, Sep. 1981, updated by RFCs 1122, 3168, 6093, 6528. [Online]. Available: http://www.rfc-editor.org/rfc/rfc793.txt

[47] C. Bestler and R. Stewart, "Stream Control Transmission Protocol (SCTP) Direct Data Placement (DDP) Adaptation," Internet Request for Comments (Proposed Standard), RFC Editor, RFC 5043, Oct. 2007, updated by RFCs 6581, 7146. [Online]. Available: http://www.rfc-editor.org/rfc/rfc5043.txt

[48] R. Stewart, "Stream Control Transmission Protocol," Internet Request for Comments (Proposed Standard), RFC Editor, RFC 4960, Sep. 2007, updated by RFCs 6096, 6335, 7053. [Online]. Available: http://www.rfc-editor.org/rfc/rfc4960.txt

[49] P. MacArthur, "Userspace RDMA verbs on commodity hardware using DPDK," in *2017 IEEE 25th Annual Symposium on High-Performance Interconnects (HOTI)*, Aug. 2017, pp. 103–110.

[50] S. Venkataraman, N. Tolia, P. Ranganathan, and R. H. Campbell, "Consistent and durable data structures for non-volatile byte-addressable memory." in *FAST*, vol. 11, 2011, pp. 61–75.

[51] M. K. Aguilera, W. Golab, and M. A. Shah, "A practical scalable distributed B-tree," *Proc. VLDB Endow.*, vol. 1, no. 1, p. 598609, August 2008. [Online]. Available: http://dx.doi.org/10.14778/1453856.1453922

[52] M. K. Aguilera, A. Merchant, M. Shah, A. Veitch, and C. Karamanolis, "Sinfonia: a new paradigm for building scalable distributed systems," in *ACM SIGOPS Operating Systems Review*, vol. 41, no. 6. ACM, 2007, pp. 159–174.

[53] B. Sowell, W. Golab, and M. A. Shah, "Minuet: A scalable distributed multiversion B-tree," *Proc. VLDB Endow.*, vol. 5, no. 9, p. 884895, May 2012. [Online]. Available: http://dx.doi.org/10.14778/2311906.2311915

[54] C. Mitchell, K. Montgomery, L. Nelson, S. Sen, and J. Li, "Balancing CPU and network in the Cell distributed B-tree store," in *2016 USENIX Annual Technical Conference*, 2016, p. 451.

[55] Y. Sagiv, "Concurrent operations on B*-trees with overtaking," *Journal of computer and system sciences*, vol. 33, no. 2, pp. 275–296, 1986.

[56] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky, "MICA: A holistic approach to fast in-memory key-value storage," in *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. Seattle, WA: USENIX Association, Apr. 2014, pp. 429–444. [Online]. Available: https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/lim

[57] A. Trivedi, N. Ioannou, B. Metzler, P. Stuedi, J. Pfefferle, I. Koltsidas, K. Kourtis, and T. R. Gross, "Flashnet: Flash/network stack co-design," in *Proceedings of the 10th ACM International Systems and Storage Conference*, ser. SYSTOR '17. New York, NY, USA: ACM, 2017, pp. 15:1–15:14. [Online]. Available: http://doi.acm.org/10.1145/3078468.3078477

[58] T. Talpey, "Rdma durable write commit," Internet Draft (Work in Progress), RFC Editor, Internet Draft draft-talpey-rdma-commit-00, Feb. 2016. [Online]. Available: https://tools.ietf.org/html/draft-talpey-rdma-commit-00

[59] P. Stuedi, A. Trivedi, J. Pfefferle, R. Stoica, B. Metzler, N. Ioannou, and I. Koltsidas, "Crail: A high-performance I/O architecture for distributed data processing," *IEEE Data Eng. Bull.*, vol. 40, no. 1, pp. 38–49, 2017. [Online]. Available: http://sites.computer.org/debull/A17mar/A17MAR-CD.pdf#page=40

[60] (2017, Apr) Apache Spark: Lightning-fast cluster computing. [Online]. Available: http://spark.apache.org/

[61] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. Swanson, "NV-Heaps: Making persistent objects fast and safe with next-generation, non-volatile memories." New York, NY, USA: ACM, 2011, p. 105118. [Online]. Available: http://doi.acm.org/10.1145/1950365.1950380

[62] (2019, May) Intel 64 and ia-32 architectures software developer manuals. [Online]. Available: https://software.intel.com/en-us/articles/intel-sdm

[63] X. Ouyang, D. Nellans, R. Wipfel, D. Flynn, and D. K. Panda, "Beyond block I/O: Rethinking traditional storage primitives," in *2011 IEEE 17th International Symposium on High Performance Computer Architecture*, 2011, pp. 301–311, iD: 1.

[64] H. Volos, A. J. Tack, and M. M. Swift, "Mnemosyne: Lightweight persistent memory," in *Architectural Support for Programming Languages and Operating Systems*. New York, NY, USA: ACM, Mar. 2011, p. 91104. [Online]. Available: http://doi.acm.org/10.1145/1950365.1950379

[65] M. Liu, M. Zhang, K. Chen, X. Qian, Y. Wu, W. Zheng, and J. Ren, "DudeTM: Building durable transactions with decoupling for persistent memory," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2017, pp. 329–343.

[66] T. Haerder and A. Reuter, "Principles of transaction-oriented database recovery," *ACM Comput. Surv.*, vol. 15, no. 4, pp. 287–317, Dec. 1983. [Online]. Available: http://doi.acm.org/10.1145/289.291

[67] "Sparse, the semantic parser," Apr 2014. [Online]. Available: https://sparse.wiki.kernel.org

[68] H. Wan, Y. Lu, Y. Xu, and J. Shu, "Empirical study of redo and undo logging in persistent memory," in *2016 5th Non-Volatile Memory Systems and Applications Symposium (NVMSA)*, 2016, pp. 1–6, iD: 1.

[69] A. Dragojević, D. Narayanan, O. Hodson, and M. Castro, "Farm: Fast remote memory," in *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, 2014, pp. 401–414.

[70] A. Dragojević, D. Narayanan, E. B. Nightingale, M. Renzelmann, A. Shamis, A. Badam, and M. Castro, "No compromises: Distributed transactions with consistency, availability, and performance," in *Proceedings of the 25th symposium on operating systems principles*, 2015, pp. 54–70. [Online]. Available: http://doi.acm.org/10.1145/2815400.2815425

[71] M. Raynal, "About logical clocks for distributed systems," *SIGOPS Oper. Syst. Rev.*, vol. 26, no. 1, pp. 41–48, Jan. 1992. [Online]. Available: http://doi.acm.org/10.1145/130704.130708

[72] J. Squyres. (2014, Sep.) HPC over UDP. Cisco Systems. [Online]. Available: http://blogs.cisco.org/performance/hpc-over-udp

[73] A. Kagi, D. Burger, and J. R. Goodman, "Efficient synchronization: Let them eat qolb /sup1/," in *Conference Proceedings. The 24th Annual International Symposium on Computer Architecture*, June 1997, pp. 170–180.

[74] S. Miano, M. Bertrone, F. Risso, M. Tumolo, and M. V. Bernal, "Creating complex network service with eBPF: Experience and lessons learned," *High Performance Switching and Routing (HPSR). IEEE*, 2018.

[75] (2019, Apr.) Memcached: High-performance distributed memory object caching system. [Online]. Available: https://memcached.org

[76] S. McCanne and V. Jacobson, "The BSD packet filter: A new architecture for user-level packet capture." in *USENIX winter*, vol. 46, 1993.

[77] P. L. Lehman and s. B. Yao, "Efficient locking for concurrent operations on b-trees," *ACM Trans. Database Syst.*, vol. 6, no. 4, pp. 650–670, Dec. 1981. [Online]. Available: http://doi.acm.org/10.1145/319628.319663

[78] D. Buntinas, A. Saify, D. K. Panda, and J. Nieplocha, "Optimizing synchronization operations for remote memory communication systems," in *Proceedings International Parallel and Distributed Processing Symposium*, April 2003, pp. 8 pp.–.

[79] T. Talpey, "Remote Persistent Memory Access: workload scenarios and RDMA semantics," presented at the 13th Annual OpenFabrics Alliance Workshop, Mar. 2017.

Appendix A

RAW URDMA PERFORMANCE NUMBERS

We present the raw urdma performance numbers, in terms of means and standard deviation.

	urdma		softiwarp		HCA	
Message size	mean	stdev	mean	stdev	mean	stdev
2	5.853	0.038	10.227	0.012	2.370	0.294
4	5.943	0.078	10.227	0.035	2.197	0.006
8	5.923	0.047	10.223	0.023	2.193	0.006
16	5.987	0.081	10.223	0.025	2.400	0.294
32	5.983	0.040	10.250	0.010	2.613	0.006
64	5.953	0.061	10.360	0.044	2.803	0.274
128	5.983	0.006	10.410	0.050	2.747	0.006
256	6.043	0.067	10.580	0.044	3.087	0.006
512	6.173	0.031	10.723	0.006	3.380	0.294
1024	6.367	0.032	10.937	0.025	3.510	0.000
2048	6.977	0.050	11.767	0.071	4.077	0.006
4096	8.507	0.031	13.123	0.076	5.233	0.015
8192	11.550	0.085	16.203	0.067	7.783	0.006
16384	14.707	0.031	17.583	0.091	9.597	0.323
32768	21.133	0.015	23.917	0.159	13.187	0.329
65536	37.717	0.064	38.647	0.148	19.647	0.046

Table A-1: Mean and standard deviation (stdev) for RDMA WRITE perftest latency microbenchmark. All measurements are in microseconds.

	urdma		softiwarp		HCA	
Message size	mean	stdev	mean	stdev	mean	stdev
2	10.607	0.051	20.247	0.090	4.713	0.681
4	10.587	0.042	20.163	0.127	4.720	0.684
8	10.510	0.035	20.190	0.053	5.110	0.676
16	10.643	0.031	20.243	0.121	4.330	0.000
32	10.597	0.064	20.210	0.087	4.737	0.687
64	10.723	0.067	20.440	0.061	4.357	0.012
128	10.690	0.079	20.460	0.053	4.407	0.015
256	10.780	0.046	20.690	0.036	4.887	0.687
512	11.027	0.074	20.780	0.036	4.657	0.006
1024	11.263	0.061	20.963	0.083	4.950	0.035
2048	11.907	0.078	21.823	0.051	5.487	0.006
4096	13.847	0.065	23.170	0.120	6.637	0.006
8192	17.507	0.060	26.267	0.046	9.147	0.006
16384	22.090	0.148	31.353	0.465	11.203	0.681
32768	33.697	0.245	33.320	0.046	14.527	0.739
65536	49.693	0.072	38.847	0.006	20.910	0.010

Table A-2: Mean and standard deviation (stdev) for RDMA READ perftest latency microbenchmark. All measurements are in microseconds.

	urdma		softiwarp		HCA	
Message size	mean	stdev	mean	stdev	mean	stdev
2	6.180	0.035	10.617	0.012	2.620	0.000
4	6.190	0.044	10.590	0.066	2.990	0.320
8	6.157	0.047	10.570	0.035	2.623	0.006
16	6.190	0.061	10.617	0.040	2.640	0.000
32	6.213	0.040	10.583	0.076	2.983	0.006
64	6.290	0.030	10.643	0.025	3.240	0.364
128	6.287	0.040	10.747	0.058	3.330	0.364
256	6.320	0.050	10.950	0.010	3.480	0.000
512	6.517	0.093	11.080	0.036	3.627	0.006
1024	6.947	0.060	11.300	0.020	4.347	0.387
2048	7.820	0.035	12.090	0.095	4.503	0.040
4096	9.587	0.075	13.563	0.055	6.057	0.387
8192	12.640	0.053	16.613	0.074	8.453	0.401
16384	16.847	0.135	17.967	0.116	9.933	0.387
32768	25.123	0.172	23.967	0.125	13.067	0.012
65536	41.530	0.070	39.147	0.453	19.867	0.006

Table A-3: Mean and standard deviation (stdev) for SEND perftest latency microbenchmark. All measurements are in microseconds.

	urdma		softiwarp		HCA	
Message size	mean	stdev	mean	stdev	mean	stdev
1024	8970.987	48.237	7109.813	167.427	32520.373	98.282
2048	16279.867	611.071	13676.933	3355.682	36400.640	0.349
4096	24040.667	3389.326	20145.413	4194.434	37251.440	3.683
8192	30858.507	2073.908	25618.800	3360.555	37692.320	0.000
16384	35548.533	2127.313	29075.040	1505.900	37687.920	4.205
32768	36228.213	1531.369	34500.267	5326.010	37685.280	1.386
65536	36115.360	1639.440	28150.107	238.409	37672.987	3.587
131072	36164.960	1529.747	31166.187	5572.835	37698.133	1.775
262144	36098.960	1473.171	28367.040	321.588	37698.027	1.589
524288	35998.533	1562.004	31333.493	5430.856	37597.680	1.257
1048576	36798.880	0.080	33333.600	4636.096	37596.320	0.080
2097152	35998.720	0.277	34400.267	5542.655	37596.960	1.386
4194304	33599.120	1599.640	30933.547	5143.434	36796.293	3.857
8688608	29828.400	0.416	29829.867	5740.755	36456.560	1.250

Table A-4: Mean and standard deviation (stdev) for RDMA WRITE perftest throughput microbenchmarks. All measurements are in Megabits per second.

	urdma		softiwarp		HCA	
Message size	mean	stdev	mean	stdev	mean	stdev
1024	7158.560	48.194	4790.213	79.895	20624.080	6276.676
2048	13599.547	105.665	8480.293	33.939	36381.440	0.684
4096	24768.987	144.691	13425.547	406.026	37250.693	1.256
8192	35360.347	152.202	18463.573	203.313	37691.600	1.317
16384	37153.120	47.440	23441.227	675.083	37690.773	1.271
32768	37098.827	43.325	27286.693	360.403	37685.307	1.339
65536	37165.733	51.548	28066.507	505.122	37673.520	0.080
131072	37132.320	76.306	27299.067	1284.272	37697.253	1.690
262144	36731.973	251.773	29232.933	610.277	37697.227	1.709
524288	36665.467	115.678	28133.467	3402.084	37597.680	1.257
1048576	35465.707	610.932	32933.467	2202.864	37598.880	0.560
2097152	30932.453	4618.664	35200.053	800.080	37597.787	1.409
4194304	29332.453	923.691	36266.800	923.876	36796.933	1.201
8688608	23201.173	0.333	36458.907	0.201	36456.933	1.746

Table A-5: Mean and standard deviation (stdev) for RDMA READ perftest throughput microbenchmarks. All measurements are in Megabits per second.

	urdma		softiwarp		HCA	
Message size	mean	stdev	mean	stdev	mean	stdev
1024	7504.080	92.775	6616.427	528.587	30662.293	8.412
2048	13377.653	434.956	11391.493	237.510	36329.840	3.039
4096	22715.253	126.339	16868.160	822.590	37213.467	1.157
8192	23712.640	262.604	22176.240	643.883	37674.293	1.414
16384	26115.680	427.561	27885.387	1264.388	37672.827	1.342
32768	28127.973	416.730	27545.920	670.129	37672.000	1.388
65536	26098.987	24.640	27583.440	2320.146	37697.333	45.181
131072	27748.533	85.725	31533.440	3284.211	37699.253	1.342
262144	28165.600	723.438	36066.587	57.527	37696.987	1.296
524288	27598.373	200.520	32733.413	3900.429	37597.760	1.250
1048576	26932.453	230.940	35200.107	3124.115	37595.440	1.317
2097152	25865.573	461.996	36800.107	0.244	37596.960	2.634
4194304	23999.093	0.514	33600.213	2771.120	36797.040	1.388
8688608	19884.987	0.987	32039.707	1913.524	36455.867	1.296

Table A-6: Mean and standard deviation (stdev) for SEND perftest throughput microbenchmarks. All measurements are in Megabits per second.