CHAPTER 18

Remote Persistent Memory

This chapter provides an overview of how persistent memory - and the programming concepts that were introduced in this book - can be used to access persistent memory contained in remote servers connected via a network. A combination of TCP/IP or RDMA network hardware and software running on the servers containing the persistent memory hardware and software provide direct remote access to persistent memory.

Having direct memory access via a high-performance network connection is a critical use case for most cloud deployments of persistent memory. Typically, in high-availability or highly redundant use cases, data written locally to persistent memory is not considered reliable until it has been replicated to two or more remote persistent memory devices on separate remote servers. We describe this push model design later in this chapter.

While it is certainly possible to use existing TCP/IP networking infrastructure to remotely access the persistent memory, this chapter focuses on the use of remote direct memory access (RDMA). Direct memory access (DMA) allows data movement on a platform to be offloaded to a hardware DMA engine that moves that data on behalf of the CPU, freeing it to do other important tasks during the data move. RDMA applies the same concept and enables data movement between remote servers to occur without the CPU on either server having to be directly involved

This chapter's content, and the PMDK `librpmem` remote persistent memory library that is discussed, assume the use of RDMA, but the concepts discussed here can apply to other networking interconnects and protocols.

Figure 18-1 outlines a simple remote persistent memory configuration with one initiator system that is replicating writes to persistent memory on a single remote target system. While this shows the use of persistent memory on both the initiator and target, it is possible to read data from initiator DRAM and write to persistent memory on the remote target system, or read from the initiator's persistent memory and write to the remote target's DRAM.

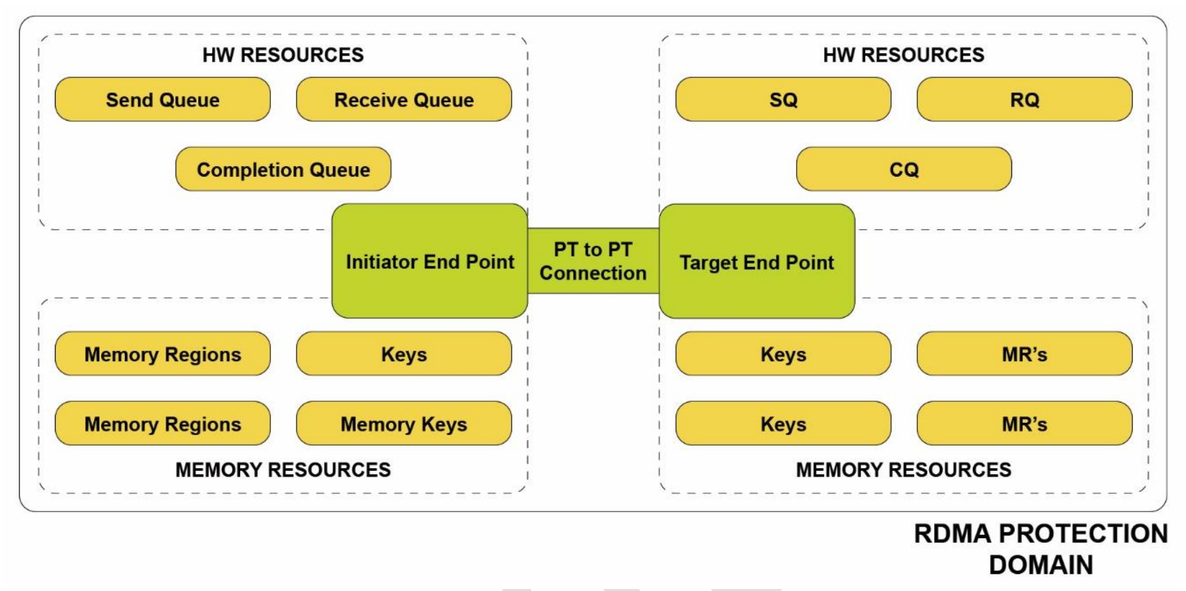

Figure 18-1. Initiator and target system using RDMA.

RDMA Networking Protocols

Examples of popular RDMA networking protocols used throughout the cloud and enterprise data centers include:

- InfiniBand is an I/O architecture and high-performance specification for data transmission between high-speed, low latency and highly scalable CPUs, processors, and storage.
- RoCE (RDMA over Converged Ethernet) is a network protocol that allows RDMA over an ethernet network.
- iWARP (Internet Wide Area RDMA Protocol) is a computer networking protocol that implements RDMA for efficient data transfer over Internet Protocol networks.

All three protocols support high-performance data movement to and from persistent memory using RDMA.

The RDMA protocols are governed by the RDMA Wire Protocol Standards, which are driven by the IBTA (InfiniBand Trade Association) and the IEFT (Internet Engineering Task Force) specifications. The IBTA (https://www.infinibandta.org/) governs the Infiniband and RoCE protocols while the IEFT (https://www.ietf.org/) governs iWarp.

Low-latency RDMA networking protocols allow the network interface controller (NIC) to control the movement of data between an initiator node source buffer and the sink buffer on the target node without needing either node's CPU to be involved in the data movement. In fact, RDMA read and RDMA write operations are often referred to as one-sided operations because all of the information required to move the data is supplied by the initiator, and the CPU on the target node is not typically interrupted or even aware of the data transfer.

To do this, the knowledge of the target node's buffers must be passed to the initiator before the remote operations will begin. This requires configuring the local initiator RDMA resources and buffers. Similarly, the remote target node's RDMA resources that will require CPU resources will need to be initialized and reported to the initiator. However, once the resources for the RDMA transfers are set up, and applications initiate the RDMA request using the CPU, the NIC does the actual data movement on behalf of the RDMA-aware application.

RDMA-aware applications are responsible for:

- Interrogating each NIC on every initiator and target system to determine supported features.
- Selecting a NIC for each end of the RDMA point-to-point connection.
- Creating the connection with the selected NICs, described as an RDMA protection domain.
- Allocating queues for the incoming and outgoing message on each NIC and assigning those hardware resources to the protection domain.
- Allocating DRAM or persistent memory buffers for use with RDMA, registering those buffers with the NIC, and assigning those buffers to the protection domain.

Three basic RDMA commands are used by most RDMA-capable applications and libraries:

RDMA Write. A one-sided operation where only the initiator supplies all of the information required for the transfer to occur. This transfer is used to write data to the remote target node. The write request contains all source and sink buffer information. The remote target system is not typically interrupted and thus completely unaware of the write operations occurring through the NIC. When the initiator's NIC sends a write to the target, it will generate a "software write completion interrupt." A *software write completion interrupt* means the write message has been sent to the target NIC and is not an indicator of the write completion. Optionally, RDMA writes can use an immediate option that will interrupt the target node CPU and allow software running there to be immediately notified of the write completion.

RDMA Read. A one-sided operation where only the initiator supplies all of the information required for the transfer to occur. This transfer is used to read data from the remote target node. The read request contains all source buffer and target sink buffer information, and the remote target system is not typically interrupted and thus completely unaware of the read operations occurring through the NIC. The initiator *software read completion interrupt* is an acknowledgement that the read has traversed all the way through the initiator's NIC, over the network, into the target system's NIC, through the target internal hardware mesh and memory controllers, to the DRAM or persistent memory to retrieve the data then it returns it all the way back to the initiator software that registered for the completion notification.

RDMA Send (and Receive). The two-sided RDMA Send means that both the initiator and target must supply information for the transfer to complete. This is because the target NIC will be interrupted when the RDMA Send is received by the target NIC and requires a hardware receive queue to be set up and pre-populated with completion entries before the NIC will receive an RDMA Send transfer operation. Data from the initiator application is bundled in a small, limited sized buffer and sent to the target NIC. The target CPU will be interrupted to handle the send operation and any data it contains. If the initiator needs to be notified of receipt of the RDMA Send, or to handle a message back to the initiator, another RDMA Send operation must be sent in the reverse direction after the initiator has set up its own receive queue and queued completion entries to it. The use of the RDMA Send command and the contents of the payload are application-specific implementation details. An RDMA Send is typically used for bookkeeping and updates of read and write activity between the initiator and the target, since the target application has no other context of what data movement has taken place. For example, because there is no good way to know when writes have completed on the target, an RDMA Send is often used to notify the target node what is happening. For small amounts of data, the RDMA Send is very efficient but it always requires target-side interaction to complete. An RDMA Write with immediate data operation will also allow the target node to be interrupted when the write has completed as a different mechanism for bookkeeping.

Goals of the Initial Remote Persistent Memory Architecture

The goal of the first remote persistent memory implementation was based on minimal changes—or ideally, no changes—to the current RDMA hardware and software stacks used with volatile memory. From a network hardware, middleware, and software architecture standpoint, writing to remote volatile memory is identical to writing to remote persistent memory. The knowledge that a specific memory-mapped file is backed by persistent memory versus volatile memory is entirely the responsibility of the application to maintain. None of the lower layers in the networking stack are aware of the fact that the write is to a persistent memory region or volatile memory. The responsibility of knowing which write persistence method to use for a given target connection, and making those remote writes persistent, falls to the application.

Guaranteeing Remote Persistence

Until this chapter, much of the book focuses on the use and programming of persistent memory on the local machine. You are now aware of some of the challenges of using persistent memory, the persistence domain, and the need to understand and use a flushing mechanism to ensure the data is persistent. These same programming concepts and challenges apply to remote persistent memory with the additional constraints of making it work within the existing network protocol and network latency.

The SNIA NVM programming model (described in Chapter 3) requires applications to flush data that has been written to persistent memory to guarantee the written data made it into the persistence domain. This same requirement applies to writing to remote persistent memory. After the RDMA Write or Send operation has moved the data from the initiator node to the persistent memory on the target node, that write or send data needs to be flushed to the persistence domain on the remote system. Alternatively, the remote write or send data needs to bypass CPU caches on the remote node to avoid having to be flushed.

Different vendor-specific platform features add an extra challenge to RDMA and to remote persistent memory. Intel® platforms typically use a feature called allocating-writes or Direct Data IO (DDIO) which allows incoming writes to be placed directly into the CPU's L3 cache. The data is immediately visible to any application wanting to read the data. However, having allocating-writes enabled means RDMA

Writes to persistent memory now have to be flushed to the persistence domain on the target node.

On Intel platforms, allocating writes can be disabled by turning on non-allocating write IO flows which forces the write data to bypass cache and be placed directly into the persistent memory, governed by the location of the RDMA Write sink buffer. This would slow down applications that will immediately touch the newly written data because they incur the penalty to pull the data into CPU cache. However, this simplifies making remote writes to persistent memory simpler and faster because cache flushing on the remote target node can be avoided. An additional complication to using non-allocating write mode on an Intel platform is that an entire PCI root complex must be enabled for this write mode. This means that any inbound writes that come through that PCI root complex, for any device connected downstream of it, will have write-data bypass CPU caches, causing possible additional performance latency as a side effect.

Intel specifies two methods for forcing writes to remote persistent memory into the persistence domain:

1. A general-purpose remote replication method that does not rely on Intel non-allocating write mode and assumes some or all of the remote write data will end up in CPU cache on the target system.
2. A high-performance appliance remote replication method that uses the Intel platform-specific non-allocating write mode and is probably more suited to an appliance product where there is complete control over the hardware configuration to control what is connected to which PCI root complex.

General-Purpose Remote Replication Method

The general-purpose remote replication method (GPRRM), also referred to as the general-purpose server persistency method (GPSPM), relies on the initiator RDMA application to maintain a list of virtual addresses on the remote target system that have been written to with previous RDMA Write requests. When all remote writes to persistent memory are issued, the application issues an RDMA Send request from the initiator NIC to the target NIC. The RDMA Send request contains a list of virtual starting addresses and lengths that the target system will consume when the application software running on the target node interrupts the system to process the send request. The application walks the list of regions, flushing each cache line in the requested region to the persistent memory using an optimized flushmachine instruction (`CLWB`, `CLFLUSHOPT`, etc.). When complete, an `SFENCE` machine instruction is required to fence those previous writes and force them to complete before handling additional writes. The application on the target system then issues an RDMA Send request back to interrupt the initiator software of the completed flush operations. This is an indicator to the application that the previous writes were made persistent.

Figure 18-2 outlines the general-purpose remote replication method sequence of operation.

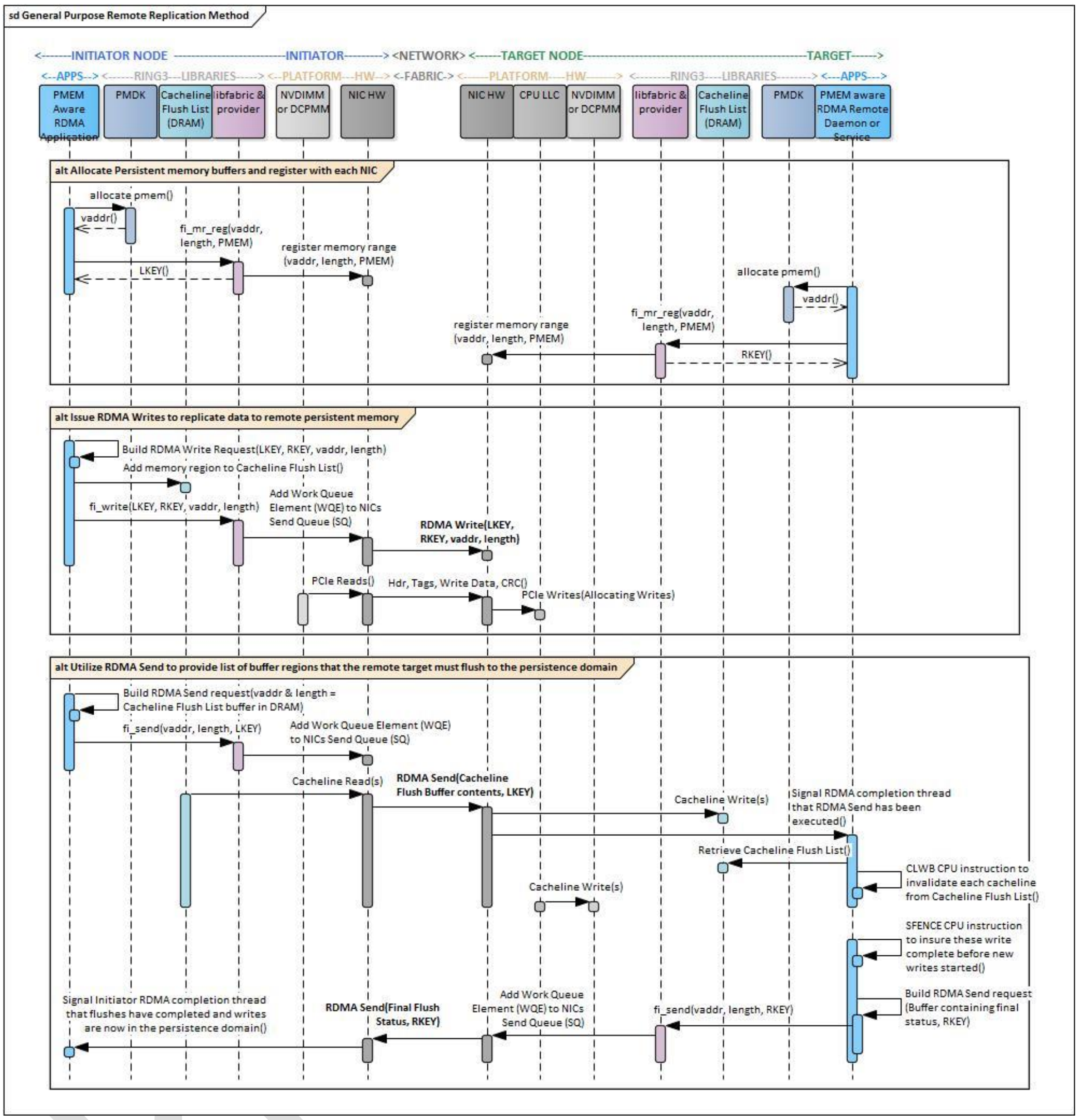
sd General Purpose Remote Replication Method
<--------INITIATOR NODE---------INITIATOR---------> <NETWORK> <-------TARGET NODE-----------TARGET------->
<--APPS--> <-------RING3----LIBRARIES-------> <--PLATFORM----HW---> <-FABRIC-> <--------PLATFORM----HW--------> <---------RING3----LIBRARIES---------> <---APPS--->
PMEM Aware RDMA Application
PMDK
Cacheline Flush List (DRAM)
libfabric & provider
NVDIMM or DCPMM
NIC HW
NIC HW
CPU LLC
NVDIMM or DCPMM
libfabric & provider
Cacheline Flush List (DRAM)
PMDK
PMEM aware RDMA Remote Daemon or Service
alt Allocate Persistent memory buffers and register with each NIC
allocate pmem()
vaddr()
fi_mr_reg(vaddr, length, PMEM)
register memory range (vaddr, length, PMEM)
LKEY()
allocate pmem()
vaddr()
fi_mr_reg(vaddr, length, PMEM)
register memory range (vaddr, length, PMEM)
RKEY()
alt Issue RDMA Writes to replicate data to remote persistent memory
Build RDMA Write Request(LKEY, RKEY, vaddr, length)
Add memory region to Cacheline Flush List()
fi_write(LKEY, RKEY, vaddr, length)
Add Work Queue Element (WQE) to NICs Send Queue (SQ)
RDMA Write(LKEY, RKEY, vaddr, length)
PCIe Reads()
Hdr, Tags, Write Data, CRC()
PCIe Writes(Allocating Writes)
alt Utilize RDMA Send to provide list of buffer regions that the remote target must flush to the persistence domain
Build RDMA Send request(vaddr & length = Cacheline Flush List buffer in DRAM)
fi_send(vaddr, length, LKEY)
Add Work Queue Element (WQE) to NICs Send Queue (SQ)
Cacheline Read(s)
RDMA Send(Cacheline Flush Buffer contents, LKEY)
Cacheline Write(s)
Signal RDMA completion thread that RDMA Send has been executed()
Retrieve Cacheline Flush List()
CLWB CPU instruction to invalidate each cacheline from Cacheline Flush List()
Cacheline Write(s)
SFENCE CPU instruction to insure these write complete before new writes started()
Build RDMA Send request (Buffer containing final status, RKEY)
Signal Initiator RDMA completion thread that flushes have completed and writes are now in the persistence domain()
RDMA Send(Final Flush Status, RKEY)
Add Work Queue Element (WQE) to NICs Send Queue (SQ)
fi_send(vaddr, length, RKEY)

Figure 18-2. The general-purpose remote replication method.

How Does the General-Purpose Remote Replication Method Make Data Persistent?

After the RDMA Write or any number of writes have been sent, the write data will either be in the L3 CPU cache (due to the default allocating-writes) or persistent memory (assuming it does not all fit in L3) with potentially some write data still pending in NIC internal buffers. An RDMA Send request, by definition, will force previous writes to be pushed out of the NIC to the target L3 CPU cache and interrupt the target CPU. At this point, all previously issued RDMA Writes to persistent memory are now in L3 or persistent memory. The RDMA Send request contains a list of cache lines that the initiator is requesting the target system to flush to its persistence domain. The target system issues `optimized flush` instructions to flush each cache line in the list to the persistence domain. This is followed by an `SFENCE` to guarantee these writes complete before new writes are handled. At this point, the previous writes that were flushed in the RDMA Send list are now persistent.

Performance Implications of the General-Purpose Remote Replication Method

The general-purpose remote replication method requires that RDMA of the initiator software follow a number of RDMA Write(s) with an RDMA Send. After the target NIC finishes flushing the requested regions, an RDMA Send from the target goes back to the initiator to affirm that the initiator application can consider those writes persistent. This additional send/receive/send/receive messaging has an effect on latency and throughput to make the writes persistent and has 50 percent higher latency than the appliance remote replication method. The extra messaging has an effect on overall bandwidth and scalability of all the RDMA connections running on those NICs.

Also, if size of the RDMA Write that needs to be made persistent is small, the efficiency of the connection drops dramatically as the extra messaging overhead becomes a significant component of the overall latency. Additionally, the target node CPU and caches are consumed for that operation. The same data is essentially transmitted twice: once from NIC (via PCIe) to the CPU L3 cache, then from the CPU L3 cache to the memory controller (iMC).

Appliance Remote Replication Method

Users of persistent memory on an Intel platform can use non-allocating write flows by enabling the feature on the specific PCI root complex where incoming writes from the NIC will enter into the CPU's internal fabric and out to the persistent memory. Using the non-allocating write flow, the incoming RDMA Writes will bypass CPU caches and go directly to the persistence domain. This means that writes do not need to be flushed to the persistence domain by the target system CPU.

The IO pipeline still needs to be flushed to the persistence domain. This is more efficiently accomplished by issuing a small RDMA Read to any memory address on the same RDMA connection as the RDMA Writes; the memory address does not need to be one that was written or is persistent. The RDMA specification clearly states that an RDMA Read will force the previous RDMA Writes to complete first. This ordering rule is also true of the PCIe interconnect to which the target NIC is connected. PCIe Reads will perform a pipeline flush and force previous PCIe writes to complete first.

Figure 18-3 outlines the basic appliance remote replication method, often referred to as the appliance remote replication method, described above.

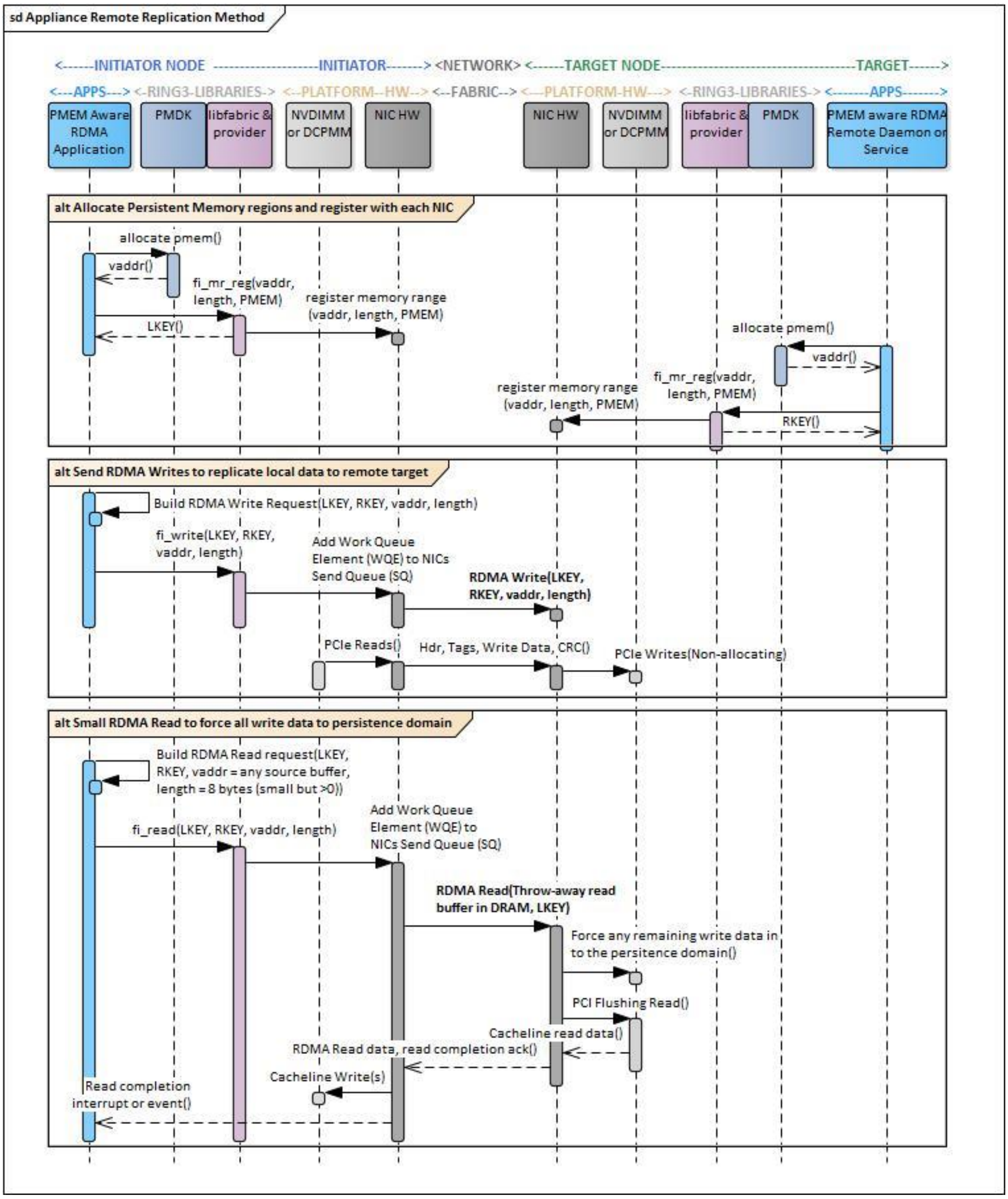
sd Appliance Remote Replication Method
<------INITIATOR NODE --------------------INITIATOR-------> <NETWORK> <------TARGET NODE----------------------------------TARGET------>
<---APPS---> <-RING3-LIBRARIES-> <--PLATFORM--HW--> <--FABRIC--> <---PLATFORM-HW---> <-RING3-LIBRARIES-> <--------APPS------->
PMEM Aware RDMA Application
PMDK
libfabric & provider
NVDIMM or DCPMM
NIC HW
NIC HW
NVDIMM or DCPMM
libfabric & provider
PMDK
PMEM aware RDMA Remote Daemon or Service
alt Allocate Persistent Memory regions and register with each NIC
allocate pmem()
vaddr()
fi_mr_reg(vaddr, length, PMEM)
register memory range (vaddr, length, PMEM)
LKEY()
allocate pmem()
vaddr()
fi_mr_reg(vaddr, length, PMEM)
register memory range (vaddr, length, PMEM)
RKEY()
alt Send RDMA Writes to replicate local data to remote target
Build RDMA Write Request(LKEY, RKEY, vaddr, length)
fi_write(LKEY, RKEY, vaddr, length)
Add Work Queue Element (WQE) to NICs Send Queue (SQ)
RDMA Write(LKEY, RKEY, vaddr, length)
PCIe Reads()
Hdr, Tags, Write Data, CRC()
PCIe Writes(Non-allocating)
alt Small RDMA Read to force all write data to persistence domain
Build RDMA Read request(LKEY, RKEY, vaddr = any source buffer, length = 8 bytes (small but >0))
fi_read(LKEY, RKEY, vaddr, length)
Add Work Queue Element (WQE) to NICs Send Queue (SQ)
RDMA Read(Throw-away read buffer in DRAM, LKEY)
Force any remaining write data in to the persitence domain()
PCI Flushing Read()
Cacheline read data()
RDMA Read data, read completion ack()
Cacheline Write(s)
Read completion interrupt or event()

Figure 18-3. The appliance remote replication method.

How Does the Appliance Remote Replication Method Make Data Persistent?

The combination of bypassing CPU caches on the target system for the inbound RDMA Writes to persistent memory with the ordering semantics of the RDMA and PCIe protocols results in an efficient mechanism to make data persistent. Since the RDMA Read to persistent memory will force previous writes first to persistent memory and the persistence domain, the RDMA Read completion that comes back after those writes are complete is the initiator application’s acknowledgment that those writes are now durable.

Chapter 2 defines the persistence domain in depth, including how the platform ensures that all writes get to the media from the persistence domain in the event of a power loss.

Performance Implications of the Appliance Remote Replication Method

This single extra round-trip using an RDMA Read is roughly 50 percent lower latency than the general-purpose server percistency method, which requires two round-trip messages before the writes can be declared durable. As with the first method, as the size of the writes to be made durable gets smaller, the RDMA Read round-trip overhead becomes a significant component of the overall latency.

General Software Architecture

The software stack for the use of remote persistent memory typically uses the same memory-mapped files discussed in Chapter 3. Persistent memory is presented to the RDMA application as a memory-mapped file. The application registers the persistent memory with the local NIC on both ends of the connection, and the resulting registry key is shared with the initiator application for use in the RDMA Read and Write requests. This is the identical process required for using traditional volatile DRAM with RDMA.

A layering of kernel and application-level software components are typically used to allow an application to make use of both persistent memory and an RDMA connection. The IBTA defines verbs interfaces that are typically implemented by the kernel drivers for the NIC and the middleware software application library. Additional libraries may be layered above the verbs layer to provide generic RDMA services via a common API- and NIC-specific provider that implements the library.

On Linux, the Open Fabric Alliance (OFA) libibverbs library provides ring-3 interfaces to configure and use the RDMA connection for NICs that support IB, RoCE, and iWarp RDMA network protocols. The OFA libfabric ring-3 application library can be layered on top of libibverbs to provide a generic high-level common API that can be used with typical RDMA NICs. This common API requires a provider plugin to implement the common API for the specific network protocol. The OFA website contains many example applications and performance tests that can be used on Linux with a variety of RDMA-capable NICs. Those examples provide the backbone of the PMDK `librpmem` library.

Windows implements remotely-mounted NTFS volumes via the ring-3 SMB Direct Application library, which provides a number of storage protocols including block storage over RDMA.

Figure 18-4 provides the basic high-level architecture for a typical RDMA application on Linux, using all of the publically available libraries and interfaces. Notice that a separate side-band connection is typically needed to set up the RDMA connections themselves.

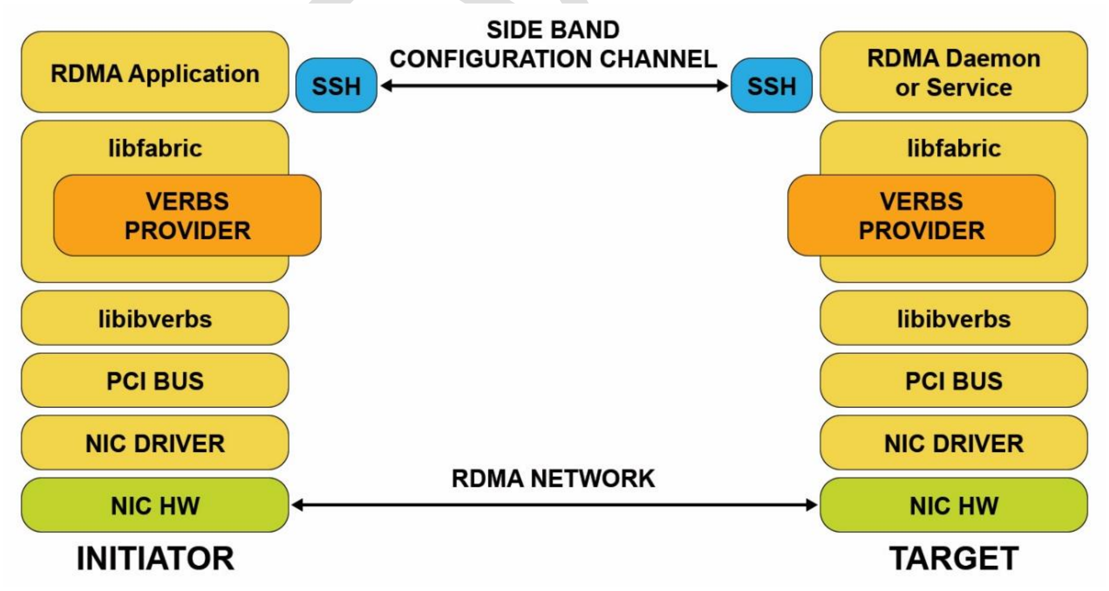

Figure 18-4. General RDMA software architecture.

librpmem Architecture and its use in Replication

PMDK implements both the general-purpose remote replication method and the appliance remote replication method in the librpmem library. As of PMDK v1.7, the librpmem library implements the synchronous and asynchronous replication of local writes to persistent memory on remote systems. librpmem is a low-level library, like libpmem, that allows other libraries to use its replication features.

libpmemobj uses a synchronous write model, meaning that the local initiator write and all of the remotely replicated writes must complete before the local write will be completed back to the application. The libpmemobj library also implements a simple active-passive replication architecture, where all persistent memory transactions are driven through the active initiator node and the remote targets passively standby, replicating the write data. While the passive target systems have the latest write data replicated, the implementation makes no attempt to fail-over, fail-back, or load balance using the remote systems. The following sections describe the significant performance drawbacks to this implementation.

libpmemobj uses the local memory-pool configuration information provided in a configuration file to describe the remote-network connected memory-mapped files. A remote rpmemd program installed on each remote target system is started and connected to the librpmem library on the initiator using a secure encrypted socket connection. Through this connection, librpmem, on behalf of libpmemobj, will set up the RDMA point-to-point connection with each target system, determine the persistence method the target supports (general purpose or appliance method), allocate remote memory-mapped persistent memory files, register the persistent memory on the remote NIC, and retrieve the resulting memory keys for the registered memory.

Once all the RDMA connections to all the targets are established, all required queues are instantiated, and memory buffers have all been allocated and registered, the libpmemobj library is ready to begin remotely replicating all application write data to its local memory-mapped file. When the application calls pmemobj_persist() in libpmemobj, the library will generate a corresponding rpmem_persist() call into librpmem which, in turn, calls the libfabric fi_write() to do the RDMA Write. librpmem then initiates the RDMA Read or Send persistence method (as governed by an understanding of the currently enabled target node's current configuration) by calling libfabric fi_read() or fi_send(). RDMA Read is used in the appliance remote replication method and RDMA Send is used in the general-purpose remote replication method.

Figure 18-5 outlines the high-level components and interfaces described above and used by both the initiator and remote target system using `librpmem` and `libpmemobj`.

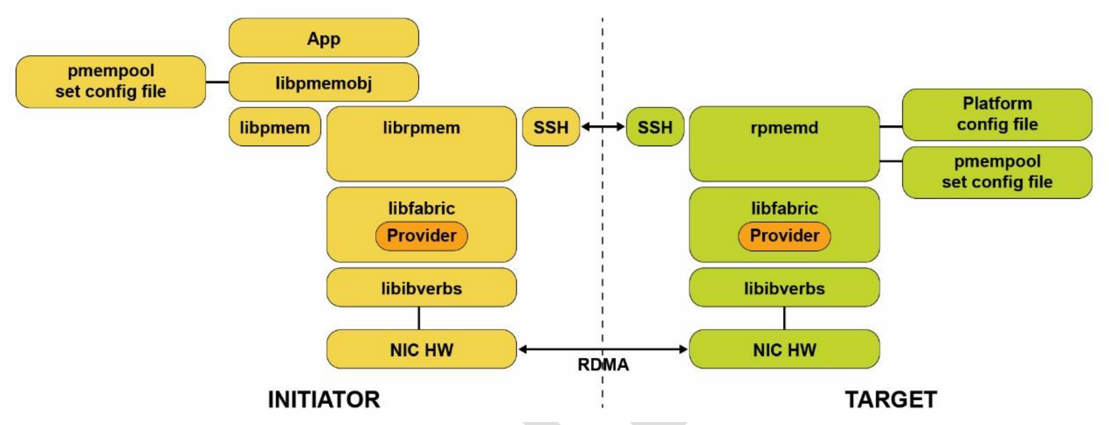

Figure 18-5. RDMA architecture using libpmemobj and librpmem.

The major components (shown in Figure 18-5) are described below to help you understand the high-level architecture that is used by the PMDK's remote replication feature:

librpmem - PMDK Remote RDMA Access Library. The container for the Initiator Node for all the initiator PMDK functionality that is related to remote replication using RDMA.

rpmemd - PMDK Remote RDMA Configuration Daemon. The container for the Target Node for all the target PMDK functionality that is related to remote replication using RDMA. It will block any local access to the pmempool set that has been configured for remote usage and executes the remote target interrupt handlers required for the general-purpose remote replication method.

Initiator and Target SSH. This component is used by both `librpmem` and `rpmemd` libraries to set up a simple socket connection, close a previously opened socket connection, and send communication packets back and forth.

Libfabric. The OFA defined high-level ring-3 application API for setting up and using a fabric connection in a fabric and vendor-agnostic way. This high-level interface supports RoCE, InfiniBand, and iWARP, as well Intel® Omni-Path Architecture products and other network protocols using libfabric-specific transport providers.

Libibverbs. The OFA defined high-level RDMA fabric-based interface. This high-level interface supports RoCE, InfiniBand, and iWARP and is commonly used in most Linux distributions.

Target Node Platform Configuration File. Simple text file generated by the IT admin or user to describe the platform capabilities of the remote Target Node. This file describes specific capabilities that affect what durability method can be used, i.e.: ADR-enabled platform, non-allocating write flows enabled by the NIC, and platform type. It also specifies the default socket-connection port that `rpmemd` will listen on.

Initiator Node PMDK pmempool Set Configuration File. An existing persistent-memory poolset configuration file is generated by the system or application administrator that describes local sets of files that will be treated as a pool of persistent memory on the local platform. It also describes local files for local replication and remote target host names for remote replication.

Target Node PMDK pmempool Set Configuration File. An existing persistent-memory poolset configuration file is generated by the system or application administrator that describes local sets of files that will be treated as a pool of persistent memory on the local platform. On the Target node, this set is the collection of files that the Initiator node is replicating data into.

Initiator and Target Node Operating System syslog. The standard Linux syslog on each node used by `librpmem` and `rpmemd` for outputting useful data for both debug and non-debug information. Since there is little information from `rpmemd` that is visible on the initiator system, extensive information will be output to the target system syslog when `rpmemd` is started with the `'-d'` (debug) runtime option. Even without the debug enabled, `rpmemd` will output socket events like open, close, create, lost connection, and similar RDMA events.

Configuring Remote Replication Using Poolsets

You are probably already familiar with using poolsets (introduced in Chapter 7) `libpmemobj` to initialize remote replication, which requires two such poolset files. The file used on the initiator side by the `libpmemobj`-enabled application must describe the local memory pool and point to poolset configuration file on the target node. Whereas the poolset file on the target node must describe the memory pool shared by the target system.

Listing 18-1 shows a poolset file that will allow replicating local writes to the “remotepool.set” on a remote host.

Listing 18-1.: poolwithremotereplica.set – An example of replicating local data to a remote host.

```
PMEMPOOLSET
256G /mnt/pmem0/pool1

REPLICA user@example.com remotepool.set
```

Listing 18-2 shows a poolset file that describes the memory-mapped files shared for the remote access. In many ways, a remote poolset file is the same as the regular poolset file, but it must fulfill additional requirements:

- Exist in a poolset directory specified in the `rpmemd` configuration file.
- Should be uniquely identified by its name, which an `rpmem`-enabled application has to use to replicate to the specified memory pool.
- Cannot define any additional replicas, local or remote.

Listing 18-2. remotereplica.set – An example of how to describe the memory pool on the remote host.

```
PMEMPOOLSET
256G /mnt/pmem1/pool2
```

Performance Considerations

Once persistent memory is accessible via a remote network connection, significantly lower latency can be achieved compared with writing to a remote SSD or legacy block storage device. This is because the RDMA hardware is writing the remote write data directly into the final persistent memory location, whereas remote replication to an SSD requires an RDMA Write into the DRAM on the remote server, followed by a second local DMA operation to move the remote write data from volatile DRAM into the final storage location on the SSD or other legacy block storage device.

The performance challenge with replicating data to remote persistent memory is that while large block sizes of 512KiB or larger can achieve good performance, as the size of the writes being replicated gets smaller, the network overhead becomes a larger portion of the total latency, and performance can suffer.

If the persistent memory is being used as an SSD replacement, the typical native block storage size is 4K, avoiding some of the inefficiencies seen with small transfers. If the persistent memory replaces a traditional SSD and data is written remotely to the SSD, the latency improvements with persistent memory can be 10x or more.

The synchronous replication model implemented in `librpmem` means that small data structures and pointer updates in local persistent memory result in small, very inefficient, RDMA Writes followed by a small RDMA Read or Send to make that small amount of write data persistent. This results in significant performance degradation compared to writing only to local persistent memory. It makes the replication performance very dependent on the local persistent memory write sequences, which is heavily dependent on the application workload. In general, the larger the average request size and the lower the number of `rpmem_persist()` calls that are required for a given workload will improve the overall latency required for guaranteeing that data is persistent.

It is possible to follow multiple RDMA Writes with single RDMA Read or Send to make all of the preceding writes persistent. This reduces the impact of the size of RDMA Writes on the overall performance of the proposed solution. But using this mitigation, remember you are not guaranteed that any of the RDMA Writes is persistent until RDMA Read completion returns or you receive RDMA Send with a confirmation. The initial implementation that allows this approach is implemented in `rpmem_flush()` and `rpmem_drain()` API call pair. Where `rpmem_flush()` performs RDMA Write and returns immediately, and `rpmem_drain()` posts RDMA Read and waits for its completion (at the time of publication it is not implemented in the write/send model).

There are many performance considerations, including the high-level networking model being used. Traditional best-in-class networking architecture typically relies on a *pull* model between the initiator and target. In a pull model, the initiator requests resources from the target, but the target server only pulls the data across via RDMA Read when it has the resources and connection bandwidth. This server-centric view allows the target node to handle hundreds or thousands of connections since it is in complete control of all resources for all of the connections and initiates the networking transactions when it chooses. With the speed and low latency of persistent memory, a *push* model can be used where the initiator and target have pre-allocated and registered memory resources and directly RDMA Write the data without waiting for server-side resource coordination. Microsoft's SNIA DevCon RDMA presentation describes the push/pull model in more detail: (https://www.snia.org/sites/default/files/SDC/2018/presentations/PM/Talpey_Tom_Remote_Persistent_Memory.pdf).

Remote Replication Error Handling

`librpmem` replication failures will occur for either a lost socket connection or a lost RDMA connection. Any error status returned from `rpmem_persist()`, `rpmem_flush()`, and `rpmem_drain()` is typically treated as an unrecoverable failure. The `libpmemobj` user of `librpmem` API should treat this as a lost socket or RDMA condition and should wait for all remaining `librpmem` API calls to complete, call `rpmem_close()` to close the connection and clean up the stack, then force the application to exit. When the application restarts, the files will be re-opened on both ends, and `libpmemobj` will check only the file metadata. We recommend you do not proceed before synchronizing local and remote memory pools with the pmempool-sync(1) command.

Say Hello to the Replicated World

The beauty of the `libpmemobj` remote replication is that it does not require any changes to the existing `libpmemobj` application. If you take any `libpmemobj` application and provide it with the poolset file configured to use the remote replica, it will simply start replicating. No coding required.

To illustrate how to replicate persistent memory, we look at a Hello World type program demonstrating the replication process directly using the `librpmem` library. Listing 18-3 shows a part of the C program that writes the "Hello world" message to remote memory. If it discovers that the message in English is already there, it translates it to Spanish and writes it back to remote memory. We walk through the lines of the program at the end of the listing.

Listing 18-3. The main routine of the Hello World program with replication.

```
37    #include <assert.h>
38    #include <errno.h>
39    #include <unistd.h>
40    #include <stdio.h>
41    #include <stdlib.h>
42    #include <string.h>
43
44    #include <librpmem.h>
45
```

```c
/*
 * English and Spanish translation of the message
 */
enum lang_t {en, es};
static const char *hello_str[] = {
    [en] = "Hello world!",
    [es] = "¡Hola Mundo!"
};

/*
 * structure to store the current message
 */
#define STR_SIZE    100
struct hello_t {
    enum lang_t lang;
    char str[STR_SIZE];
};

/*
 * write_hello_str -- write a message to the local memory
 */
static inline void
write_hello_str(struct hello_t *hello, enum lang_t lang)
{
    hello->lang = lang;
    strncpy(hello->str, hello_str[hello->lang], STR_SIZE);
}

int
main(int argc, char *argv[])
{
    /* for this example, assume 32MiB pool */
    size_t pool_size = 32 * 1024 * 1024;
    void *pool = NULL;
    int created;

    /* allocate a page size aligned local memory pool */
    long pagesize = sysconf(_SC_PAGESIZE);
    assert(pagesize >= 0);
    int ret = posix_memalign(&pool, pagesize, pool_size);
    assert(ret == 0 && pool != NULL);

    /* skip to the beginning of the message */
    size_t hello_off = 4096; /* rpmem header size */
    struct hello_t *hello = (struct hello_t *)(pool + hello_off);

    RPMEMpool *rpp = remote_open("target", "pool.set", pool, pool_size,
            &created);
    if (created) {
        /* reset local memory pool */
        memset(pool, 0, pool_size);
        write_hello_str(hello, en);
    } else {
        /* read message from the remote pool */
        ret = rpmem_read(rpp, hello, hello_off, sizeof(*hello), 0);
        assert(ret == 0);

        /* translate the message */
        const int lang_num = (sizeof(hello_str) / sizeof(hello_str[0]));
        enum lang_t lang = (enum lang_t)((hello->lang + 1) % lang_num);
        write_hello_str(hello, lang);
    }

    /* write message to the remote pool */
    ret = rpmem_persist(rpp, hello_off, sizeof(*hello), 0, 0);
    printf("%s\n", hello->str);
    assert(ret == 0);

    /* close the remote pool */
    ret = rpmem_close(rpp);
    assert(ret == 0);

    /* release local memory pool */
```

```
149         free(pool);
150         return 0;
151     }
```

- Line 68: Simple helper routine for writing message to the local memory.
- Line 115: Allocate a big enough block of memory, which is aligned to the page size. The required block size is hardcoded, whereas the alignment is required if you want to make this memory block available for RDMA transfers.
- Line 122: The `remote_open()` routine creates or opens the remote memory pool.
- Line 126-127: The local memory pool is initialized here. It is performed only once when the remote memory pool was just created so it does not contain any message.
- Line 130: A message from the remote memory pool is read to the local memory here.
- Lines 134-1136: If a message from the remote memory pool was read correctly, it is translated locally.
- Line 140: The newly initialized or translated message is written to the remote memory pool.
- Line 145: Close the remote memory pool.
- Line 149: Release remote memory pool.

The last missing piece of the whole process is how the remote replication is set up. It is all done in the `remote_open()` routine presented in Listing 18-4.

Listing 18-4. A remote_open routine from the Hello World program with replication.

```
/*
 * remote_open -- setup the librpmem replication
 */
static inline RPMEMpool*
remote_open(const char *target, const char *poolset, void *pool,
        size_t pool_size, int *created)
{
    /* fill pool_attributes */
    struct rpmem_pool_attr pool_attr;
    memset(&pool_attr, 0, sizeof(pool_attr));
    84        strncpy(pool_attr.signature, "HELLO",
RPMEM_POOL_HDR_SIG_LEN);
    85
    86        /* create a remote pool */
    87        unsigned nlanes = 1;
    88        RPMEMpool *rpp = rpmem_create(target, poolset,
pool, pool_size, &nlanes,
    89                &pool_attr);
    90        if (rpp) {
    91            *created = 1;
    92            return rpp;
    93        }
    94
    95        /* create failed so open a remote pool */
    96        assert(errno == EEXIST);
    97        rpp = rpmem_open(target, poolset, pool,
pool_size, &nlanes, &pool_attr);
    98        assert(rpp != NULL);
    99        *created = 0;
   100
   101        return rpp;
   102    }
```

- Line 88: A remote memory pool can be either created or opened. When it is used for the first time, it must be created so that it is available for the opening afterwards. We first try to create it here.
- Line 97: Here we attempt to open the remote memory pool. We assume it exists because of the error code received during the create try (EEXIST).

Execution Example

The Hello World application produces the output shown in Listing 18-5.

Listing 18-5. An output from the Hello World application for librpmem.

```
[user@initiator]$ ./hello
Hello world!
[user@initiator]$ ./hello
¡Hola Mundo!
```

Listing 18-6 shows the contents of the target persistent memory pool where we see the “Hola Mundo” string.

Listing 18-6. The ¡Hola Mundo! snooped on the replication target.

```
[user@target]$ hexdump -s 4096 -C /mnt/pmem1/pool2
00001000  01 00 00 00 c2 a1 48 6f  6c 61 20 4d 75 6e 64 6f
|......Hola Mundo|
00001010  21 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
|!...............|
00001020  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
|................|
*
00002000
```

Summary

It is important to know that neither the general-purpose remote replication method nor the appliance remote replication method is ideal because vendor-specific platform features are required to use non-allocating writes, adding the complication of effecting performance on an entire root device. Conversely, flushing remote writes using allocating-writes requires a painful interrupt of the target system to intercept an RDMA Send request and flush the list of regions contained within the send buffer. Waking the remote node is extremely painful in a cloud environment because there are hundreds or thousands of inbound RDMA requests from many different connections; avoid this if possible.

There are cloud service providers using these two methods today and getting phenomenal performance results. If the persistent memory is used as a replacement for a remotely accessed SSD, huge reductions in latency can be achieved.

As the first iteration of remote persistence support, we focused on application/library changes to implement these high-level persistence methods, without hardware, firmware, driver, or protocol changes.   At the time of publication, IBTA and IETF drafts for a new wire protocol extension for persistent memory is nearing completion. This will provide native hardware support for RDMA to persistent memory and allow hardware entities to route each IO to its destination memory device without the need to change allocating-write mode, and without the potential to adversely affect performance on collateral devices connected to the same root port. See Appendix E for more details on the new extensions to RDMA, specifically for remote persistence.

RDMA protocol extensions are only one step into further remote persistent memory development. Several other areas of improvement are already identified and shall be addressed to the remote persistent memory users community, including: Atomicity of remote operations, advanced error handling (including RAS), dynamic configuration of remote persistent memory and custom setup, and real 0 percent CPU utilization on remote/target replication side

As this book has demonstrated, unlocking the true potential of persistent memory may require new approaches to existing software and application architecture. Hopefully, this chapter gave you an overview of this complex topic, the challenges of working with persistent memory remotely, and the many aspects of software architecture to consider when unlocking the true performance potential.