Project Technical Report: Implementing Hyperloop

Computer Architecture and Systems Lab
School of Computing, KAIST

Insu Jang insujang@casys.kaist.ac.kr
Jongyul Kim jongyul@casys.kaist.ac.kr

July 13, 2020

KAIST School of Computing

Contents

Chapter 1
Introduction

The report describes prerequisite RDMA backgrounds, how we implemented Hyperloop, and limitations that we faced during implementation. Our implementation is not perfect due to the limitations, and only simulates its behaviors.

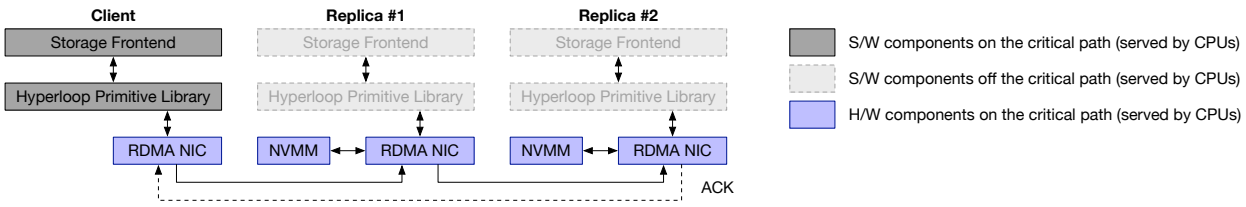
Hyperloop is a high-performance framework that removes CPU involvement from the critical path of replicated transactions in storage systems [21]. It primarily focus on large latencies on replicated transactions, which brings worse storage application performance. While other proposals used CPU I/O polling to reduce latencies, Hyperloop leverages Remote Direct Memory Access (RDMA) to eliminate CPU's involvement from the critical path of replicated transactions entirely by offloading operations to RDMA Network Interface Cards (RNICs). Figure 1.1 illustrates how replicated transactions in Hyperloop based storage system works.

Its design contains two key ideas: leveraging *RDMA WAIT operations* to trigger operations without CPUs, and *remote work request manipulation* to indicate *how* to replicate data. WAIT enables RNICs to wait for a completion of one or more work requests (WRs), and to trigger other WRs that are pre-posted on the same WR queue that the WAIT WR is posted, working as a barrier. Its main contribution is *remote work request manipulation*; the preposted WRs are useless due to lack of information regarding which data a client would want to operate, however, *remote work request manipulation* enables the client to remotely modify the WRs without need of CPU's operations in the replica, so that the manipulated WRs operate as desired.

However, after deeply analyzing RDMA, we concluded that Hyperloop would only work in strict circumstance with specific hardware configurations, which we do not have. Hence, the implementation of Hyperloop in the report approximately simulates it and does not work identically to the original one. The report introduces these limitations as well, and what would be needed to get an actual working Hyperloop.

In Chapter 2, we introduce RDMA backgrounds, including software architecture and Infiniband verb API. Then, we introduce our implementation of Hyperloop and its limitations in Chapter 3 and 4, respectively. Finally, we evaluate the performance and compare it with that of Hyperloop from the paper in Chapter 5.

Figure 1.1: Hyperloop architecture. Hyperloop works as a chain-based manner, but CPUs are not used during performing replicated transactions.



Chapter 2
Backgrounds

In this chapter, we introduce RDMA architecture and software APIs. Also, we provide some C++ backgrounds to help understanding the implementation.

2.1 RDMA Architecture

Hyperloop is a software framework that is entirely based on Infiniband based RDMA. RDMA provides high-performance, low latency, and low CPU overhead data communication. Figure 2.1 briefly illustrates how RDMA works. Once the memory buffer is registered to be accessible, remote peer nodes can read or write directly to the memory buffer using RDMA verb APIs.

RNICs have *(host) channel adapters (HCAs)*, which can create Infiniband packets for RDMA operations. Each CA is represented as a PCIe device function in Linux as follows.

```
$ lspci -v -s 18:00.*
18:00.0 Infiniband controller: Mellanox Technologies MT27700 Family [ConnectX-4]
  Subsystem: Mellanox Technologies MT27700 Family [ConnectX-4]
  Flags: bus master, fast devsel, latency 0, IRQ 193, NUMA node 0
  Memory at 387ffe000000 (64-bit, prefetchable) [size=32M]
  Expansion ROM at aae00000 [disabled] [size=1M]
  Kernel driver in use: mlx5_core
  Kernel modules: mlx5_core

18:00.1 Infiniband controller: Mellanox Technologies MT27700 Family [ConnectX-4]
  Subsystem: Mellanox Technologies MT27700 Family [ConnectX-4]
  Flags: bus master, fast devsel, latency 0, IRQ 215, NUMA node 0
  Memory at 387ffc000000 (64-bit, prefetchable) [size=32M]
  Expansion ROM at aad00000 [disabled] [size=1M]
  Kernel driver in use: mlx5_core
  Kernel modules: mlx5_core
```

The system above has a Mellanox MT27700 ConnectX-4 PCIe device that has two PCIe device functions. It has two Infiniband ports, which are associated to two PCIe device functions, respectively.

Several software libraries are provided to use RDMA. Linux mainline also contains kernel features to support Infiniband, but its functionalities are limited. Mellanox OFED device drivers provides richer

Figure 2.1: RDMA architecture. With RNICs that support RDMA, applications on one node can access remote node's memory space. This operation is handled by RNICs without CPU participation, reducing CPU overheads in communication. An interconnect medium between RNICs can be either Infiniband or traditional Ethernet (RoCE: RDMA over Converged Ethernet).

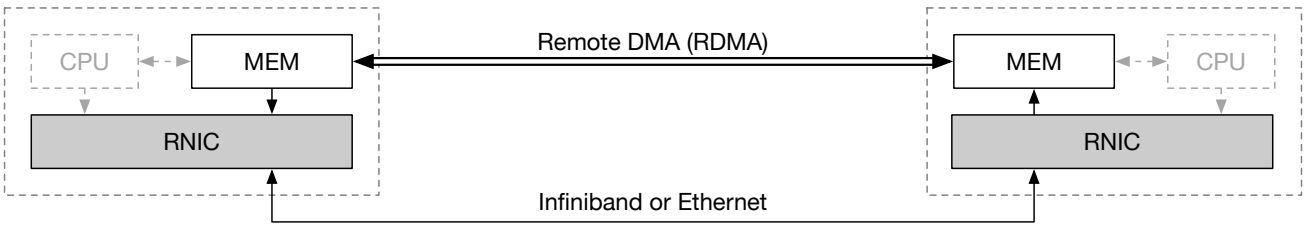
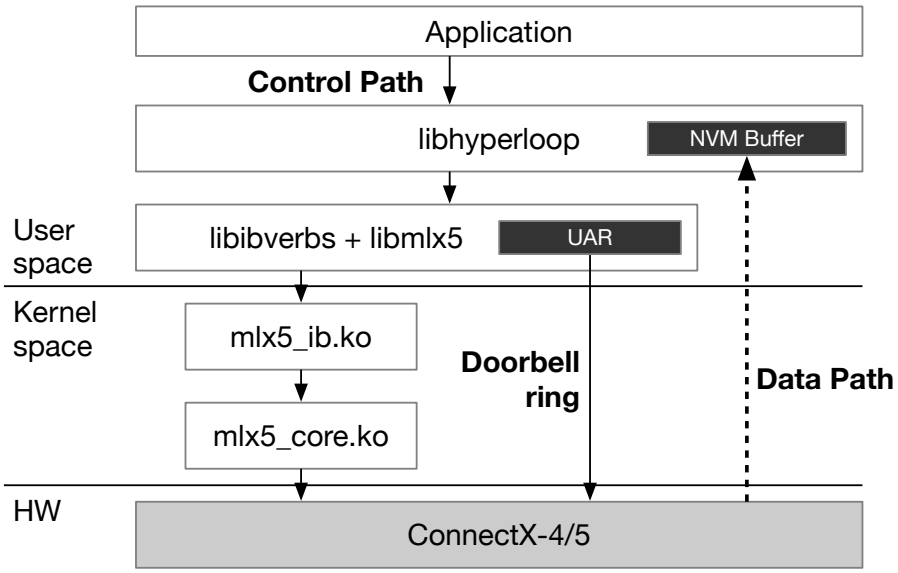


Figure 2.2: Control paths and a data path between an Infiniband RNIC and a software application.



functionalities, hence is recommended to be used [30].

With the Mellanox OFED device driver, communication paths from Hyperloop to the hardware can be illustred as Figure 2.2, similar to those in Mellanox Data Plane Development Kit (DPDK) [32].

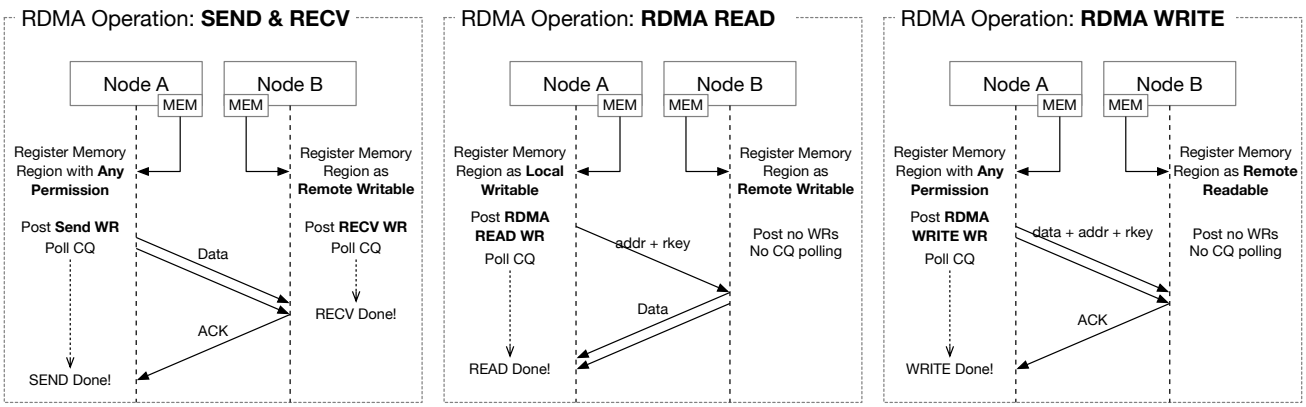
`libibverbs` provides RDMA verb API functions, hiding device-specific behaviors from developers (unlike its name, it also works with RoCE as well, not only Infiniband). It internally uses `libmlx4` or `libmlx5` depending on the hardware an user application specifies. It uses both kernel modules and a direct communication path to talk to the device. The userspace drivers use the kernel modules to register objects (queue pairs, memory regions, etc) and to get a memory-mapped Doorbell register address that can directly access to a device's Doorbell register. Once objects are registered and the memory-mapped register address is given, `libmlx5` can ring the doorbell directly without user-kernel context switches, reducing context switch overheads. Once the doorbell is rung, the RNIC performs DMA operations to read user's command and data. Note that user-requested operations are also copied into the device via DMAs initiated by the device [20]. When the device is notified via a doorbell ring, it first reads UAR to read which operations the user requested. Then, it accesses the user buffer specified in the user requests.

2.1.1 Work Requests (WRs) and Work Queues (WQs)

Clients' operations are represented as *Work Requests (WRs)*. A WR is also called as a *Work Queue Entry (WQE)*, since it is an entry posted into a WQ. For instance, a send data request can be expressed by a *SEND WR* or *SEND WQE*. `libibverbs` provides several types of WRs to be used:

- SEND WR: send the given data to the remote node. It must be paired with RECV WRs at the peer node; otherwise it returns a Receive Not Ready (RNR) error. Gather list feature is supported.
- RECV WR: wait until data arrives, and copy data to the specified buffer in the node. Scatter list feature is supported. It must be paired with SEND WRs at the peer node, and must be posted before the SEND WRs are posted to prevent a Receive Not Ready (RNR) error.
- RDMA READ WR: read data in the peer node **with no notifications to processes running in the peer node.** No software can know whether the operation is performed in the node.

Figure 2.3: Flow diagrams for some WR types.



- RDMA WRITE WR: write data into the peer node **with no notifications to processes running in the peer node.** No software can know whether the operation is performed in the node.
- Compare and Swap (CAS) WR: check whether 8-byte data at the peer node's memory is same with value 1, and change the data to value 2 if so.
- Fetch and Add (FAA) WR: provide an atomic increase for 8-byte data at the peer node.
- WAIT WR: it is an experimental feature; wait until some works are completed, preventing following operations from being performed.

Figure 2.3 illustrates how some of the WRs work. Fetch and Add and Compare and Swap operations are grouped and called as *Atomic operations*.

When the client posts WRs, they are posted into Work Queues (WQs), which are internally managed by `libmlx5`. There are three types of WQs:

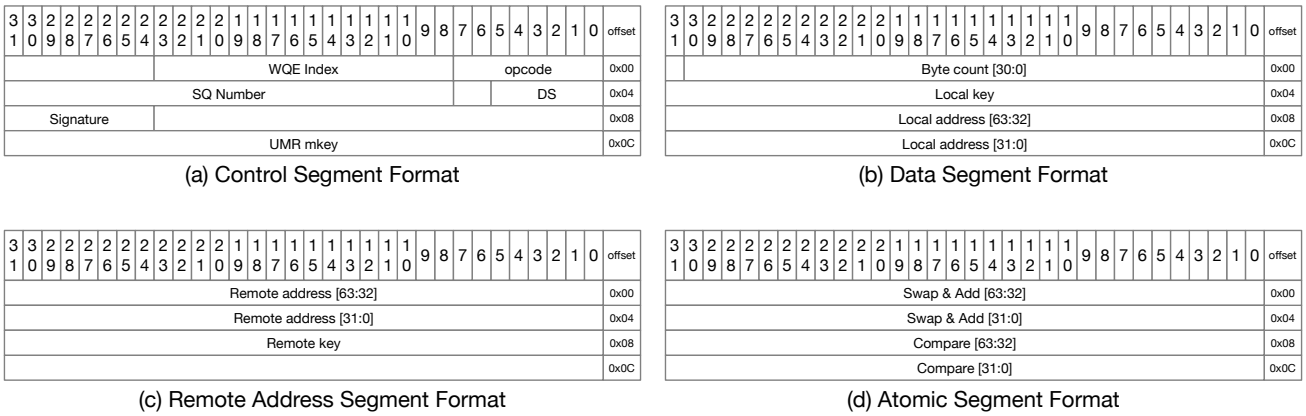
- Send Queue (SQ): a queue where SEND WRs, RDMA READ WRs, RDMA WRITE WRs, WAIT WRs, and Atomic WRs are posted.
- Receive Queue (RQ): a queue where RECV WRs are posted.
- Work Completion Queue (CQ): a queue where Work Completions (WCs) are posted.

While the client can post WRs into SQs and RQs, CQs are different; it is the hardware that generates WCs and posts them into CQs. The client reads WCs from the CQs to check whether the posted WRs are completed. Basically it adopts a polling method, so the client needs to poll the CQ continuously until the hardware posts a WC into the CQ after posting a WR, however, it also provides an event based blocking method.

2.1.1.1 Internal Structure of Work Queue Entries (WQEs)

Hyperloop implements *remote work request manipulation*, which requires background knowledge about the structure of WRs. WRs consist of a group of *Work Request Entry Basic Blocks (WQEBBs)*, each of which contains a set of *segments*. In Mellanox implementation, the size of a WQEBB is 64 bytes and the size of segments is a multiple of 16 bytes. Size of WR is a modulo of 64 bytes and it varies depending on the number of segments it contains.

Figure 2.4: Data format of 4 types of segments. Wait segment is omitted due to its simplicity.



(a) Control Segment Format

(b) Data Segment Format

(c) Remote Address Segment Format

(d) Atomic Segment Format

This report only describes SEND, RDMA WRITE, CAS, NOP, and WAIT WRs, which contain the following 4 types of segments (Section 7.4.4.1 in [24]):

- Control Segment: it contains control information for the WQE (Section 7.4.4.1.1 in [24]).
- Data Pointer Segment: it contains pointers and a byte count for the scatter/gather list (Section 7.4.4.1.4 and 7.4.4.1.5 in [24]).
- Remote Address Segment: it contains pointers at remote side.
- Atomic Segment: it contains information about Atomic operations (FAA and CAS).
- Wait Segment: it contains wait information that how many WCs it waits from which QP.

Some segments are not documented in the manual, however, they can be found in source code.

```
1  struct mlx5_wqe_ctrl_seg {
2    uint32_t  opmod_idx_opcode;
3    uint32_t  qpn_ds;
4    uint8_t   signature;
5    uint8_t   rsvd[2];
6    uint8_t   fm_ce_se;
7    uint32_t  imm;
8  };
9
10 struct mlx5_wqe_data_seg {
11   uint32_t    byte_count;
12   uint32_t    lkey;
13   uint64_t    addr;
14 };
15
16 struct mlx5_wqe_raddr_seg {
17   uint64_t  raddr;
18   uint32_t  rkey;
19   uint32_t  reserved;
20 };
21
22 struct mlx5_wqe_atomic_seg {
23   uint64_t  swap_add;
24   uint64_t  compare;
25 };
```

```
26
27 struct mlx5_wqe_wait_en_seg {
28   uint8_t   rsvd0[8];
29   uint32_t  pi;
30   uint32_t  obj_num;
31 };
```

Listing 2.1: Source code from `libmlx5/src/mlx5dv.h` and `libmlx5/src/wqe.h`.

Based on the manul and the source code, segments' format can be described as Figure 2.4.

A WQE is represented as a group of segments. For instance, a SEND WR with a gather list with 2 pointers contains 3 segments: a control segment and two data pointer segments. In this case, size of the SEND WR is 64 bytes, made up of one WQEBB, since those three segments can be packed into a WQEBB. As an another example, a RDMA WRITE WR with a gather list with 4 pointers consists of 6 segments: a control segment, a remote address segment, and 4 data pointer segments, each of which represent a pointer in the gather list. The RDMA WRITE WR is 128 bytes, containing two WQEBBs.

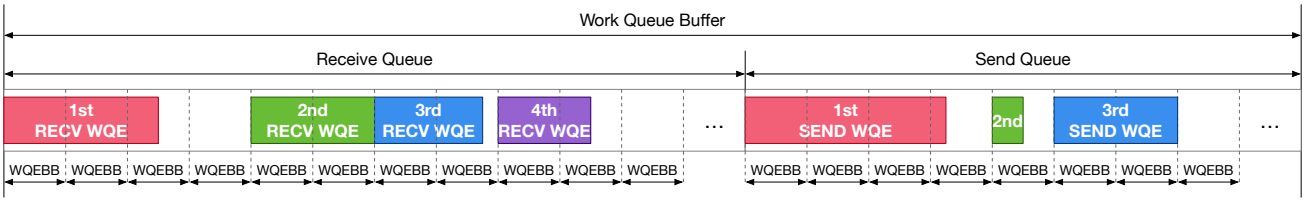
Writing segment data into WQE buffer is done by several corresponding functions.

```
1  static MLX5DV_ALWAYS_INLINE
2  void mlx5dv_set_ctrl_seg(struct mlx5_wqe_ctrl_seg *seg, uint16_t pi,
3                           uint8_t opcode, uint8_t opmod, uint32_t qp_num,
4                           uint8_t fm_ce_se, uint8_t ds,
5                           uint8_t signature, uint32_t imm) {
6    seg->opmod_idx_opcode = htonl(((uint32_t)opmod << 24) |
7                                  ((uint32_t)pi << 8) |
8                                  opcode);
9    seg->qpn_ds           = htonl((qp_num << 8) | ds);
10   seg->fm_ce_se         = fm_ce_se;
11   seg->signature        = signature;
12   seg->imm              = imm;
13 }
```

Listing 2.2: Source code from `libmlx5/src/mlx5dv.h`.

```
1  static inline int set_data_ptr_seg(struct mlx5_wqe_data_seg *dseg,
2                                     struct ibv_sge *sg,
3                                     struct mlx5_qp *qp,
4                                     int offset) {
5    dseg->byte_count = htonl(sg->length - offset);
6    dseg->lkey       = htonl(sg->lkey);
7    dseg->addr       = htonll(sg->addr + offset);
8    return 0;
9  }
10
11 static inline void set_raddr_seg(struct mlx5_wqe_raddr_seg *rseg,
12                                   uint64_t remote_addr, uint32_t rkey) {
13   rseg->raddr    = htonll(remote_addr);
14   rseg->rkey     = htonl(rkey);
15   rseg->reserved = 0;
16 }
17
18 static void set_atomic_seg(struct mlx5_wqe_atomic_seg *aseg,
19                            enum ibv_wr_opcode opcode,
20                            uint64_t swap,
21                            uint64_t compare_add) {
22   if (opcode == IBV_WR_ATOMIC_CMP_AND_SWP) {
23     aseg->swap_add = htonll(swap);
24     aseg->compare  = htonll(compare_add);
25   } else {
```

Figure 2.5: An example of Work Queue with 4 RECV WRs and 3 SEND WRs posted.




```
26      aseg->swap_add = htonll(compare_add);
27      aseg->compare  = 0;
28    }
29  }
30
31  static inline void set_wait_en_seg(void *wqe_seg,
32                                     uint32_t obj_num, uint32_t count)
33  {
34    struct mlx5_wqe_wait_en_seg *seg = (struct mlx5_wqe_wait_en_seg *)wqe_seg;
35    seg->pi      = htonl(count);
36    seg->obj_num = htonl(obj_num);
37  }
```

Listing 2.3: Source code from `libmlx5/src/qp.c`.

The order of segments for each WR type is not documented either, but also found in the source code with careful analysis. We analyzed the segment order of the following WRs:

- SEND WR: 1 control segment (opcode: 0x0a), and *n* data pointer segments
- RDMA WRITE WR: 1 control segment (opcode: 0x08), 1 remote address segment, and *n* data pointer segments
- WAIT WR: 1 control segment (opcode: 0x0f) and 1 wait segment
- Compare and Swap (CAS) WR: 1 control segment (opcode: 0x11), 1 remote address segment, and 1 atomic segment
- NOP WR: 1 control segment (opcode: 0x00)

2.1.1.2 Internal Structure of Work Queues (WQs)

Understanding the structure of WQE lets us know *how* to modify WQEs, but we also need to know about the structure of WQ to know *where* data for manipulation should be written.

The WQ is a virtually-contiguous memory buffer used by software to post WRs for RDMA execution. A WQ is comprised of WQEBBs; it is the reason WRs must be aligned to size of WQEBB and their size is modulo of WQEBB size. The WQ size is also a power-of-two number of WQEBBs as well. A WQ buffer contains a Send WQ (SQ) and a Receive WQ (RQ) *adjancently*. The RQ resides in the beginning of the buffer, followed by the SQ.

Figure 2.5 illustrates an example of a WQ. It looks like the WQ is a sequential buffer in the illustration, however, it is actually a ring buffer so that WQEs will be posted at the beginning of the WQ if the next location reaches the end of it. However, it is an experimental feature and not enabled by default, hence it is a programmer's responsibility to explicitly set a flag into the WQ.

`libmlx5` uses `mlx5_get_send_wqe()` to find where the next Send WR should be posted:

```
1  static inline int __mlx5_post_send(struct ibv_qp *ibqp,
2                                     struct ibv_exp_send_wr *wr,
3                                    struct ibv_exp_send_wr **bad_wr,
4                                    int is_exp_wr) {
5    struct mlx5_qp *qp = to_mqp(ibqp);
6    void *uninitialized_var(seg);
7    int nreq;
8    unsigned idx;
9
10   for (nreq = 0; wr; ++nreq, wr = wr->next) {
11     idx = qp->gen_data.scur_post & (qp->sq.wqe_cnt - 1);
12     seg = mlx5_get_send_wqe(qp, idx);
13     err = qp->gen_data.post_send_one(wr, qp, exp_send_flags, seg, &size);
14
15     qp->gen_data.wqe_head[idx] = qp->sq.head + nreq;
16     qp->gen_data.scur_post += DIB_ROUND_UP(size * 16, MLX5_SEND_WQE_BB);
17   }
18   ...
19 }
20
21 static inline void *mlx5_get_send_wqe(struct mlx5_qp *qp, int n) {
22   return qp->gen_data.sqstart + (n << MLX5_SEND_WQE_SHIFT);
23 }
```

Listing 2.4: Source code from `libmlx5/src/qp.c` and `libmlx5/src/mlx5.h`.

where `qp->gen_data.sqstart` indicates the base address of the SQ, initialized as follows during Queue Pair creation.

```
1  static int mlx5_calc_wq_size(struct mlx5_context *ctx,
2                               struct ibv_exp_qp_init_attr *attr,
3                               struct mlx5_qp *qp){
4    int ret;
5    int result;
6
7    ret = mlx5_calc_sq_size(ctx, attr, qp);
8    result = ret;
9
10   ret = mlx5_calc_rq_size(ctx, attr, qp);
11   result += ret;
12
13   qp->sq.offset = ret;
14   qp->rq.offset = 0;
15
16   return result;
17 }
18
19 static struct ibv_qp *create_qp(struct ibv_context *context,
20                                 struct ibv_exp_qp_init_attr *attrx,
21                                 int is_exp) {
22   ...
23   qp->gen_data.sqstart = qp->buf.buf + qp->sq.offset;
24   qp->gen_data.sqend = qp->buf.buf + qp->sq.offset +
25                        (qp->sq.wqe_cnt << qp->sq.wqe_shift);
26   ...
27 }
```

Listing 2.5: Source code from `libmlx5/src/verbs.c`. The code calculates the size and the offset of the RQ and the SQ.

In summary, we can find where WQs reside in the process virtual memory address space:

- Send Queue: `[qp->gen_data.sqstart~qp->gen_data.sqend)`
  (The base address is equal to `qp->buf.buf+qp->sq.offset = qp->sq.buff`.)
- Receive Queue: `[qp->rq.buff~qp->rq.buff+qp->sq.offset)`
  (The base address is equal to `qp->buf.buf`.)

2.1.2 User Access Region (UAR), Doorbell Register, and Doorbell Region

2.1.2.1 User Access Region (UAR)

The User Access Region (UAR) is a part of PCI address space mapped for direct access to the device (Section 7.2 in [24]). Each page in the UAR contains registers that control the device operations. Different processes have different UARs; they are isolated and protected by the kernel module, hence a process can control the device through its own UAR.

The virtual address mapped to PCIe address space of the device, where the client can access, is managed by the kernel module `mlx5_ib.ko`. It registers `mlx5_ib_mmap()` as a callback function for user mmap request, which eventually calls `rdma_user_mmap_io()`, a public function served by the other kernel module `mlx5_core.ko`.

```
1  static const struct ib_device_ops mlx5_ib_dev_ops = {
2    .mmap = mlx5_ib_mmap,
3  };
4
5  static int mlx5_ib_mmap(struct ib_ucontext *ibcontext,
6                          struct vm_area_struct *vma) {
7    struct mlx5_ib_ucontext *context = to_mucontext(ibcontext);
8    struct mlx5_ib_dev *dev = to_mdev(ibcontext->device);
9    unsigned long command;
10   command = get_command(vma->vm_pgoff);
11
12   switch (command) {
13     case MLX5_IB_MMAP_WC_PAGE:
14     case MLX5_IB_MMAP_NC_PAGE:
15     case MLX5_IB_MMAP_REGULAR_PAGE:
16     case MLX5_Ib_MMAP_ALLOC_WC:
17       return uar_mmap(dev, command, vma, context);
18   }
19 }
20
21 static int uar_mmap(struct mlx5_ib_dev *dev,
22                     enum mlx5_ib_mmap_cmd cmd,
23                     struct vm_area_struct *vma,
24                     struct mlx5_ib_ucontext *context) {
25   int err;
26   phys_addr_t pfn;
27
28   pfn = uar_index2pfn(dev, uar_index);
29   err = rdma_user_mmap_io(&context->ibucontext, vma, pfn, PAGE_SIZE, prot, NULL);
30   ...
31 }
```

Listing 2.6: Source code from `drivers/infiniband/hw/mlx5/main.c`.

`rdma_user_mmap_io()` uses `io_remap_pfn_range()` to remap kernel space address to user space virtual address, where the target kernel space address is in device's PCIe address space.

```c
/*
 * Map IO memory into a process. This is to be called by drivers as part of
 * their mmap() functions if they wish to send somthing like PcI-E BAR memory
 * to userspace.
 */
int rdma_user_mmap_io(struct ib_ucontext *ucontext,
                      struct vm_area_struct *vma,
                      unsigned long pfn, unsigned long size,
                      pgprot_t prot, struct rdma_umap_priv *priv) {
  vma->vm_page_prot = prot;
  if (io_remap_pfn_range(vma, vma->vm_start, pfn, size, prot)) {
    return -EAGAIN;
  }
  ...
}
```

Listing 2.7: Source code from `drivers/infiniband/core/uverbs_main.c`.

During Infiniband context creation procedure, `libmlx5` initializes multiple UAR pages.

```c
static int mlx5_alloc_context(struct verbs_device *vdev,
                              struct ibv_context *ctx, int cmd_fd) {
  offset = 0;
  set_command(MLX5_MMAP_MAP_DC_INFO_PAGE, &offset);
  context->cc.buf = mmap(NULL, 4096 * context->num_ports, PROT_READ, MAP_PRIVATE,
                         cmd_fd, page_size * offset);

  num_sys_page_map = context->tot_uuars /
                     (context->num_uars_per_page *
                      MLX5_NUM_NON_FP_BFREGS_PER_UAR);
  for (i = 0; i < num_sys_page_map; ++i) {
    uar_mmaped = 0;

    // In actual code the second argument varies depending on configurations.
    // Either MLX5_MMAP_GET_WC_PAGES_CMD, MLX5_MMAP_GET_NC_PAGES_CMD, or
    // MLX5_MMAP_GET_REGULAR_PAGES is used.
    context->uar[i].regs = mlx5_uar_mmap(i, MLX5_MMAP_GET_WC_PAGES_CMD,
                                         page_size, cmd_fd);
  }
  ...
}

void *mlx5_uar_mmap(int idx, int cmd, int page_size, int cmd_fd) {
  off_t offset = 0;
  set_command(cmd, &offset);
  set_index(idx, &offset);

  return mmap(NULL, page_size, PROT_WRITE, MAP_SHARED, cmd_fd,
              page_size * offset);
}
```

Listing 2.8: Source code from `libmlx5/src/mlx5.c`.

In the test system, `num_sys_page_map` was calculated as 8, hence it allocated 1+8 UAR pages per context. The first one was always the second page from the beginning of PCIe BAR address (`BAR0+0x1`), and the remaining 8 pages are allocated continuously from the seventeenth page from the beginning of PCIe BAR (`BAR0+0x11`). Hence, the first process has 8 UAR pages `BAR0+0x11` $\sim$ `BAR0+0x18`, then the next process has another 8 UAR pages following the ones of the first process `BAR0+0x19` $\sim$ `BAR0+0x21`, etc.

2.1.2.2 Doorbell Register

Details of UAR pages are not well documented, however, it would be enough to know to understand the report that the Doorbell register is in the UAR. The Doorbell register is directly mapped to the device's doorbell register, so that when the client rings a Doorbell register by wrting a value to it, its ring is directly sent to the device through a PCIe transaction without user-kernel context switches. Doorbell registers are at offset `0x800` of the UAR pages (Refer to Section 7.2.2 in [24] for more detailed UAR page format). Two pages `(0x800 ~ 0x9ff)` of the UAR page are called *blue flame (BF) registers*, and Doorbell registers are at offset 0 of the BF registers.

For receive operations, receive Doorbell ring is not required. Though there is no explicit explanations in the manual, it may be due to the property of the receive operation; when the HCA tries to send data to the remote node, it contacts to the HCA at the node. At this moment the HCA at the remote node can check whether there exist posted RECV WRs without Doorbell register ring.

For send operations, send Doorbell ring is done in `libmlx5/src/doorbell.h`, by writing the first 8 bytes of the WQE (if multiple WQEs are posted at once, those of the last one) to the register.

```
1  static inline int __ring_db(struct mlx5_qp *qp,
2                              const int db_method,
3                              uint32_t curr_post,
4                              unsigned long long *seg,
5                              int size)
6                   __atribute__((always_inline)) {
7    struct mlx5_bf *bf = qp->gen_data.bf;
8
9    qp->gen_data.last_post = curr_post;
10
11   /*
12    * Make sure that descriptors are written before
13    * updating doorbell record and ringing the doorbell
14    */
15    wmb();
16    qp->gen_data.db[MLX5_SND_DBR] = htonl(curr_post);
17
18    /* This wmb ensures ordering between DB record and DB ringing */
19    wmb();
20    mlx5_write64((__be32 *)seg, bf->reg + bf->offset, &bf->lock);
21 }
```

Listing 2.9: Source code from `libmlx5/src/doorbell.h`. The code is used for ringing a send Doorbell register.

```
1  static inline int __mlx5_post_send(struct ibv_qp *ibqp,
2                                     struct ibv_exp_send_wr *wr,
3                                     struct ibv_exp_send_wr **bad_wr,
4                                     int is_exp_wr) {
5    err = qp->gen_data.post_send_one(wr, qp, exp_send_flags, seg, &size);
6    ...
7    __ring_db(qp, qp->gen_data.bf->db_method, qp->gen_data.scur_post & 0xffff,
8              wqe2ring, (size + 3) / 4);
9  }
```

Listing 2.10: Source code from `libmlx5/src/qp.c`.

The location of its Doorbell register (`bf->reg`), which is actually `qp->gen_data.bf->reg`, is initialized as follows. `map_uuar` is called during creating a QP.

```
1  static int mlx5_alloc_context(struct verbs_device *vdev,
```

```
2                                  struct ibv_context *ctx, int cmd_fd) {
3   // UARs (context->uar[i].regs) are initialized before.
4   for (i = 0; i < num_sys_page_map; i++) {
5     for (j = 0; j < context->num_uars_per_page; j++) {
6       for (k = 0; k < NUM_BFREGS_PER_UAR; k++) {
7         bfi = (i * context->num_uars_per_page + j) *
8               NUM_BFREGS_PER_UAR + k;
9         context->bfs[bfi].reg = context->uar[i].regs +
10                                MLX5_ADAPTER_PAGEPSIZE * j +
11                                MLX5_BF_OFFSET + k * context->bf_reg_size;
12  ...
13 }
14
15 static void map_uuar(struct ibv_context *context,
16                      struct mlx5_qp *qp, int uuar_index) {
17   struct mlx5_context *cctx = to_mctx(context);
18   qp->gen_data.bf = &ctx->bfs[uuar_index];
19 }
```

Listing 2.11: Source code from `libmlx5/src/mlx5.c (mlx5_alloc_context)` and `libmlx5/src/verbs.c (mmap_uuar)`.

2.1.2.3 Doorbell Record

In the function `__ring_db()`, *Doorbell Record* is newly introduced (Section 7.4.2 in [24]). It is a 8-byte memory that the HCA can access via DMA, storing the number of WQEBBs for SQ and RQ.

Doorbell Record is an important feature in terms of *WQE ownership* (Section 7.4.3 in [24]); in communication between hardware and software, data synchronization is important since hardware can read and execute WQEs *asynchnously* to software posting new WQEs. Without synchronization, data corruption can happen and hardware would show undefined behavior.

To prevent this, Mellanox introduced *WQE ownership*. It was represented as a bit in each WQE in `libmlx4`, but has been moved to a dedicated memory buffer in `libmlx5`. The HCA will execute WQEs only if its ownership is transferred to itself, by inspecting the value of Doorbell Record. It is definitely different from Doorbell register; Doorbell register it at PCI address space, while Doorbell record is at memory. Doorbell register is sued to notify the HCA that there exists WRs to be performed, but Doorbell record is used to indicate WQE ownership.

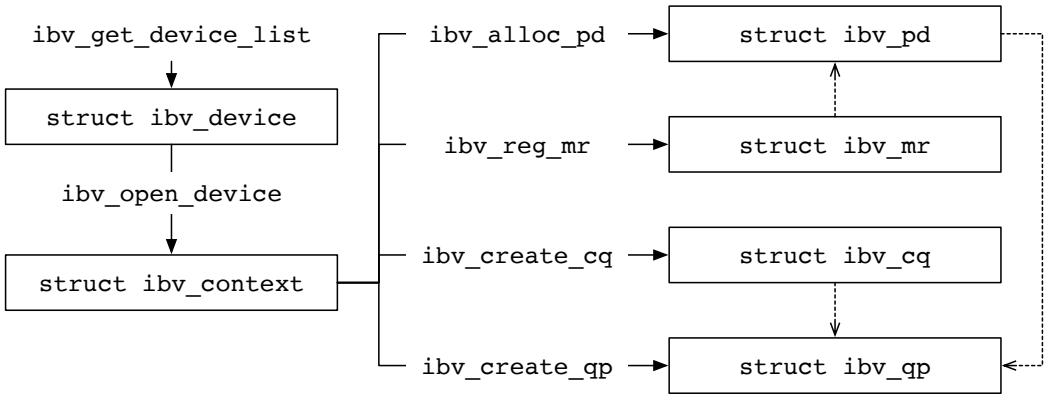
For software, Posted WQEs are in physical memory, and can even be modified until their ownership is transferred to hardware. After transferring the ownership, software should not modify the WQEs or data buffers, however, there seems no privileged force that denies the modification ([24] says: *Altering WQE or associated data buffer can have lethal consequences to that WQ*, which means it is at least possible to modify related data even after transferring WQE ownership).

Hence, posting a WR into the HCA consist of the following three stpes (Section 7.4.3 in [24]):

1. Write WQE to the WQE buffer sequentially to previously-posted WQE.
2. Update Doorbell Record associated with the queue.
3. For send request ring Doorbell by writing to the Doorbell register.

However, we found out that the HCA executes WQEs with no errors even our modified `libmlx5` library does not update Doorbell Record. Therefore, at this moment, we are not sure whether the concept of Doorbell Record is fully implemented.

Figure 2.6: Verb structures used in Hyperloop and their dependencies. Dot arrows indicate reference.



2.2 RDMA Verbs Application Programming Interface (API)

This section describes how to use an RDMA software library `libibverbs`. Many tutorials and examples are published, however, many features required for Hyperloop implementation were still undocumented and not explained well. Though the report does not cover everything, however, details are attached to APIs related to Hyperloop implementation. For more detailed explanation for functions that are not covered in the report, refer to [5, 23].

Figure 2.6 illustrates 6 structures provided by `libibverbs` that Hyperloop uses, and their dependencies. For instance, to create an `ibv_qp`, you need an `ibv_context`, an `ibv_pd`, and one or two `ibv_cq(s)`. There are experimental versions of the functions `ibv_create_cq` and `ibv_create_qp`: `ibv_exp_create_cq` and `ibv_exp_create_qp`. They provide richer features that are required for Hyperloop. Details will be discussed later.

2.2.1 Context and Protection Domain (PD)

```
struct ibv_device **ibv_get_device_list(int *num_devices);
struct ibv_context *ibv_open_device(struct ibv_device *device);

void ibv_free_device_list(struct ibv_device **list);
int ibv_close_device(struct ibv_context *context);
```

In every machine, there can be one ore more RDMA devices. `ibv_get_device_list` returns all RDMA devices as an array of `struct ibv_device`. We can find a device and open it to create a context by calling `ibv_open_device`. Note that it is the caller's responsibility to free the device list returned from `ibv_get_device_list` by calling `ibv_free_device_list`.

```
struct ibv_pd *ibv_alloc_pd(struct ibv_context *context);
int ibv_dealloc_pd(struct ibv_pd *pd);
```

After creating a context, a *protection domain* needs to be created. As inferred from its name, the protection domain is a domain for protection of verb objects. For instance, accessing resources from different PD will result a Work Completion with an error [5]. Not all resources are in PD. As you can see in 2.6, a Completion Queue (CQ) is not dependent to PD, so that it is not protected by the HCA.

```
1 static ibv_context* createInfinibandContext(const std::string& device_name) {
2   int num_devices;
3   auto device_list = ibv_get_device_list(&num_devices);
4   for (int i = 0; i < num_devices; i++) {
5     if (device_name.compare(ibv_get_device_name(device_list[i])) == 0) {
```

```
6         auto context = ibv_open_device(device_list[i]);
7         ibv_free_device_list(device_list);
8         return context;
9       }
10    }
11
12    abort();
13  }
14
15  static struct ibv_pd* createInfinibandProtectionDomain(
16      struct ibv_context& context) {
17    return ibv_alloc_pd(&context);
18  }
```

Listing 2.12: Source code from our Hyperloop implementation.

2.2.2 Queue Pairs (QPs) and Completion Queues (CQs)

```
static inline struct ibv_qp *
ibv_exp_create_qp(struct ibv_context *context,
                  struct ibv_exp_qp_init_attr *qp_init_attr);
struct ibv_cq *ibv_create_cq(struct ibv_context *context,
                             int cqe, void *cq_context,
                             struct ibv_comp_channel *channel,
                             int comp_vector);

static inline int ibv_exp_modify_qp(struct ibv_qp *qp,
                                    struct ibv_exp_qp_attr *attr,
                                    uint64_t exp_attr_mask);
static inline int ibv_exp_modify_cq(struct ibv_cq *cq,
                                    struct ibv_exp_cq_attr *cq_attr,
                                    int cq_attr_mask);

int ibv_destroy_qp(struct ibv_qp *qp);
int ibv_destroy_cq(struct ibv_cq *cq);
```

Queue Pairs (QPs) are where client Work Requests (WRs) are posted, and Completion Queues (CQs) are where their results will be pushed by hardware. In hyperloop, experimental features are required, hence functions with `_exp_` are used.

The experimental feature for QPs that Hyperloop uses is *cross channel* and *ignore overflow* [33]:

- Cross Channel: Required to post WAIT WRs; Posting a WAIT WR to a QP without this feature will return `EINVAL`. `IBV_EXP_QP_CREATE_CROSS_CHANNEL` indicates this feature.
- Ignore Overflow: Posting too many WRs that exceeds the capacity without polling WCs will return `ENOMEM`. `IBV_EXP_QP_CREATE_IGNORE_SQ_OVERFLOW` and `IBV_EXP_QP_CREATE_IGNORE_RQ_OVERFLOW` are used for SQs and RQs, respectively, to prevent an error from occuring without handling WCs.

WAIT WRs wait a CQ that is connected to the other QP, hence it is the reason that this feature is called *cross channel*. Refer to Appendix D. in [23] for more details about cross channel communication.

For CQs, *ignore overrun* experimental feature is used, a similar one to ignore overflow in the QP. Once a CQ becomes full then additional WCs are discarded, WAIT WRs that depends on newly completed WCs do not work. To prevent this problem, old WCs need to be flushed, but Hyperloop does not

poll CQs so that WAIT WRs should be stuck. To prevent it, ignore overrun is introduced in CQs to overwrite CQs so that events can be delivered to WAIT WRs even in a full condition.

Unlike QP flags that can be used in *creating* a QP (the flags contain a term _CREATE_ that indicates these flags are used in QP creation), The CQ flag `IBV_EXP_CQ_IGNORE_OVERRUN` is used in modifying a CQ.

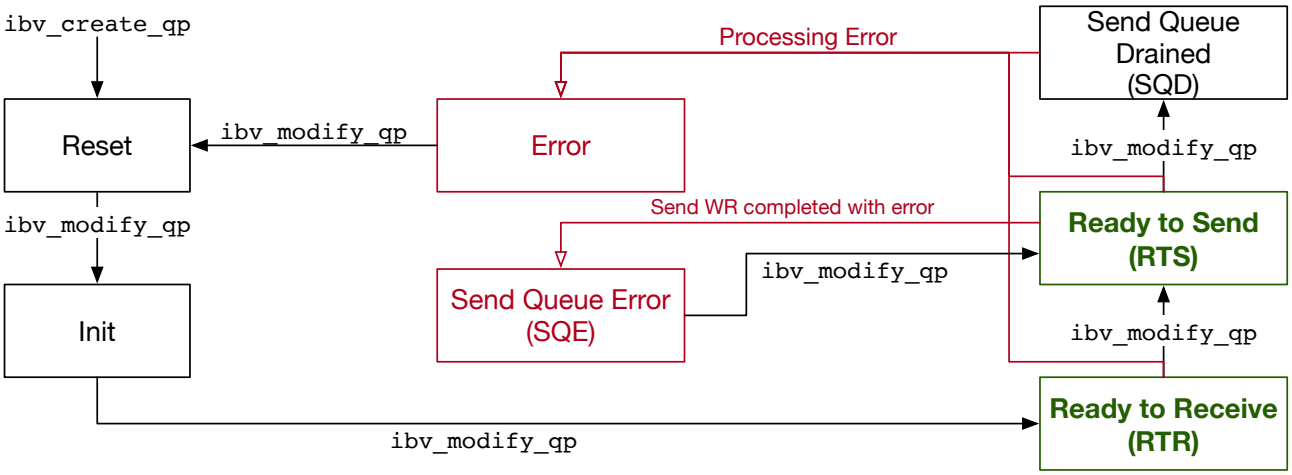
```
1  static struct ibv_qp* createInfinibandQueuePair(struct ibv_context& context,
2                                                   struct ibv_pd& pd,
3                                                   struct ibv_cq& rcq,
4                                                   struct ibv_cq& scq) {
5    struct ibv_exp_qp_init_attr qp_init_attr;
6    memset(&qp_init_attr, 0, sizeof(qp_init_attr));
7    qp_init_attr.qp_type = IBV_QPT_RC;
8    // sg_sig_all = 0 means that in every WR posted to the SQ,
9    // the user must decide whether to generate a Work Completion.
10   qp_init_attr.sq_sig_all = 0;
11   qp_init_attr.send_cq = &scq;
12   qp_init_attr.recv_cq = &rcq;
13   qp_init_attr.cap.max_send_wr = max_wr_size;
14   qp_init_attr.cap.max_recv_wr = max_wr_size;
15   qp_init_attr.cap.max_send_sge = 5;
16   qp_init_attr.cap.max_recv_sge = 5;
17   qp_init_attr.exp_create_flags = IBV_EXP_QP_CREATE_CROSS_CHANNEL |
18                                   IBV_EXP_QP_CREATE_IGNORE_SQ_OVERFLOW |
19                                   IBV_EXP_QP_CREATE_IGNORE_RQ_OVERFLOW;
20   qp_init_attr.comp_mask =
21         IBV_EXP_QP_INIT_ATTR_CREATE_FLAGS | IBV_EXP_QP_INIT_ATTR_PD;
22   qp_init_attr.pd = &pd;
23
24   return ibv_exp_create_qp(&context, &qp_init_attr);
25 }
26
27 static ibv_cq* createCompletionQueue(struct ibv_context& context) {
28   auto cq = ibv_create_cq(&context, max_wr_size, nullptr, nullptr, 0);
29
30   ibv_exp_cq_attr cq_attr;
31   memset(&cq_attr, 0, sizeof(cq_attr));
32
33   cq_attr.cq_cap_flags = IBV_EXP_CQ_IGNORE_OVERRUN;
34   cq_attr.comp_mask = IBV_EXP_CQ_CAP_FLAGS;
35
36   ibv_exp_modify_cq(cq, &cq_attr, IBV_EXP_CQ_CAP_FLAGS);
37   return cq;
38 }
```

Listing 2.13: Source code from our Hyperloop implementation.

2.2.2.1 QP States

A QP cannot be used right after creation, but needs to be connected to a peer QP. `libibverbs` manages the status of QPs with QP state machine, and Figure 2.7 illustrates it. All Hyperloop QPs remain the state Ready-to-Send (RTS), so that they can send a request to their peer QP.

Figure 2.7: Queue Pair State Machine Diagram [2]

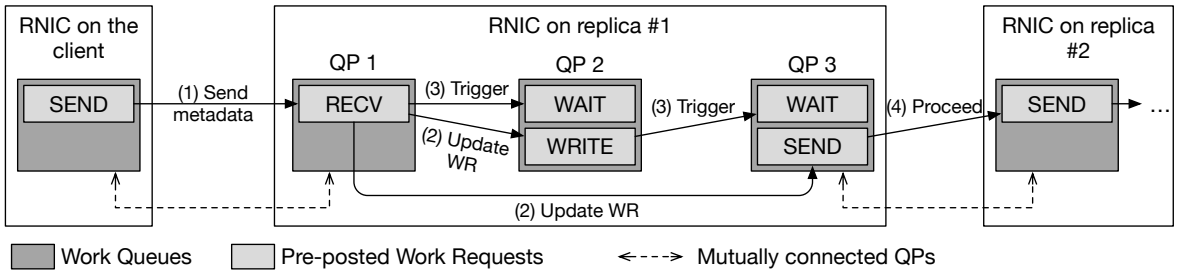



```
1  bool QueuePair::connect(const QueueIdentifier& peer_id) {
2    struct ibv_qp_attr init_attr; memset(&init_attr, 0, sizeof(init_attr));
3    init_attr.qp_state = ibv_qp_state::IBV_QPS_INIT;
4    init_attr.port_num = device_port_;
5    init_attr.pkey_index = 0;
6    init_attr.qp_access_flags = IBV_ACCESS_LOCAL_WRITE | IBV_ACCESS_REMOTE_READ |
7                                IBV_ACCESS_REMOTE_WRITE |
8                                IBV_ACCESS_REMOTE_ATOMIC;
9
10   ibv_modify_qp(
11       queue_pair_.get(), &init_attr,
12       IBV_QP_STATE | IBV_QP_PKEY_INDEX | IBV_QP_PORT | IBV_QP_ACCESS_FLAGS);
13
14   struct ibv_qp_attr rtr_attr; memset(&rtr_attr, 0, sizeof(rtr_attr));
15   rtr_attr.qp_state = ibv_qp_state::IBV_QPS_RTR;
16   rtr_attr.path_mtu = ibv_mtu::IBV_MTU_1024;
17   rtr_attr.ah_attr.port_num = device_port_;
18   rtr_attr.dest_qp_num = peer_id.qp_number;
19   rtr_attr.ah_attr.dlid = peer_id.local_id;
20   // some attributes are omitted...
21
22   ibv_modify_qp(queue_pair_.get(), &rtr_attr,
23                 IBV_QP_STATE | IBV_QP_AV | IBV_QP_PATH_MTU | IBV_QP_DEST_QPN |
24                     IBV_QP_RQ_PSN | IBV_QP_MAX_DEST_RD_ATOMIC |
25                     IBV_QP_MIN_RNR_TIMER);
26
27   struct ibv_qp_attr rts_attr; memset(&rts_attr, 0, sizeof(rts_attr));
28   rts_attr.qp_state = ibv_qp_state::IBV_QPS_RTS;
29   rts_attr.max_rd_atomic = 0;
30   // some attributes are omitted...
31
32   ibv_modify_qp(queue_pair_.get(), &rts_attr,
33                 IBV_QP_STATE | IBV_QP_TIMEOUT | IBV_QP_RETRY_CNT |
34                     IBV_QP_RNR_RETRY | IBV_QP_SQ_PSN | IBV_QP_MAX_QP_RD_ATOMIC);
35   return true;
36 }
```

Listing 2.14: Source code from our Hyperloop implementation.

A newly created QP must change its state from RESET to INIT, RTR, and then to RTS. For each state

Figure 2.8: Datapath of gMEMCPY primitive operation in Hyperloop [21].



transition, not only its desired state, but also other attributes should be modified altogether. The third argument of `ibv_modify_qp` is the attribute mask that which argument we want to modify during QP state transition. Required attributes to be modified are well explained in Section 3.5 in [23].

One important attribute regarding performance is `max_rd_atomic`. This attribute is given to the RNIC when transiting from RTR to RTS, and set as 0 in code 2.14. According to the manual, the value of `max_rd_atomic` represents *the number of outstanding RDMA reads and atomic operations that can be handled by this QP as an initiator*. It seems to say how many operations can be handled at the same time, indicating the number of hardware queues. However, there is no explanation what the value 0 means. We think that it means *the maximum value as much as possible*, since our experiment with `max_rd_atomic=0` shows similar throughput with another one with the maximum possible value. You can use `ibv_devinfo` to find out the maximum possible value of `max_rd_atomic`:

```
$ ibv_devinfo -d mlx5_1 -v
hca_id: mlx5_1
        transport:          InfiniBand (0)
        fw_ver:             18.27.2008
        node_guid:          b859:9f03:00df:0d1a
        sys_image_guid:     b859:9f03:00df:0d1a
        vendor_id:          0x02c9
        vendor_part_id:     41682
        ...
        max_qp_rd_atom:     16
        ...
```

2.2.2.2 Connecting QP to Itself

Conventionally a QP is connected to some another QP, so that they can communicate with each other. However, in Hyperloop, a special type of connection is required for its primitive operations. `gMEMCPY`, for instance, is a Hyperloop primitive operation that uses three QPs as illustrated in Figure 2.8. QP1 in the replica #1 is connected to the QP in the client, and QP3 in the replica #1 is connected to the QP1 in the replica #2. However, we are not sure which one QP2 in the replica#1 should connect to, despite of the fact that it should be connected to some QP to post WRs (WAIT and RDMA WRITE), requiring the QP's state to be RTS. To make the QP2 be able to handle WRs and accessible to its local node, it should connect to itself. It does not work normally if it is connected to other QPs, such as QP1 or QP3, even though it does not return any error during QP state transition.

2.2.2.3 Configuring RDMA over Converged Ethernet (RoCE)

For RNICs that uses Ethernet as a media, which we call RoCE, further configruations are required when connecting QPs. As it uses Ethernet, we need to provide some information regarding *global routing*. Global Routing Header (GRH) is used for using RoCE. This flag must be set for RoCE, and the RNIC would see `ah_attr` attributes to configure RoCE.

```
bool QueuePair::connect(const QueueIdentifier& peer_id) {
  ...
  rtr_attr.ah_attr.is_global = 1;
  rtr_attr.ah_attr.port_num = device_port_;
  rtr_attr.ah_attr.grh.dgid = peer_id.gid;
  rtr_attr.ah_attr.grh.sgid_index = gid_index;
  rtr_attr.ah_attr.grh.flow_label = 0;
  rtr_attr.ah_attr.grh.hop_limit = 10;
  rtr_attr.ah_attr.grh.traffic_class = 0;

  ibv_modify_qp(queue_pair_.get(), &rtr_attr,
                IBV_QP_STATE | IBV_QP_AV | IBV_QP_PATH_MTU | IBV_QP_DEST_QPN |
                    IBV_QP_RQ_PSN | IBV_QP_MAX_DEST_RD_ATOMIC |
                    IBV_QP_MIN_RNR_TIMER);
  ...
```

Listing 2.15: Source code from our Hyperloop implementation.

In code 2.15, we set `ah_attr.is_global` to 1, indicating that the RNIC is using RoCE and `ah_attr.grh` field is valid. Variables in `ah_attr.grh` should be filled to indicate how RDMA messages should be routed via RoCE. An important variable in the `grh` is *GID index*. GID index indicates which RoCE version and an IP address that we use for communication.

Mellanox provides a script that conveniently checks which GID index we can use: *show_gids*:

```
$ show_gids
DEV     PORT INDEX GID              IPv4         VER DEV
---     ---- ----- ---------------  ---------    --- ---
mlx5_0 1    0     fe80:0000:...                 v1  enp94s0f0
mlx5_0 1    1     fe80:0000:...                 v2  enp94s0f0
mlx5_0 1    2     0000:0000:...   192.168.**    v1  enp94s0f0
mlx5_0 1    3     0000:0000:...   192.168.**    v2  enp94s0f0
mlx5_1 1    0     fe80:0000:...                 v1  enp94s0f1
mlx5_1 1    1     fe80:0000:...                 v2  enp94s0f1
```

For example, if you use GID index 3, port number 1 on the device mlx5_0 will be used with RoCEv2. A main difference between Infiniband and RoCE is flow control mechanism and congestion control mechanism; while Infiniband media are loseless based on credit based flow control, RoCE uses lossy Ethernet as media; hence Ethernet-based flow control (e.g. Priority Flow Control, PFC) and Quality of Service (e.g. Differentiated Service Code Point, DSCP) are used in RoCE. Refer to [35] for more information about RoCE.

2.2.3 Memory Regions (MRs)

```
struct ibv_mr *ibv_reg_mr(struct ibv_pd *pd, void *addr,
                          size_t length, int access);

int ibv_dereg_mr(struct ibv_mr *mr);
```

Memory region (MR) is a virtually contiguous memory block that was registered [5]. Any *physical memory buffer* in the process' virtual address space can be registered. When the process ask to register a memory buffer, the kernel module `mlx5_ib.ko` handles it by (1) pinning the pages so that data always exist in memory, (2) passing its physical address and permissions to the HCA so that it can read data from the buffer or write data to the buffer without CPU.

There are 5 permission types, which can be used together with other permissions:

- Local operations (Local Read is always supported)
  - IBV_ACCESS_LOCAL_WRITE: the HCA can write data into this registered buffer.
  - IBV_ACCESS_BW_BIND
- Remote operations
  - IBV_ACCESS_REMOTE_READ: HCAs in remote nodes can read data from the registered buffer via RDMA_READ.
  - IBV_ACCESS_REMOTE_WRITE: HCAs in remote nodes can write data into the registered buffer via RDMA_WRITE.
  - IBV_ACCESS_REMOTE_ATOMIC: HCAs in remote nodes can perform atomic operations (FAA and CAS) into the registered buffer.

```
1 bool Context::createMemoryRegion(const void *address, const size_t size) {
2   auto mr = ibv_reg_mr(protection_domain_.get(), address, size,
3                        IBV_ACCESS_LOCAL_WRITE | IBV_ACCESS_REMOTE_READ |
4                            IBV_ACCESS_REMOTE_WRITE | IBV_ACCESS_REMOTE_ATOMIC);
5   ...
6 }
```

Listing 2.16: Source code from our Hyperloop implementation.

2.2.3.1 Physical Address Memory Region (PA-MR)

Since ConnectX-4, the HCA can access *any physical memory* even that is not mapped into the process' virtual address space [34, 37]. Due to its security concerning, it is disabled by default in the kernel modules. To enable this, we need to recompile MLNX_OFED kernel modules with `--with-pa-mr` and use the result, otherwise registering PA-MR will return `EINVAL` error. Also, it requires `CAP_SYS_RAWIO` capability for access permission.

PA-MR is a feature for performance optimization [37]; traditional MRs use *virtual memory address* to register memory buffer, which should be translated to the corresponding physical memory, incuring address translation overheads (refer Section 6 in [24] regarding HCA address translation). PA-MR eliminates address translation overheads so that the HCA can access the memory faster.

However, not all physical memory address works with PA-MR. Its purpose is to access *physical memory* faster and takes a physical memory address, however, *physical memory address* is a superset of *physical memory* (Refer to Figure 2 in [14]). PCI devices are mapped to the system's physical memory address space and can be used as a way of accessing the devices (Doorbell register in Section 2.1.2.2 is an example). Those address regions not mapped to physical memory cannot be used with PA-MR.

2.2.4 Posting Work Requests (WRs)

```
static inline int ibv_exp_post_send(struct ibv_qp *qp,
                                    struct ibv_exp_send_wr *wr,
                                    struct ibv_exp_send_wr **bad_wr);
static inline int ibv_post_recv(struct ibv_qp *qp, struct ibv_recv_wr *wr,
                                struct ibv_recv_wr **bad_wr);
```

`libibverbs` provides a way of posting WRs into a WQ without user-kernel context switches, as introduced in Section 2.1. Hyperloop uses `ibv_exp_post_send` to post WRs into a SQ, since WAIT operation is currently an experimental feature.

all operations that Hyperloop uses (WAIT, SEND, RDMA WRITE, CAS, and NOP) except RECV are posted with `ibv_exp_post_send()`. The second argument `wr` contains opcode that classifies what is the type of this WR.

```
bool QueuePair::postWaitRequest(const QueuePair& qp,
                                const bool wait_recv_request,
                                const bool generate_cqe_when_done) {
  struct ibv_exp_send_wr wait_wr, *bad_wr = nullptr;
  memset(&wait_wr, 0, sizeof(wait_wr));
  wait_wr.sg_list = nullptr;
  wait_wr.num_sge = 0;
  wait_wr.exp_send_flags = IBV_EXP_SEND_WAIT_EN_LAST;
  if (generate_cqe_when_done) {
    wait_wr.exp_send_flags |= IBV_EXP_SEND_SIGNALED;
  }
  wait_wr.exp_opcode = IBV_EXP_WR_CQE_WAIT;
  wait_wr.next = nullptr;
  wait_wr.task.cqe_wait.cq = wait_recv_request ? qp.rq_completion_queue_.get()
                  : qp.sq_completion_queue_.get();
  wait_wr.task.cqe_wait.cq_count = 1;

  ibv_exp_post_send(queue_pair_.get(), &wait_wr, &bad_wr);
  return true;
}

bool QueuePair::postRDMAWriteRequest(const std::vector<MemoryIdentifier>& sge,
                                     const MemoryIdentifier& peer_memory_id,
                                     const bool generate_cqe_when_done) {
  struct ibv_exp_send_wr write_wr, *bad_wr = nullptr;
  memset(&write_wr, 0, sizeof(write_wr));

  auto write_sge = new ibv_sge[sge.size()];
  for (int i = 0; i < sge.size(); i++) {
    write_sge[i].addr = reinterpret_cast<uintptr_t>(sge[i].address);
    write_sge[i].length = sge[i].length;
    write_sge[i].lkey = sge[i].local_key;
  }
  write_wr.sg_list = write_sge;
  write_wr.num_sge = sge.size();
  write_wr.exp_opcode = IBV_EXP_WR_RDMA_WRITE;
  write_wr.exp_send_flags = (generate_cqe_when_done ? IBV_EXP_SEND_SIGNALED : 0);
  write_wr.wr.rdma.remote_addr = peer_memory_id.address;
  write_wr.wr.rdma.rkey = peer_memory_id.remote_key;
  write_wr.next = nullptr;

  ibv_exp_post_send(queue_pair_.get(), &write_wr, &bad_wr);
  delete[] write_sge;
  return true;
}
```

Listing 2.17: Implementation of some WR posting: WAIT and RDMA WRITE.

For WAIT WR, the flag `IBV_EXP_SEND_WAIT_EN_LAST` indicates that the WAIT waits *the last operation* in the target CQ; otherwise, it seems we can wait a specific CQ by giving index. And it cannot wait more than one CQ with the flag set seeing internal implementation in `libmlx5`.

```
static int __mlx5_post_send_one_rc_dc(struct ibv_exp_send_wr *wr,
                                      struct mlx5_qp *qp,
                                      uint64_t exp_send_flags,
                                      void *seg, int *total_size) {
```

```
5     ...
6     switch(wr->exp_opcode) {
7       case IBV_EXP_WR_CQE_WAIT: {
8         struct mlx5_cq *wait_cq = to_mcq(wr->task.cqe_wait.cq);
9         uint32_t wait_index = 0;
10
11        wait_index = wait_cq->wait_index +
12            wr->task.cqe_wait.cq_count;
13        wait_cq->wait_count = max(wait_cq->wait_count,
14            wr->task.cqe_wait.cq_count);
15
16        if (exp_send_flags & IBV_EXP_SEND_WAIT_EN_LAST) {
17          wait_cq->wait_index += wait_cq->wait_count;
18          wait_cq->wait_count = 0;
19        }
20
21        set_wait_en_seg(seg, wait_cq->cqn, wait_index);
22      }
23      ...
24    }
25  }
```

Listing 2.18: `libmlx5` implementation for WAIT WR in `libmlx5/src/qp.c`.

The other flag, `IBV_EXP_SEND_SIGNALED`, is used to indicate whether a WC should be generated after the HCA completed its execution. When we create a QP, we set `sq_sig_all` 0, which means not all SQE generates WCs in Listing 2.13. If it is set to 1, the HCA always generates WCs for all posted Send WRs, regardless whether Send WRs have the flag `IBV_EXP_SEND_SIGNALED` set. If it is set to 0, the HCA will only generate WCs for Send WRs posted with `IBV_EXP_SEND_SIGNALED` set. This is only applicable to Send WRs (remember that the flag starts with `sq_`), and the HCA will generate WCs for every RECV WRs, since there is no other way but using CQ for software to know the posted RECV WRs are done.

For RDMA WRITE, *Scatter/Gather Elements (sge)* are given. All WRs that send or receive data (RDMA READ, RDMA WRITE, SEND, and RECV) requires the field. This means when you want to send data (such as by RDMA WRITE), the elements work as a *gather elements*; data in those buffer are gathered and sent to the peer node, while you specify them with RDMA READ or RECV, they work as a *scatter elements*; received data will be scattered to the specified buffer. They should be specified as a continuous array of *ibv_sge* structure, and their number should be written to `num_sge` in the WR. This can be empty by specifying `sg_list` as `nullptr` and `num_sge` as 0 if you send nothing but a WR itself. It is useful when you post a SEND WR as a *notification*.

Refer to Section 3.6 in [23] to see basic explanations for the other arguments and the list of possible flags for each operation.

2.2.5 Waiting Work Completions (WCs)

```
int ibv_poll_cq(struct ibv_cq *cq, int num_entries,
                struct ibv_wc *wc);

struct ibv_comp_channel *ibv_create_comp_channel(
                                    struct ibv_context *context);
int ibv_req_notify_cq(struct ibv_cq *cq, int solicited_only);
int ibv_get_cq_event(struct ibv_comp_channel *channel,
                     struct ibv_cq **cq, void **cq_context);
void ibv_ack_cq_events(struct ibv_cq *cq, unsigned int nevents);
```

libibverbs provides two ways to wait WRs to be completed using a Completion Queue (CQ). The first way is to poll CQ using ibv_poll_cq, retrieving the result continuously.

```
bool QueuePair::pollCompletion(const unsigned int timeout_msec) {
  auto start = std::chrono::steady_clock::now();
  int64_t elapsed = 0;
  struct ibv_wc wc;
  int result;

  do {
    result = ibv_poll_cq(cq, 1, &wc);
    auto end = std::chrono::steady_clock::now();
    elapsed = std::chrono::duration_cast<std::chrono::microseconds>
                (end - start).count();
  } while (result == 0 && elapsed < timeout_msec * 1000);

  // You can get string of status by calling ibv_wc_status_str(wc.status).
  if (result == 1 && wc.status == ibv_wc_status::IBV_WC_SUCCESS) {
    return true;
  }
  return false;
}
```

Listing 2.19: Waiting a WR to be completed using CQ poll.

ibv_poll_cq returns the number of WRs that completed; as we specified the second argument (num_entries) 1, it should be either 1 (a WR completes its execution) or 0 (an error returns). It breaks iteration when either result becomes 1 (a WC is generated) or timeout expires.

Polling is not the only option to wait WCs generation. ibv_req_notify_cq provides a notification, blocked and wakes up when a new WC is generated. It is useful when the program is not latency-critical. It requires a *completion channel* to be attached to a CQ when creating it as follows. The third argument of ibv_create_cq is the completion channel *channel* that will be used to return completion events.

```
struct ibv_comp_channel* channel = ibv_create_comp_channel(context_.get());
struct ibv_cq* cq = ibv_create_cq(&context, max_wr_size, nullptr, channel, 0);

// Request notification before any WC is created.
ibv_req_notify_cq(cq, 0);

// Wait for a completion event. It will be blocked until an event is received.
// When it returns, event_cq stores a pointer to a CQ that stores the WC.
// Not clear what the third argument is.
struct ibv_cq* event_cq = nullptr;
ibv_get_cq_event(channel, &event_cq, nullptr);

// All received events must be acknowledged to prevent races.
ibv_ack_cq_events(event_cq, 1);

// Until now we received an EVENT. Poll CQ to get an actually generated WC.
struct ibv_wc wc;
int result = ibv_poll_cq(event_cq, 1, &wc);
```

Listing 2.20: Waiting a WC creation using a completion channel and a CQ event.

From Listing 2.20, the second argument of ibv_req_notify_cq indicates whether it receives an event for *any* WC or only for a *solicited* WC event [4]. You can specify an event as solicited by specifying ibv_send_wr.send_flags |= IBV_SEND_SOLICITED when posting a SEND WR. If set other than 0, only a solicited event can wake this thread up.

2.3 How to Use Modified `libibverbs` and `libmlx5`

Hyperloop requires userspace device drivers to be modified to implement its primitive operations. However, there is no tutorials to use modified device drivers and our program returns several errors when we use modified one. This section introduces how to use modified device drivers.

2.3.1 Linking Modified Libraries

Our program is linked to `libibverbs` library with the flag `-libverbs`, hence it should be no problem to use modified `libibverbs` with an additional flag `-L <path>`. However, we do not specify specific path of `libmlx4` or `libmlx5`; With modified inlinecodelibibverbs, the program will return an error:

```
libibverbs: Warning: couldn't open config directory '/usr/local/etc/libibverbs.d'
libibverbs: Warning: no userspace device-specific driver found for /sys/class/
    infiniband_verbs/uverbs1
libibverbs: Warning: no userspace device-specific driver found for /sys/class/
    infiniband_verbs/uverbs0
```

Listing 2.21: Error returned by modified libibverbs.

If you install `libibverbs` by MLNX_OFED archive, it internally builds packages from source code (i.e., `.deb` or `.rpm`) using `dpkg-buildpackage` or `rpm-build` command, and installs them by using `dpkg -i` or `yum localinstall`. `libibverbs` library installed in this way uses `/etc/libibverbs.d` directory (not `/usr/local/etc/libibverbs.d`), files in which are installed together by the package management system.

In `/etc/libibverbs.d`, there are 3 driver files with specific driver names:

```
/etc/libibverbs.d$ ls
mlx4.driver  mlx5.driver  rxe.driver
```

where each file contains a path of the low-level device-specific driver, such as for mlx5.driver:

```
/etc/libibverbs.d$ cat mlx5.driver
driver /usr/lib/libibverbs/libmlx5
```

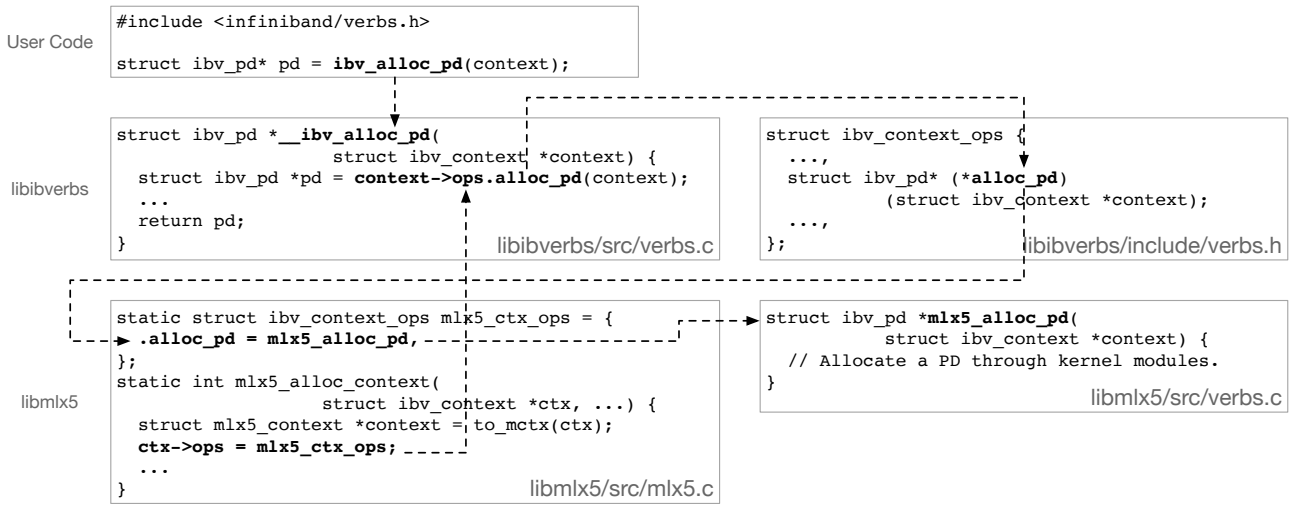
indicates that `libibverbs` uses `/usr/lib/libibverbs/libmlx5` as `libmlx5` driver. There is no files named `libmlx5` in `/usr/lib/libibverbs` however, it has `libmlx5-rdmav2.so`, a name of which is a concatenation of `libmlx5` and `-rdmav2.so`. `libmlx5-rdmav2.so` is actually a symbolic link to a real `libmlx5.so` library:

```
/usr/lib/libibverbs$ ls -l
-rw-r--r-- 1 root root libmlx4-rdmav2.so
lrwxrwxrwx 1 root root libmlx5-rdmav2.so -> ../libmlx5.so.1.0.0
```

Therefore, we can assume that our modified `libibverbs` will work with a modified `libmlx5` as follows:

1. `libibverbs` first looks at `/usr/local/etc/libibverbs.d/libmlx5`.
   Note that this is because we use connectX-4 that uses `libmlx5`.
2. The file contains a path: `driver /path/to/libmlx5/libmlx5`. Then `libibverbs` find a file at `/path/to/libmlx5/libmlx5-rdmav2.so`.
3. `/path/to/libmlx5/libmlx5-rdmav2.so` can be either a symbolic link or a regular shared library.

Figure 2.9: Userspace device driver function flow. User programs only use verbs API, that internally calls device-specific library functions.



2.3.2 Providing an Additional User Interface

While `libmlx5` actually handles requested operations, the program only uses verbs API defined in `libibverbs`. As illustrated in Figure 2.9, it is `libibverbs` that internally calls device-specific functions and the program is not directly linked to `libmlx5`. To make a new function in device-specific library, we also need to implement a new verb API in verb library.

Device-specific libraries and `libibverbs` are connected through a group of function pointers. `libibverbs` first provides the interface `struct ibv_context_ops`:

```
1  struct ibv_context_ops {
2    int      (*query_device)(struct ibv_context *context,
3                  struct ibv_device_attr *device_attr);
4    int      (*query_port)(struct ibv_context *context, uint8_t port_num,
5                  struct ibv_port_attr *port_attr);
6    struct ibv_pd *    (*alloc_pd)(struct ibv_context *context);
7    int      (*dealloc_pd)(struct ibv_pd *pd);
8    struct ibv_mr *    (*reg_mr)(struct ibv_pd *pd, void *addr, size_t length,
9              int access);
10   ...
11 };
```

Listing 2.22: Definition of `struct ibv_context_op` from `libibverbs/include/infiniband/verbs.h`.

and `libmlx5` assigns the pointers of its functions to a group and passes it to `libibverbs`.

```
1  static struct ibv_context_ops mlx5_ctx_ops = {
2    .query_device  = mlx5_query_device,
3    .query_port    = mlx5_query_port,
4    .alloc_pd      = mlx5_alloc_pd,
5    .dealloc_pd    = mlx5_free_pd,
6    .reg_mr        = mlx5_reg_mr,
7    ...
8  };
9
10 static int mlx5_alloc_context(struct verbs_device *vdev,
11                                struct ibv_context *ctx, int cmd_fd) {
12   ...
```

```
13    ctx->ops = mlx5_ctx_ops;
14    ...
15  }
16
17  static struct verbs_device *mlx5_driver_init(const char *uverbs_sys_path,
18                                               int abi_version) {
19    ...
20    /*
21     * mlx5_init_context will initialize provider calls
22     */
23    dev->verbs_dev.init_context = mlx5_alloc_context;
24    dev->verbs_dev.uninit_context = mlx5_free_context;
25    dev->verbs_dev.verbs_uninit_func = mlx5_driver_uninit;
26
27    return &dev->verbs_dev;
28  }
```

Listing 2.23: Source code from `libmlx5/src/mlx5.c`. `libmlx5` passes the pointers of functions to the upper layer.

`libibverbs` can create a context by calling `init_context`. And then, `libibverbs` can access all `libmlx5` operations via `context->ops.<function_name>`:

```
1  struct ibv_context *__ibv_open_device(struct ibv_device *device) {
2    struct ibv_context *context;
3    ...
4    ret = verbs_device->init_context(verbs_device, context, cmd_fd);
5    ...
6    return context;
7  }
```

Listing 2.24: Source code from `libibverbs/src/device.c`. At first, `libibverbs` uses `verbs_device` to call initialize provider functions.

```
1  struct ibv_pd *__ibv_alloc_pd(struct ibv_context *context)
2  {
3    struct ibv_pd *pd = context->ops.alloc_pd(context);
4    ...
5    return pd;
6  }
7  default_symver(__ibv_alloc_pd, ibv_alloc_pd);
```

Listing 2.25: Source code from `libibverbs/src/verbs.c`. After creating a context, `libibverbs` accesses device-specific functions via pointers in `context->ops`. The code above is an example of allocating a PD.

Therefore, to provide an additional verb API, you need to:

1. Add a new function pointer into `ibv_context_ops` in `libibverbs`.
2. Implement a new device-specific function in `libmlx5`.
3. Assign the pointer of the function into `mlx5_ctx_ops` in `libmlx5`.
4. Implement a verb API function in `libibverbs`.

For instance, Hyperloop need to know where the WQE buffer starts for remote work request manipulation. The following code returns the base address of the WQE buffer of the given QP.

```
struct ibv_hyperloop_wqe_buffer {
  void *address;
  size_t size;
};

// Step 1: add a new function pointer.
struct ibv_context_ops {
  ...,
  struct ibv_hyperloop_wqe_buffer (*get_wqe_buffer) (struct ibv_qp* qp);
};

// Step 4: implement a verb API.
static inline struct ibv_hyperloop_wqe_buffer
ibv_hyperloop_get_wqe_buffer (struct ibv_qp* qp) {
  return qp->context->ops.get_wqe_buffer(qp);
}
```

Listing 2.26: Code added into `libibverbs/include/infiniband/verbs.h`.

```
// Step 3: assign the pointer of the function.
static struct ibv_context_ops mlx5_ctx_ops = {
  ...,
  .get_wqe_buffer = mlx5_hyperloop_get_wqe_buffer,
};

// Step 2: implement a new device-specific function.
struct ibv_hyperloop_wqe_buffer
mlx5_hyperloop_get_wqe_buffer(struct ibv_qp* ibqp) {
  struct mlx5_qp* qp = to_mqp(ibqp);
  struct ibv_hyperloop_wqe_buffer buffer{
    .address = qp->gen_data.sqstart,
    .size = qp->gen_data.sqend - qp->gen_data.sqstart,
  };
  return buffer;
}
```

Listing 2.27: Code added into `libmlx5/src/mlx5.c`

With the modifications above, Hyperloop can get the base address of the WQE buffer:

```
auto buffer = ibv_hyperloop_get_wqe_buffer(qp);

// Register buffer.address with size=buffer.size as a MR
// so that the HCA can access the memory buffer.
ibv_reg_mr(pd, buffer.address, buffer.size,
           IBV_ACCESS_LOCAL_WRITE | IBV_ACCESS_REMOTE_WRITE);
...
```

2.4 C++ Semantics

In case a reader is not familiar with C++, we provide a basic modern C++ semantics used in the implementation for better understanding. C++14 is a C++ standard version introduced in 2014. C++ continues to integrate modern programming language features that are beneficial for better, more comfortable programming. This section contains features in C++98, C++11, and C++14.

2.4.1 C++98 Reference

```
1 int a = 5;
2 int& ar = a;
3
4 ar = 10;
5 std::cout << "a:" << a << std::endl;  // will print 10.
```

Listing 2.28: An example of C++ reference.

Reference is an alternative name for an existing variable. `<type>&` is used to indicate that it is a reference. Note that `&` in `&a` indicates an address of `a`; its meaning differs according to the location of `&`. When it is included in *type*, it indicates a reference, while it is an address if it is followed by the name of a variable. Its usage is similar to pointers, however, there are several differences between references and pointers.

1. A pointer has its own memory space to store the address of a variable. Its address is different from that of the variable it points to. However, the address of reference is the same with the original one.

```
$ cat test.cpp
int a = 5;
int& ar = a;
std::cout << &a << ", " << &ar << std::endl;
$ g++ test.cpp -o test
$ ./test
$ 0x7fff09392dcc, 0x7fff09392dcc
```

2. A reference cannot have a `NULL` value. As it is an another name of an existing variable, the variable that the reference indicates must exist. For this property, references are usually used as an argument that must not be `nullptr` (can omit `nullptr` check).

3. With a reference, accessing the value is fast. With a pointer, the machine should access the memory twice (read the value of the pointer, and access the address to read an actual value). However, a reference is just an another name of the variable, it can directly access the variable.

2.4.2 C++98 Function Overloading

```
1  // 1. Basic function
2  void example_func(int a);
3
4  // 2. possible only in C++: same function name, but different signature.
5  void example_func(double b);
6
7  // 3. neither possible in C nor C++: same signature.
8  void example_func(int b);
9
10 // 4. neither possible in C nor C++: same signature.
11 //    return type is not a part of the signature.
12 int example_func(int a);
```

Listing 2.29: An example of C++ function overloading

In C, no two functions can have the same name. In C++, it is possible if their *signature*s are different; if two functions with the same name have different parameter *types*, their signatures are different and can be used together.

```
$ cat example.cpp
void example_func (int a) {}
void example_func (double a) {}
$ g++ -O0 -o example example.cpp
$ objdump -t example
example:     file format elf64-x86-64
SYMBOL TABLE:
...
000000000000113b g     F .text  0000000000000026              _Z12example_funcd
0000000000001129 g     F .text  0000000000000012              _Z12example_funci
...
```

From a dumped ELF content, `example_func(int)` and `example_func(double)` have different function signature: `_Z12example_funci` and `_Z12example_funcd`, respectively.

This makes C++ be incompatible with C by default, and should give an indicator to the compiler that this might be compiled with C semantic:

```
1 #ifdef __cplusplus
2 extern "C" {
3 #endif
4
5   void example_func(int a);
6
7 #ifdef __cplusplus
8 }
9 #endif
```

Listing 2.30: Extern semantic is used for C/C++ compatibility.

Then, the compiler builds a symbol that C programs can link to without signature information:

```
$ objdump -t example
example:     file format elf64-x86-64

SYMBOL TABLE:
...
00000000000005fa g     F .text  000000000000000a              example_func
...
```

2.4.3 C++98 Template

A template is a C++'s implementation for generic programming. Function overloading enables programmers to use the same function name with different signatures, however, they are still required to explicitly implement functions for each combination of types of arguments. By using template, they can be templatized and only one definition is enough.

```
1 template <typename T>
2 T example_template(const T& var) { ... }
3
4 example_template<int>(4);
5 example_template<std::string>("hello");
```

Listing 2.31: An example of C++ template.

The function `example_template` is used with the types `<int>` and `<std::string>`. When a compiler compiles the program, it uses the function template to *instantiate* two corresponding functions: `int example_function (const int&)` and `std::string example_function (const std::string& var)`.

```
$ objdump -t example_template
...
_Z16example_templateIiET_RKS0_
_Z16example_templateINSt7__cxx1112basic_stringIcSt11char_traitsIcESaIcEEEET_RKS6_
...
```

The compiler **only** instantiates functions that are used in code. In other words, if the template function is not used, there can even be no instantiated function in a binary file. The compiler builds a binary with two instantiated functions with Listing 2.31.

2.4.4 C++11 Variadic Template

C++ Template enables programmers to abstract several functions, performing the same logic with different types of variables, into one template function. Still, it does not provide a flexible way to handle arbitary number of arguments, like a variadic function `printf()` in C.

```
1 int printf(const char* fmt, ...) {
2   va_list args;
3   va_start(args, fmt);
4 }
```

Listing 2.32: An example of variadic argument in C.

Hence, a modern C++ way of handling variadic arguments is required, and a template parameter pack is the answer.

```
1 template <typename... Types>
2 void f(Types... args) {}
3 f()        // OK: args contains no arguments
4 f(1)       // OK: args contains one argument: int
5 f(2, 1.0); // OK: args containers two arguments: int and double
```

Listing 2.33: An example of C++ variadic template.

Usually, a variadic template is used to pass arbitrary number of function arguments. Although the signature of the function is fixed at compile time, they can packed into a parameter pack and passed altogether for function call without duplicated definition.

```
1 auto f = [](int a, int b, int c) -> int { return a + b + c; }
2
3 template <typename T, typename... Args>
4 auto pass_func(T&& func, Args&&... args) {
5   return func(args...);
6 }
7
8 pass_func(f, 2, 3, 4);  // will return 9.
```

2.4.5 C++11 Smart Pointers

A pointer is a powerful but dangerous feature of C. For backward compatibility with C, C++ also uses the pointers. However, the pointer is so dangerous that many C-based programs are faulted due to abuse of pointers, such as accessing a memory location that is already freed or memory leak, etc.

To prevent pointer problems, four types of smart pointers are introduced in C++11: `std::shared_ptr`, `std::unique_ptr`, `std::weak_ptr`, and `std::auto_ptr` (`std::auto_ptr` has been removed since C++17). Here, we only introduce the unique pointer and the shared pointer.

2.4.5.1 Unique Pointer (`std::unique_ptr`)

`std::unique_ptr` is a C++ class that contains a raw pointer with *ownership*. In C, there is no explicit statement who should free an allocated memory for the pointer. With the unique pointer, it cannot be copied and the holder of `std::unique_ptr` must release the allocated memory before being released. When another entity that does not own the unique pointer wants to access it, the owner can pass a raw pointer returned from `unique_ptr::get()`, but semantically it does not have the ownership for the pointer, it should not free it.

```
1 std::unique_ptr<int> up1 = std::make_unique<int>(4);
2 // The pointer is moved to up2 and up1 contains a nullptr.
3 std::unique_ptr<int> up2 = std::move(up1);
4 // Will invoke a segmentation fault.
5 std::cout << *up1 << std::endl;
```

Listing 2.34: An example of moving the ownership of a unique pointter.

If the owner wants to *transfer* the ownership, we can use `std::move` semantic. move semantic is also introduced in C++11 with the smart pointers, and is used to move the ownership if it is used with the smart pointers. Once the ownership is moved, the existing unique pointer is invalidated.

2.4.5.2 Shared Pointer (`std::shared_ptr`)

`std::shared_ptr` is a C++ class that contains a raw pointer, but also manages a reference count to prevent memory leak and presents a *shared ownership*. If a shared pointer variable is copied, the shared reference (shared by several copied shared pointer variables) is increased by one.

```
1  class A {
2    A (std::shared_ptr<int> ptr) { ptr_ = ptr; }
3    std::shared_ptr<int> ptr_;
4  };
5
6  int main() {
7    std::shared_ptr data = std::make_shared<int>(4);
8    A* a1 = new A(data);  // ref count = 2
9    A* a2 = new A(data);  // ref count = 3
10   free(a1);             // ref count = 2
11   data.reset();         // ref count = 1
12   free(a2)              // ref count = 0: pointer is automatically freed.
13 }
```

Listing 2.35: An example of the shared pointer.

2.4.6 C++11 Lambda Expression

```
1 auto lmbda = [&](const int a, const int b) -> int {
2   return a + b;
3 }
4
5 auto result = lmbda(4, 5); // result is int type, and has value 9.
```

Listing 2.36: An example of a lambda expression.

A lambda expression is a useful way of defining an *anonymous function object or closure*. A function should no longer be globally defined in the code, but can exist as a function object variable. Or, even can be passed directly without any name, reducing the number of variable definitions.

```
int a = 4;
int b = 6;
[&]() -> int {
  a = 0;
  return a + b;
}();    // immediately call the function; it returns 6. a will be modified to 0.
```

Listing 2.37: An example of using captures in lambda expressions.

What makes C++ lambda expressions special is captures; captured variables are located between `[]`, and those can be captured by value or captured by reference. The example above captures `a` and `b` that are used in the lambda function by reference (not by value), so that modifying variables outside the lambda is possible.

2.4.7 C++11 Type Inference and `auto` Semantic

```
auto a = 4.0;   // 4.0 is a double literal; a will be type double.
int i = 3;      // i is an integer variable;
auto& ir = i;   // auto will be deduced as type int and auto& becomes int&.
```

Listing 2.38: An example of auto type inference.

With C++11, the compiler now can automatically *infer or deduce* the type of a variable, and let programmers use *auto* type.

```
// auto is deduced as int (*)(int).
auto func_incr = [](int a) -> int { return a + 1;}
func_incr(4); // will return 5.
```

Listing 2.39: An example of using auto type to define a function.

It can also be used to define a function object, not only to define the type of variables. In the case above, the type of `func_incr` will be a function pointer: `int (*)(int)`.

2.4.8 Examples of C++ Usage in Hyperloop Implementation

2.4.8.1 TCP Message Send and Receive

```
class Channel {
  template <typename T>
  bool sendMessage(const T& message) {
    ::send(socket_, &message, sizeof(message), 0);
    return true;
  };

  template <typename T>
  bool sendMessage(const T buffer[], const unsigned int num) {
    ::send(socket_, buffer, sizeof(T) * num, 0);
    return true;
  }
}

channel.sendMessage(4);                          // OK. Send integer 4.
double da[4] = ...; channel.sendMessage(da, 4); // OK. Send 4 double variables.
```

Listing 2.40: Source code from common/tcp/channel.h.

Sending and receiving messages through the TCP channel is done through *a stream*; they do not care which type the stream has, but just regard it as a byte stream. In other words, we can send and receive anything; in this case, C++ template is useful to cover all possible types in one template function, replacing a dangerous `void*`. Also, two functions have the same name `sendMessage`, but it is allowed via function overloading with different signatures.

2.4.8.2 Measuring Time for Function Call

```
1  template <class F, class... Args>
2  unsigned long measureTime(F&& f, Args&&... args) {
3    auto start = std::chrono::steady_clock::now();
4    f(args...);
5    auto end = std::chrono::steady_clock::now();
6    return std::chrono::duration_cast<std::chrono::microseconds>(end - start)
7           .count();
8  }
9
10 int time_consuming_function(int a, double, b) { ... }
11 measureTime(time_consuming_function, 4, 5.0);    // OK.
12 measureTime([](int a, double b){ ... }, 3, 4.0); // OK.
```

Listing 2.41: Source code from libhyperclient/time.hpp.

The function is more advanced that uses a variadic template. As it should take any function that a programmer wants to measure elapsed time, a template and a parameter pack is a perfect case to be used. The programmer can simply pass a function pointer and the list of arguments used for function call to `measureTime`. Also, lambda expressions can be used for the function, enhancing the code readability.

Chapter 3
Hyperloop Implementation

This chapter illustrates our implementation of Hyperloop and how to use it. Our Hyperloop implementation uses C++14.

3.1 Implementation Details

Our Hyperloop implementation consists of two libraries for server nodes and clients, respectively, and two executable binaries using those libraries. Figure 3.1 illustrates how each submodules communicate and how software stack is implemented.

- `hyperloop/libhyperloop`: the library code for server nodes.
- `hyperloop/libhyperclient`: the library code for clients.
- `hyperloop/apps/server`: a Hyperloop server node program that uses libhyperloop.
- `hyperloop/apps/client`: an example Hyperloop client program that uses libhyperclient.

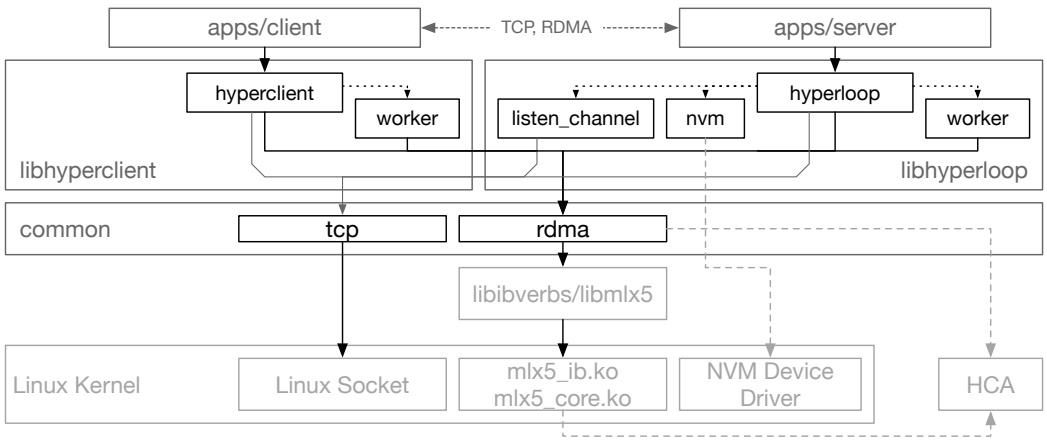
Other directories than `apps`, `libhyperloop`, and `libhyperclient` are for:

- `hyperloop/third_party`: contains third-party libraries that our code depends on.
- `hyperloop/common`: common code that both `libhyperloop` and `libhyperclient` use.

where `third_party` directory contains the following third-party libraries:

- cxxopts (MIT): C++ command line option parser. Used by apps to parse several options for Hyperloop operation.

Figure 3.1: Hyperloop implementation overview.



- hexdump (BSD): C++ hexdump formatter [12]. Used to implement a test function that prints NVM buffer.
- loguru (No license): C++ logging library [16]. Used for thread-safe stdout logging.
- readerwriterqueue (BSD): C++ single-producer, single-consumer lock-free queue [15]. Used for internal communication between two threads in Hyperclient.
- threadpool (zlib): C++11 thread pool implementation [26]. Hyperclient uses two threads for operations, and it is used to implement an efficient parallel execution.

3.1.1 `common/rdma`

This submodule provides several classes regarding RDMA that wrap `libibverbs` functions. Functions in `libibverbs` require user-specified attributes to be given, hence our class methods initialize proper attributes and calls them.

We implemented two major classes: `Context` and `QueuePair` for RDMA operations. These classes can be used by both `libhyperloop` and `libhyperclient` and should be consistent, which is why they are managed in the `common` package.

3.1.1.1 `Context` class

```
1  class Context {
2    /* Functions that creates dependent objects */
3    uint32_t createQueuePair();
4    bool createMemoryRegion(const uintptr_t address, const size_t size);
5    bool removeMemoryRegion(const uintptr_t address);
6
7    /* Getter functions */
8    struct ibv_context* getContext() const;
9    struct ibv_pd* getProtectionDomain() const;
10   struct ibv_mr* getMemoryRegion(const uintptr_t address) const;
11   QueuePair* getQueuePair(const uint32_t qp_num) const;
12
13   /* Private member fields managed by the class */
14   const std::string device_name_;
15   const int device_port_;
16   std::unique_ptr<struct ibv_context> context_;
17   std::unique_ptr<struct ibv_pd> protection_domain_;
18   std::map<uintptr_t, std::unique_ptr<struct ibv_mr>> memory_regions_;
19   std::map<uint32_t, std::unique_ptr<QueuePair>> queue_pairs_;
20 };
```

Listing 3.1: Context class in `common/rdma/context.h`. It manages a context, a protection domain, which are global objects that exist only one instance per context, and memory regions and queue pairs that depend on the context.

`Context` manages global objects, an Infiniband Context (`struct ibv_context*`) and a Protection Domain (PD, `struct ibv_pd*`). Those structures are initialized in `Context`'s constructor when an instance is created. It also manages `ibv_mr` and `QueuePair`, objects that depend on the global objects (`QueuePair` is a class that contains CQs and a QP, which will be discussed below). There is no class containing a memory region since there is no additional operations except creation and destruction of the object, while `QueuePair` class contains several operations so that it is implemented as an additional class.

Objects managed by `Context` are automatically released and deallocated in its destructor when a `Context` instance is about to be destroyed.

3.1.1.2 `QueuePair` class

```
1  class QueuePair {
2    /* Functions related to QueuePair itself */
3    bool connect(const QueueIdentifier& peer_id);
4    QueueIdentifier getQueueIdentifier() const;
5    uint32_t getQueuePairNumber() const;
6
7    /* Functions related to Work Requests */
8    bool postWaitRequest(const QueuePair& wait_qp,
9                         const bool wait_recv_request,
10                        const bool receive_cqe_when_done);
11   bool postRDMAWriteRequest(const std::vector<MemoryIdentifier>& sge,
12                             const MemoryIdentifier& peer_memory_id,
13                             const bool receive_cqe_when_done);
14   bool postRDMAReadRequest(const std::vector<MemoryIdentifier>& sge,
15                            const MemoryIdentifier& memory_id,
16                            const bool receive_cqe_when_done);
17   bool postSendRequest(const std::vector<MemoryIdentifier>& sge,
18                        const bool receive_cqe_when_done);
19   bool postCASRequest(const MemoryIdentifier& result_memory_id,
20                       const MemoryIdentifier& target_memory_id,
21                       const uint64_t compare_value,
22                       const uint64_t new_value,
23                       const bool receive_cqe_when_done);
24   bool postNopRequest(const bool receive_cqe_when_done);
25   bool postReceiveRequest(const std::vector<MemoryIdentifier>& sge);
26   bool pollCompletion(const bool wait_recv, const unsigned int timeout_msec);
27
28   /* Private member fields managed by the class */
29   struct ibv_cq* rq_completion_queue_;
30   struct ibv_cq* sq_completion_queue_;
31   struct ibv_qp* queue_pair_;
32 };
```

Listing 3.2: QueuePair class in `common/rdma/queue_pair.h`. It provides Queue Pair related functions: posting WRs, polling a CQ, etc.

`QueuePair` is a class that has a QP, the corresponding CQs, and several operations for those queues.

As described in Section 2.2.2.1, a QP needs to connect to another QP, and `QueuePair::connect()` is used for this purpose. We need to know which QP we want for the QP connect to. `QueueIdentifier` is a structure representing a QP identity, defined in `common/rdma/identifiers.h`:

```
struct QueueIdentifier {
  uint32_t qp_number;
  uint16_t local_id;
  ibv_gid gid;
};
```

where `local_id` represents a node's local ID, and `qp_number` indicates a QP ID in the node; a combination of the two numbers uniquely identifies a queue in the network.

And gid is only used in RDMA over Converged Ethernet (RoCE, Refer to Section 2.2.2.3).

Local ID for this node and gid for the given gid index can be given by using `ibv_query_port()` libibverbs API:

```
inline QueueIdentifier createQueueIdentifier(
          struct ibv_context& context,
          const int device_port,
          const struct ibv_qp& qp) {
  ibv_port_attr port_attr;
  ibv_query_port(&context, device_port, &port_attr);

  ibv_gid gid;
  ibv_query_gid(&context, device_port, gid_index, &gid);
  return {qp.qp_num, port_attr.lid, gud};
}
```

Listing 3.3: local ID and gid can be queried from the HCA device port by using `ibv_query_port`.

When posting a Work Request (WR), we need to specify which memory buffer should be used for data communication, i.e., as *sge* (Refer to Section 2.2.4). Each Scatter/Gather element is represented as `MemoryIdentifier` in our implementation, defined in `common/rdma/identifiers.h`:

```
struct MemoryIdentifier {
  uint64_t address;
  uint64_t length;
  uint32_t local_key;
  uint32_t remote_key;
};
```

With the definition above, Scatther/Gather elements are represented as a vector of those: `std::vector<MemoryIdentifier>`. Member functions for RDMA READ, RDMA WRITE, SEND, or RECV WRs take it as one of their arguments.

Functions posting SEND-related WRs (WAIT, RDMA READ, RDMA WRITE, SEND, CAS, and NOP) all take the argument `receive_cqe_when_done`. When this is set *true*, a Work Completion (WC) is generated and pushed into the Compeletion Queue for SQ (`sq_completion_queue_`). Internally, when it is set true, a flag `IBV_EXP_SEND_SIGNALED` is set in a WR. After the posted work is done, the HCA generates the corresponding WC into the CQ to notify the software that the work is done (Refer to Section 2.2.4). WCs for RECV operations will always be generated and pushed into `rq_completion_queue_`.

Compare and Swap is the most complex function [5]. It first fetches 64-bit data from `target_memory_id` in this node. If the data equals to `compare_value`, it swaps the data with `new_value`; otherwise, does nothing. Regardless of whether a swap happens, the fetched 64-bit data is stored into the buffer `result_memory_id`.

3.1.2 common/tcp

3.1.2.1 Channel class

```
class Channel {
    explicit Channel(const int socket);

    template <typename T>
    T receiveMessage();
    template <typename T>
    void receiveMessage(T buffer[], const unsigned int num);
    template <typename T>
    bool sendMessage(const T& message);
    template <typename T>
```

```
11      bool sendMessage(const T buffer[], const unsigned int num);
12
13      int socket_;
14    };
```

Listing 3.4: Channel class in `common/tcp/channel.h`. It provides a basic send, receive, and connect functions. Listening is a server-specific function, hence it is not in the `common` package and another class is implemented in `libhyperloop` for this purpose.

Sending and receiving messages use *C++ template*: `template <typename T>` for generic arguments. For instance, calling `receiveMessage<int>()` will return `int`, because typename T is deduced to `int`, hence the signature of the instantiated function is `int receiveMessage()`.

There are two functions with the same name `receiveMessage` and `sendMessage` using *function overloading* in C++. The second function, with the arguments `T buffer[], const unsigned int num`, is for optimized burst TCP send/receive.

```
1   template <typename T>
2   bool Channel::sendMessage(const T buffer[], const unsigned int num) {
3     ::send(socket_, buffer, sizeof(T) * num, 0);
4     return true;
5   }
```

Listing 3.5: `sendMessage` for bulk data. The number of `::send()` calls is reduced.

Instead of calling `::send()` (or `::recv()` `num` number of times, it treats the given array as a byte stream and sends them at once (`sizeof(T) * num` bytes). For data receive, it may call `::recv()` some number of times due to Linux kernel's limited receive buffer, but still enough to small number of times (1 call for 64K data each).

Note that a TCP latency optimization is applied here: use `TCP_NODELAY` for all sockets [19].

```
int one = 1;
setsockopt(socket, SOL_TCP, TCP_NODELAY, &one, sizeof(one));
```

Listing 3.6: Use `TCP_NODELAY` to reduce small write latency.

Without the option, we observed 40ms of delay for sending or receiving small TCP packets that incurs huge overheads in Hyperloop circumstance. While this option slightly reduced TCP throughput, its impact was negligible.

3.1.3 `libhyperloop`

`libhyperloop` is a core package of server replica. It contains several classes based on common package that do actual Hyperloop works. Four classes exist: `hyperloop`, `nvm`, `listen_channel`, and `worker`.

3.1.3.1 `ListenChannel` class

```
1  class ListenChannel {
2    explicit ListenChannel(const int listen_port);
3    tcp::Channel* acceptConnection();
4
5    int listen_socket_;
6  };
```

Listing 3.7: ListenChannel class in `libhyperloop/listen_channel.h`.

ListenChannel class provides a TCP server functionality. It binds a socket on a defined port, and creates a `common::tcp::Channel` instance for each established connection. As a prototype implementation, however, it does not provide Hyperloop service to several clients at the same time; another connection will be accepted only after the existing connection is destroyed.

3.1.3.2 `NonVolatileMemory` class

```
class NonVolatileMemory {
  NonVolatileMemory(struct ibv_mr* nvm_mr,
                    const uintptr_t address,
                    const size_t size);
  rdma::MemoryIdentifier getMemoryIdentifier(const uint64_t offset,
                                             const size_t length);

  struct ibv_mr* nvm_mr_;
  const uintptr_t address_;
  const size_t size_;
};
```

In the current version, `NonVolatileMemory` class is just a wrapper of the given memory buffer: `uintptr_t address`. The address is initialized outside of `libhyperloop` to provide more flexibility for buffer allocation. Users do not have to initialize the first argument `struct ibv_mr`; this class is internally managed by the Hyperloop master.

3.1.3.3 `Worker` class

```
class Worker {
  /* Software queue related functions and variables */
  void enqueueOperations(const rdma::Operation ops[],
                         const unsigned int op_num);
  unsigned int postOperations(unsigned int num_operation);
  moodycamel::ReaderWriterQueue<rdma::Operation> queue_;
  unsigned int num_queue_items_;

  /* RDMA QP related functions and variables */
  bool postGWrite(const rdma::GroupWrite& data);
  bool postGMemcopy(const rdma::GroupMemoryCopy& data);
  bool postGCompareAndSwap(const rdma::GroupCompareAndSwap& data);
  bool postGFlush(const rdma::GroupFlush& data);

  std::shared_ptr<rdma::Context> context_;
  std::shared_ptr<NonVolatileMemory> nvm_;
  unsigned int node_index_;
  rdma::MemoryIdentifier peer_nvm_id_;
  rdma::QueuePair* qps_[3];
  uint64_t gcas_result_;
  ibv_mr* gcas_result_mr_;
};
```

Listing 3.8: Worker class. It has a software queue and RDMA QPs to handle RDMA operations.

Worker class is the owner of `QueuePair` class instances and handles all RDMA operations. The Hyperloop master uses the Worker as the abstraction of RDMA operations.

```
enum class OperationType : unsigned int {
  GroupWrite = 0,
  GroupMemoryCopy,
```

```
4     GroupFlush,
5     GroupCompareAndSwap
6   };
7
8   struct Operation {
9     OperationType type;
10    union {
11      GroupWrite gWrite;
12      GroupMemoryCopy gMemcpy;
13      GroupFlush gFlush;
14      GroupCompareAndSwap gCas;
15    } op; /* contains required information for each ops */
16  };
17
18  void Worker::enqueueOperations(const rdma::Operation ops[],
19                                 const unsigned int op_num) {
20    for (int i = 0; i < op_num; i++) {
21      queue_.enqueue(ops[i]);
22    }
23  };
24
25  unsigned int Worker::postOperations(const unsigned int num_operation) {
26    rdma::Operation op;
27    for (int i = 0; i < num_operation; i++) {
28      queue_.try_dequeue(op);
29
30      if (op.type == rdma::OperationType::GroupWrite) {
31        postGWrite(op.op.gWrite);
32      } else if (op.type == rdma::OperationType::GroupMemoryCopy) {
33        postGMemcopy(op.op.gMemcpy);
34      } else if (op.type == rdma::OperationType::GroupCompareAndSwap) {
35        postGCompareAndSwap(op.op.gCas);
36      } else if (op.type == rdma::OperationType::GroupFlush) {
37        postGFlush(op.op.gFlush);
38      }
39    }
40
41    return num_operation;
42  }
```

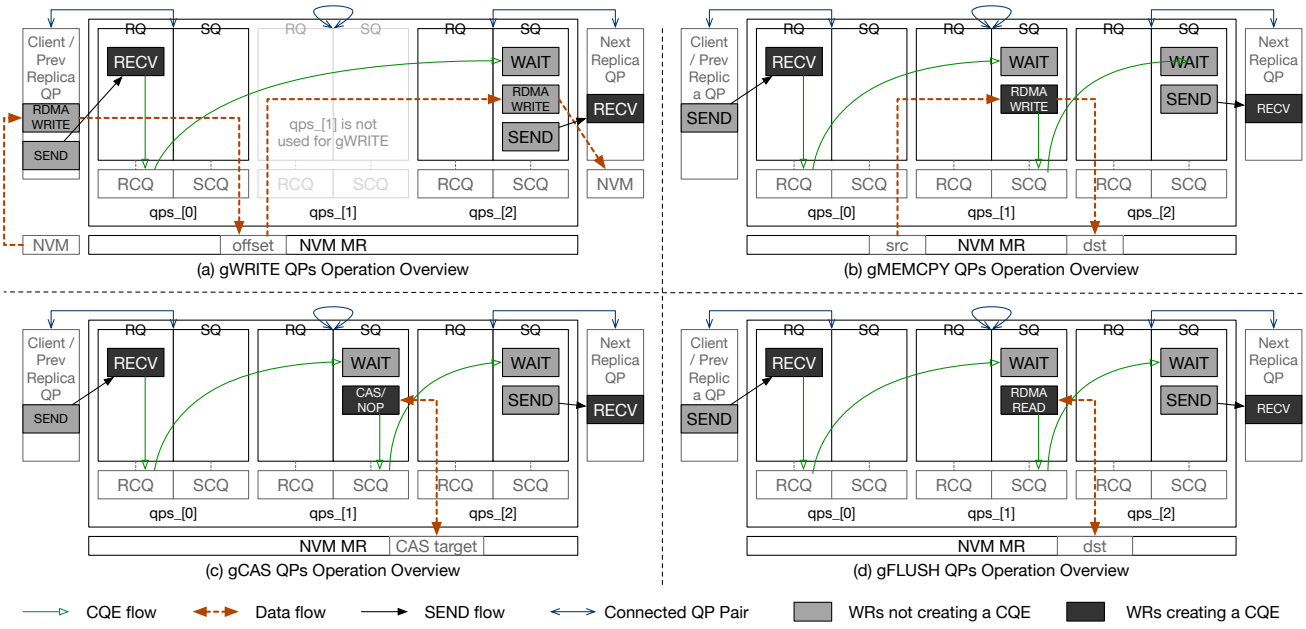
Listing 3.9: Implementation of `Worker::enqueueOperations`. Hyperloop operations are abstracted into `rdma::Operation`.

All Hyperloop operations are encapsulated into `rdma::Operations`, and the master calls `Worker::enqueueOperations` to push Hyperloop operations into the Worker's software queue `queue_`. The software queue `queue_` is not an RDMA QP and is used to temporaily store Hyperloop operations. Then, the Hyperloop master calls `Worker::postOperations` to post actual corresponding RDMA WRs into RDMA QPs.

We faced several problems regarding *remote work request manipulation* and could not implement them; therefore we implement **a simulated Hyperloop: receiving operations via the TCP, and posting them into workers in advance**. Details will be discussed in Section 4.

```
1   bool Worker::postGWrite(const infiniband::GroupWrite& data) {
2     /**
3      * gWRITE: post the following work requests:
4      * 1. RECV       on rq[0]
5      * 2. WAIT       on sq[1] waiting rq[0]
6      * 3. RDMA WRITE on sq[1] only if peer connected
7      * 4. SEND       on sq[1]
```

Figure 3.2: Worker QPs for primitive operations and how they work.




```
8       */
9
10    qps_[2]->postWaitRequest(*qps_[0], true, false);
11
12    if (isPeerNodeConnected()) {
13      auto target_mem_id = nvm_->getMemoryIdentifier(data.offset, data.size);
14      auto target_peer_mem_id =
15          rdma::createMemoryIdentifier(peer_nvm_id_, data.offset, data.size);
16      qps_[2]->postRDMAWriteRequest({target_mem_id}, target_peer_mem_id, false);
17    }
18
19    qps_[2]->postSendRequest({}, false);
20    qps_[0]->postReceiveRequest({});
21
22    return true;
23  }
24
25  bool Worker::postGMemcopy(const infiniband::GroupMemoryCopy& data) {
26    /**
27      * gMEMCPY: post the following work requests:
28      * 1. RECV       on rq[0]
29      * 2. WAIT       on sq[1] waiting rq[0]
30      * 3. RDMA WRITE on sq[1]
31      * 4. WAIT       on sq[2] waiting sq[1]
32      * 5. SEND       on sq[2]
33      */
34    qps_[1]->postWaitRequest(*qps_[0], true, false);
35    auto source_nvm_id = nvm_->getMemoryIdentifier(data.src_offset, data.size);
36    auto target_nvm_id = nvm_->getMemoryIdentifier(data.dst_offset, data.size);
37    qps_[1]->postRDMAWriteRequest({source_nvm_id}, target_nvm_id, true);
38    qps_[2]->postWaitRequest(*qps_[1], false, false);
39    qps_[2]->postSendRequest({}, false);
40    qps_[0]->postReceiveRequest({});
41
42    return true;
```

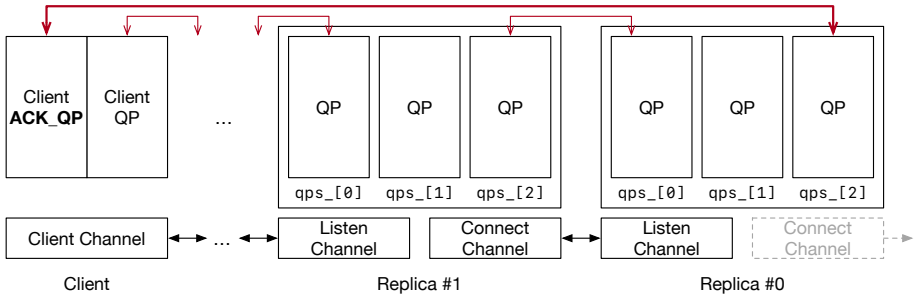
```
43 }
44
45 bool Worker::postGCompareAndSwap(const infiniband::GroupCompareAndSwap& data) {
46   /**
47    * gCAS: post the following work requests:
48    * 1. RECV       on rq[0]
49    * 2. WAIT       on sq[1] waiting rq[0]
50    * 3. CAS        on sq[1] (execute_map[index] == 1)
51    *    NOP        on sq[1] (execute_map[index] == 0)
52    * 4. WAIT       on sq[2] waiting sq[1]
53    * 5. SEND       on sq[2]
54    */
55   qps_[1]->postWaitRequest(*qps_[0], true, false);
56
57   if (data.execute_map[node_index_]) {
58     auto result_mem_id =
59         rdma::createMemoryIdentifier(*gcas_result_mr_, 0, sizeof(uint64_t));
60     auto target_mem_id =
61         nvm_->getMemoryIdentifier(data.offset, sizeof(uint64_t));
62     qps_[1]->postCASRequest(result_mem_id, target_mem_id, data.old_value,
63                             data.new_value, true);
64   } else {
65     qps_[1]->postNopRequest(true);
66   }
67
68   qps_[2]->postWaitRequest(*qps_[1], false, false);
69   qps_[2]->postSendRequest({}, false);
70   qps_[0]->postReceiveRequest({});
71
72   return true;
73 }
74
75 bool Worker::postGFlush(const infiniband::GroupFlush& data) {
76   /**
77    * gFLUSH: post the followng work requests:
78    * 1. RECV      on rq[0]
79    * 2. WAIT      on sq[1] waiting rq[0]
80    * 3. RDMA READ on sq[1]
81    * 4. WAIT      on sq[2] waiting sq[1]
82    * 5. SEND      on sq[2]
83    */
84   qps_[1]->postWaitRequest(*qps_[0], true, false);
85   auto nvm_id = nvm_->getMemoryIdentifier(data.offset, data.size);
86   qps_[1]->postRDMAReadRequest({nvm_id}, nvm_id, true);
87   qps_[2]->postWaitRequest(*qps_[1], false, false);
88   qps_[2]->postSendRequest({}, false);
89   qps_[0]->postReceiveRequest({});
90
91   return true;
92 }
```

Listing 3.10: Implementation of operations in Figure 3.2.

It looks weird that we temporaily stores the operations in the software queue. The reason the Worker should have a *software queue* will be explained in Section 3.1.4.

From the view of chaining nodes, the last QPs of the last node will be connected to *client's ACK QPs*, rather than leaved as unconnected state, as illustrated in Figure 3.3. By connecting the QPs to the client's ACK QPs, the replica can send *an ACK* to the client directly, following Figure 1.1. The last node recognizes it is the last one when its `peer_nvm_id` is invalid. `peer_nvm_id` is a

Figure 3.3: The last QPs of the last node will be connected to client, while TCP ConnectChannel in the last node remains `nullptr`.



`rdma::MemoryIdentifier` that represents an MR of the next node's NVM buffer, hence should not be valid its the node is the last one. For gWRITE, the last node will not send the RDMA WRITE WR, but just send an ACK message, because data does not need to be written to NVM of the client unnecessarily. Hence, it only posts a WAIT WR and a SEND WR into its `qps_[2]`.

3.1.3.4 `Hyperloop` class

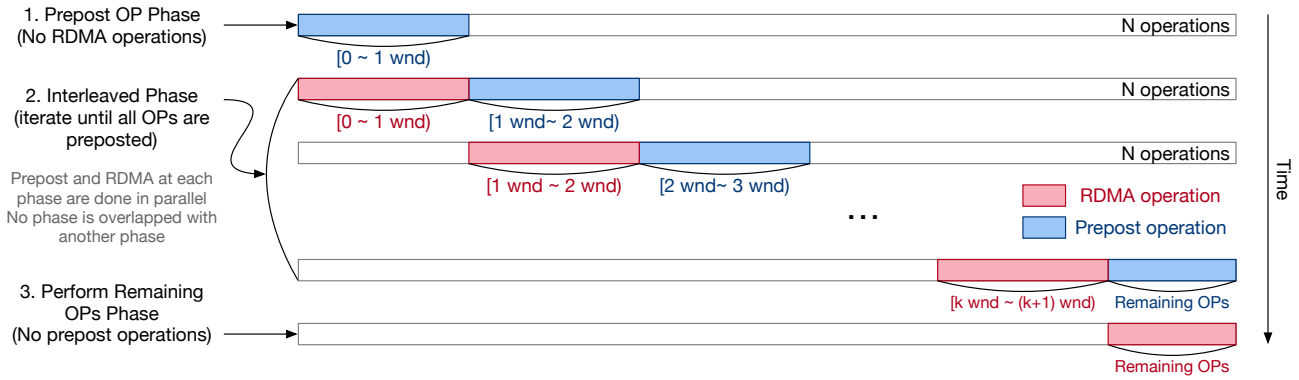
```
1  class Hyperloop {
2    /* Functions that clients use. */
3    Hyperloop(...);
4    void run();
5
6    /* RDMA related functions and variables */
7    void initializeWorker();
8    std::shared_ptr<rdma::Context> context_;
9    std::unique_ptr<Worker> worker_;
10
11   /* TCP related functions and variables */
12   std::unique_ptr<ListenChannel> listen_channel_;
13   std::unique_ptr<tcp::Channel> previous_node_channel_;
14   std::unique_ptr<tcp::Channel> next_node_channel_;
15
16   std::shared_ptr<NonVolatileMemory> nvm_;
17   rdma::MemoryIdentifier peer_nvm_id_;
18   int node_index_;
19 };
```

Listing 3.11: Hyperloop class. It is a main class providing functions of Hyperloop server replicas.

As a core class, it manages all the other subclasses; `rdma::Context`, `Worker`, TCP channels, and `NonVolatileMemory`. Its main responsibility is to communicate with a client and servers through the TCP channels, and to post works received from the TCP channel into the Worker. Again, the original Hyperloop does not necessarily have this step: *receiving works through the TCP channels*; but necessary for our implementation due to lack of *remote work remote manipulation*. For performance evaluation, we do not measure TCP overheads for work transfer; only overheads for RDMA operations are measured.

```
1  void Hyperloop::run() {
2    // TCP channels are established.
3    while (true) {
4      auto message = previous_node_channel_->receiveMessage<tcp::MessageType>();
5      if (message == tcp::MessageType::send_operation) {
6        auto op_num = previous_node_channel_->receiveMessage<unsigned int>();
```

Figure 3.4: An illustration that how window is used to perform operations.




```
7         auto ops = new rdma::Operation[op_num];
8         previous_node_channel_->receiveMessage(ops, op_num);
9
10        // Enqueue oeprations. The queue is a SW queue, not an RDMA queue.
11        worker_->enqueueOperations(ops, op_num);
12        delete[] ops;
13      } else if (message == tcp::MessageType::refill_operation) {
14        auto refill_op_num =
15            previous_node_channel_->receiveMessage<unsigned int>();
16        worker_->postOperations(refill_op_num);
17      }
18      ...
19    }
20  }
```

Listing 3.12: The implementation of Hyperloop core behavior.

It first establishes TCP connections with a client (initializing `previous_node_channel_`) and another server (initializing `next_node_channel_`), building a chained connection. Then, the client mainly sends two types of message for Hyperloop operations: operations transfer and a request for operation refill.

First, the client sends all Hyperloop operations to be performed in advance (for the purpose of pre-post) through TCP. `ops` (line 7) represents the received Hyperloop operations. The Hyperloop operations are pushed into the Hyperloop Worker's software queue. As explained in Section 3.1.3.3, corresponding RDMA WRs for the pushed Hyperloop operations are not posted to the HCA yet.

Second, the client sends a request to post RDMA WRs for the received Hyperloop operations. A reason that sending operations and posting operations are not unified is that *RDMA QP buffers are not big enough to post more than 100k operations.* Therefore, the client asks to post a portion of the pre-transmitted Hyperloop operations. The number of operations that can be posted at the same time is defined as *window*, also called *tx depth*. If, for example, the client wants to perform 2k Hyperloop operations, the number of which is larger than `max_window_size` (assumed to be set 1,000), it first should ask to post 1k operations, perform RDMA operations to consume the posted 1k operations, and then ask to post the other 1k operations. Otherwise, the RDMA QPs would return an error `ENOMEM` indicating that the QPs are full and no more RDMA WRs can be posted. Figure 3.4 illustrates how window is used for large number of Hyperloop operations.

```
1  unsigned long Hyperclient::doOperations(const std::vector<rdma::Operation>& ops,
2                                          unsigned int op_window_size) {
3    ...
4    if (op_window_size > rdma::max_window_size) {
```

```
5       // Print a warning message... Code omitted
6       op_window_size = rdma::max_window_size;
7     }
8     ...
9   }
```

Listing 3.13: The Hyperclient checks whether a request exceeds the `max_window_size`.

The window size check is done by the Hyperclient master as stated in Listing 3.13, when sending a `tcp::MessageType::refill_operation` message. It guarantees `refill_op_num` (line 14 in Listing 3.12) does not exceed `max_window_size`.

Once RDMA WRs are posted, the Hyperloop master does not wait any Work Completion, but just wait another TCP message from the client, to apply Hyperloop's fundamental principle: *eliminate server CPU operations from the critical path.*

3.1.3.5 Determining the Maximum Size of Window (TX Depth)

Currently, our implementation has the maximum window size 2,000. With Hyperloop operations, each of which requires at most 3 RDMA WRs, each QP should be capable to handle at most `max_window_size * 3`: 6,000. Although the device says it can support up to 32,768 outstanding WRs per QP (`max_qp_wr` from `ibv_devinfo -v`), it returns an `ENOMEM` error when we try to create a QP with the maximum WR configuration more than 6k. Therefore, we choose 2,000 as our maximum window size. However, it can be configured depending on hardware conditions.

3.1.4 `libhyperclient`

`libhyperclient` is a client library that can be used by any program. Its internal architecture is very similar to that of `libhyperloop`; it has `hyperclient` and `worker`.

3.1.4.1 `Hyperclient` class

Similar to `Hyperloop` core class (Section 3.1.3.4 and 3.1.5), `Hyperclient` is a core class of the client library. It internally manages a Channel and Workers, and provides API interface for Hyperloop operations to clients. Clients are required to use public functions defined in this class only.

```
1  class Hyperclient {
2    /* Operation related functions */
3    unsigned long doOperations(const std::vector<rdma::Operation>& ops,
4                               unsigned int op_window_size);
5
6    void sendOperations(const std::vector<rdma::Operation>& ops);
7    void postOperations(const unsigned int op_num);
8    void requestPostOperations(const unsigned int op_num);
9
10   /* Hyperclient Subclasses */
11   std::shared_ptr<rdma::Context> context_;
12   std::unique_ptr<tcp::Channel> channel_;
13   std::unique_ptr<Worker> worker_;
14
15   /* ThreadPool for efficient parallel execution */
16   ThreadPool thread_;
17 };
```

Listing 3.14: Hyperclient core class.

The Hyperclient master class is a core class of our Hyperloop implementation. An algorithm for hiding the overheads of preposting Hyperloop operations are implemented in `Hyperclient::sendOperations`. The Implementation consists of one public API and three private APIs:

- `Hyperclient::doOperations`: the only function that is exposed to clients. It internally uses the following three private functions for Hyperloop operations.
- `Hyperclient::sendOperations`: transfer `op_num` number of operations to server replicas through TCP (step 1). Hyperclient also posts operations into client Worker's software queue after assigning a Worker to each operation.
- `Hyperclient::requestPostOperations`: ask the server replicas to post `op_num` operations (step 2). The request message is sent through TCP as well. It is the step that preposting WRs in server replicas is done.
- `Hyperclient::postOperations`: post RDMA WRs for the given Hyperloop operations in the client (step 3). Actual RDMA communication is done in this step.

Step 2 and 3 are executed in parallel. By executing step 2 and 3 in parallel, our implementation can hide the overheads of refilling Hyperloop operations.

```
1  unsigned long Hyperclient::doOperations(const std::vector<rdma::Operation>& ops,
2                                          unsigned int op_window_size) {
3    sendOperations(ops);
4
5    unsigned int num_preposted_op = 0;
6    unsigned int num_performed_op = 0;
7
8    // 1. Prepost some operations.
9    if (ops.size() < op_window_size) {
10     requestPostOperations(ops.size());
11     num_preposted_op += ops.size();
12   } else {
13     requestPostOperations(op_window_size);
14     num_preposted_op += op_window_size;
15   }
16
17   // 2. Until all operations are preposted, iterate post and refill operations.
18   while (num_preposted_op < ops.size()) {
19     auto result = thread_.enqueue(
20       [&](const unsigned int prepost_op_num) -> unsigned long {
21         num_preposted_op += prepost_op_num;
22         return measureTime([&]() { requestPostOperations(prepost_op_num); });
23       },
24       std::min(static_cast<unsigned int>(ops.size()) - num_preposted_op,
25                op_window_size));
26
27     auto post_op_num =
28         std::min(static_cast<unsigned int>(ops.size()) - num_performed_op,
29                  op_window_size);
30
31     total_time += measureTime([&]() {
32       postOperations(post_op_num);
33       result.wait();
34     });
35     num_performed_op += post_op_num;
36   }
37
```

```
38    // 3. Preposting operations is done, iterate post operations.
39    while (num_performed_op < ops.size()) {
40      auto post_op_num =
41          std::min(static_cast<unsigned int>(ops.size()) - num_performed_op,
42                   op_window_size);
43
44      total_time += measureTime([&]() { postOperations(post_op_num); });
45      num_performed_op += post_op_num;
46    }
47  }
48
49  void Hyperclient::requestPostOperations(const unsigned int op_num) {
50    channel_->sendMessage(tcp::MessageType::refill_operation);
51    channel_->sendMessage(op_num);
52  }
53
54  void Hyperclient::postOperations(const unsigned int op_num) {
55    worker_->performOperations(op_num);
56    worker_->pollCompletion(op_num);
57  }
```

Listing 3.15: Source code for parallel execution. A C++11 lambda function is used.

After trasmitting the given Hyperloop operations through TCP by calling `sendOperations()` (line 3), the Hyperclient master performs Hyperloop operations in three steps by using two functions: `Hyperclient::requestPostOperations()` to transfer requests for preposting WRs through TCP and `Hyperclient::postOperations()` to perform RDMA operations. Detailed explanation for each steps is explained below.

1. The Hyperclient first requests to prepost some Hyperloop operations (line 9 $\sim$ 15) to prevent low RDMA utilization [3]. The number of preposted WRs is $window\_size$, or the number of the given Hyperloop operations if the number of the given operations are smaller than the default value. In this step, the Hyperclient does not perform any RDMA operations (no `postOperations` call).

2. After preposting some WRs, the Hyperclient proceeds to the second phase by executing and interleaving both operations together: the main thread begins perform RDMA operations by calling `postOperations()`, while another thread keeps sending requests for preposting WRs. The thread pool `thread_` is used for the parallel execution. Since the parallel execution is done frequently, an optimization to reduce the overheads for thread construction and destruction would be effective; once the thread pool is created, a thread in the thread pool can be recycled. `[&](const unsigned int prepost_op_num){...}` is a C++ lambda expression that represents an anonymous function with one argument `prepost_op_num` and returns an elapsed time as `unsigned long`. The function is enqueued into the thread pool's work queue and automatically executed by a worker thread. The worker thread executes `requestPostOperations()` function (line 22), which sends TCP messages to the server replicas to refill the next RDMA WRs, and the main thread executes `postOperations()` (line 32) that actaully performs RDMA operations. Then, the main thread waits until the worker thread completes its execution by waiting the result (line 33).

   Note that if RDMA operations are so fast that the overheads for preposting WRs are not hidden, the performance and network utilization will be degraded but is an intended behavior that we want to observe. Although Hyperloop does not clearly address this problem, refilling consumed WRs are required for continuous operation. If refilling WRs

is not fast enough, clients' requests will be delayed and blocked until servers post empty WRs for remote work request manipulation.

3. Preposting WRs should be finished *before* actual RDMA operations are all executed. After all WRs are preposted, the Hyperclient goes onto the third phase that only the main thread performs remaining RDMA operations (line 44). The worker thread is no longer used since preposting WRs are done.

3.1.4.2 `Worker` class

The `hyperclient::Worker` is similar to Hyperloop Worker class (Section 3.1.3.3), while its implementation is slightly different from that of Hyperloop. Differences can be summarized as follows:

1. WRs in the Hyperclient Worker will be executed immediately, while WRs in the Hyperloop Worker will be blocked by RECV and WAIT WRs until a RECV WR receives something from the client.

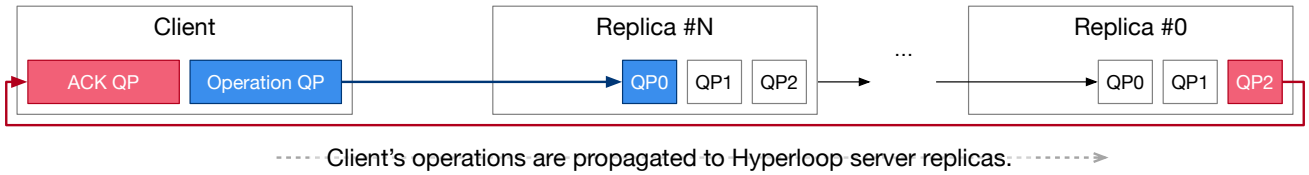
2. The Hyperclient Worker polls CQs until the execution finishes, while the Hyperloop Worker ignores all Work Completion events. The Hyperclient Worker has `pollCompletion()` function for this purpose.

```
1  class Worker {
2    bool doGWrite(const rdma::GroupWrite& op);
3    bool doGMemoryCopy(const rdma::GroupMemoryCopy& op);
4    bool doGCompareAndSwap(const rdma::GroupCompareAndSwap& op);
5    bool doGFlush(const rdma::GroupFlush& op);
6    void pollCompletion(const unsigned int op_num);
7
8    /* QP related functions and variables */
9    unsigned int performOperations(unsigned int op_num);
10   std::shared_ptr<rdma::Context> context_;
11   rdma::QueuePair* operation_qp_;
12   rdma::QueuePair* ack_qp_;
13
14   /* Software Queue that temporarily stores operations */
15   void enqueueOperations(const rdma::Operation ops[],
16                          const unsigned int op_num);
17   moodycamel::BlockingReaderWriterQueue<rdma::Operation> queue_;
18   unsigned int num_queue_items_;
19 };
20
21 /**
22  * gWRITE: post the following work requests:
23  * 1. RDMA on qp
24  * 2. SEND on qp
25  *
26  * 3. RECV on ack_qp (for ACK)
27  *
28  * In hyperclient, poll until 3 completes.
29  * For timing issue, 3. RECV is inserted before 1 and 2.
30  */
31 bool Worker::doGWrite(const rdma::GroupWrite& op) {
32   auto buffer_mr = context_->getMemoryRegion(op.client_buffer_base);
33   auto target_mem_id =
34       rdma::createMemoryIdentifier(*buffer_mr, op.offset, op.size);
35   auto target_peer_mem_id =
```

```
36        rdma::createMemoryIdentifier(server_nvm_id_, op.offset, op.size);
37
38    ack_qp_->postReceiveRequest({});
39    operation_qp_->postRDMAWriteRequest({target_mem_id}, target_peer_mem_id,
40                                        false);
41    operation_qp_->postSendRequest({}, false);
42    return true;
43  }
44  /**
45   * gMEMCPY: post the following work requests:
46   * 1. SEND on qp
47   *
48   * 2. RECV on ack_qp (for ACK)
49   *
50   * In hyperclient, poll until 2 completes.
51   * For timing issue, 2. RECV is inserted before 1.
52   */
53  bool Worker::doGMemoryCopy(const rdma::GroupMemoryCopy& op) {
54    ack_qp_->postReceiveRequest({});
55    operation_qp_->postSendRequest({}, false);
56    return true;
57  }
58
59  /**
60   * gCAS: post the following work requests:
61   * 1. SEND on qp
62   *
63   * 2. RECV on qp (for ACK)
64   *
65   * In hyperclient, poll until 2 completes.
66   * For timing issue, 2. RECV is inserted before 1.
67   */
68  bool Worker::doGCompareAndSwap(const rdma::GroupCompareAndSwap& op) {
69    ack_qp_->postReceiveRequest({});
70    operation_qp_->postSendRequest({}, false);
71    return true;
72  }
73
74  /**
75   * gFLUSH: post the following work requests:
76   * 1. SEND on qp
77   *
78   * 2. RECV on qp (for ACK)
79   *
80   * In hyperclient, poll until 2 completes.
81   * For timing issue, 2. RECV is inserted before 1.
82   */
83  bool Worker::doGFlush(const rdma::GroupFlush& op) {
84    ack_qp_->postReceiveRequest({});
85    operation_qp_->postSendRequest({}, false);
86    return true;
87  }
88
89  void Worker::pollCompletion(const unsigned int op_num) {
90    for (int i = 0; i < op_num; i++) {
91      ack_qp_->pollCompletion(true, 1000);
92    }
93  }
```

Listing 3.16: Implementation of Hyperclient operations.

Figure 3.5: How QPs of the Hyperclient Worker are connected to Hyperloop servers replicas.



Operations in the Hyperclient Worker is simpler than those in the Hyperloop Worker. It roles as an initiator of Hyperloop operations. The actual code posts a RECV WR first, different from the comments. It ensures to avoid a timing issue; an RDMA operation can be done before a RECV WR is posted and would invoke an error. By posting a RECV WR before initiating an operation, this kind of errors can be avoided. Figure 3.2 also illustrates the Hyperclient Worker operations as well.

The Hyperclient Worker has two QPs: `operation_qp_` and `ack_qp_`. Figure 3.5 illustrates how QPs are connected. The operation QP is connected to the first QPs of the last replica node (replica #N) that the client is directly connected to. We introduced the *ACK QP* in Section 3.1.3.3 (Refer to Figure 3.3). It is connected to the last QP of the first replica node (replica #0).

The Hyperclient Worker posts a RECV WR to wait an ACK with the ACK QP to ensure that all operations have been operated in all server replicas. If the first replica (replica #0) sends an ACK to the client, this means all replicated transactions are done successfully.

All operations pushed by the Hyperclient master into Workers are temporarily stored in software queue `queue_`. This is because WRs in the client would immediately be executed, hence should be posted at proper time. `Worker::performOperations()` dequeues `op_num` number of operations from the software queue and posts the corresponding WRs into the operation QP.

3.1.5 Running Hyperloop

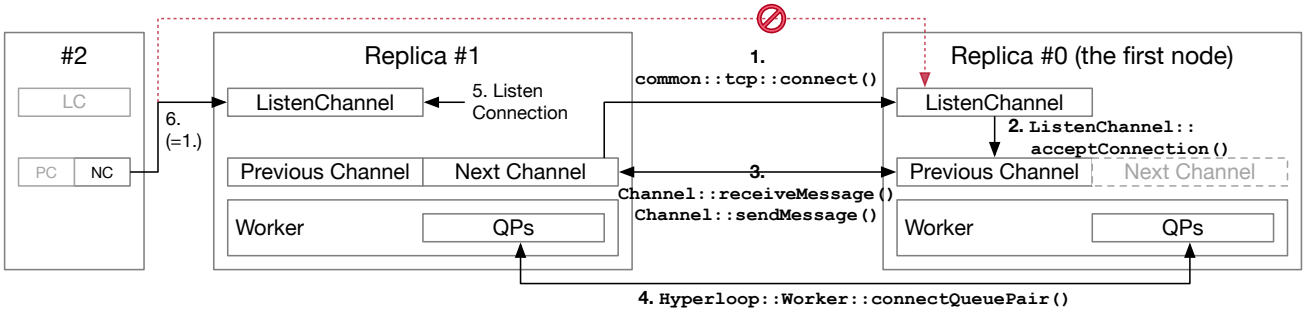
In this section, we integrate how the Hyperloop and the Hyperclient interact and work together.

3.1.5.1 Establishing a Chained Connection between Hyperloop Servers

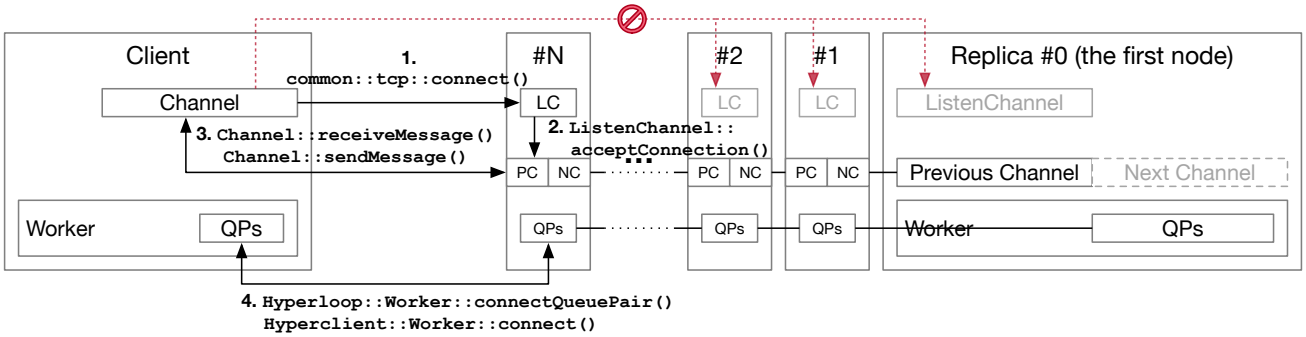
At first, the server replicas should establish a chained connection before providing Hyperloop service to a client.

```
1  Hyperloop::Hyperloop(const std::string& device_name,
2                       const int device_port,
3                       const uintptr_t nvm_buffer,
4                       const size_t nvm_size,
5                       const int tcp_listen_port,
6                       const std::string& peer_ip_port)
7      : context_(new rdma::Context(device_name, device_port)),
8        nvm_(new NonVolatileMemory(
9        context_->createMemoryRegion(nvm_buffer, nvm_size),
10       nvm_buffer,
11       nvm_size)),
12       listen_channel_(new ListenChannel(tcp_listen_port)),
13       previous_node_channel_(nullptr),
14       next_node_channel_(nullptr),
15       node_index_(0),
16       worker_(nullptr) {
17   if (peer_ip_port.empty() == false) {
```

Figure 3.6: Connection chain establishment mechanism. The Hyperloop servers and clients only communicate with an adjacent one.



(a) Initial channel establishment between two server replicas.
The next replica #2 connects to #1 (not to #0), implementing a connection chain.



(b) After server replicas establish a connection chain, a client connects to the last node (#N).

```
18      next_node_channel_.reset(new tcp::Channel(tcp::connect(peer_ip_port)));
19    }
20  }
21
22  void Hyperloop::run() {
23    while (true) {
24      previous_node_channel_.reset(listen_channel_->acceptConnection());
25      initializeWorker();
26      ...
27    }
28  }
```

Listing 3.17: Source code for initializing Hyperloop servers.

The first Hyperloop server has no another server process to connect to, hence it runs as a sole server instance, binding a listening socket to the given port (the listening port is bound in the constructor of `ListenChannel` class). Then, the next server can connect to the server, establishing a connection between server instances. The first server has an empty `peer_ip_port`, while the others has a valid IP address for the previous server. Following the code (line 17 $\sim$ 19), the first server has no `next_node_channel` instance (leaved as `nullptr`), while the others has a valid class instance.

Once the server has a connected connection (line 24), the process does not accept a further connection request until the current connection is destroyed. The newly connected server process instead accepts a new connection request by calling `listen_channel_->acceptConnection()`. By continuing the operation, server replicas has a chained connection. Figure 3.6 illustrates this step.

3.1.5.2 Establishing a Connection between the last Hyperloop Server and a Client

```
Hyperclient::Hyperclient(const std::string& device_name,
                         const int device_port,
                         const std::string& server_ip_port,
                         const uintptr_t buffer_address,
                         const size_t buffer_size)
      : context_(new rdma::Context(device_name, device_port)),
        channel_(new tcp::Channel(tcp::connect(server_ip_port))),
        thread_(1) {
  ...
}
```

Listing 3.18: Source code for connecting to the Hyperloop server from a client.

All the Hyperloop servers listen a connection request at least once (line 24). If another server process connects to the listening server instance, the existing server becomes an intermediate one and the new server becomes a new listening server. If a client connects to the listening server, connection chaining is done the client begins Hyperloop operations. Both Hyperloop servers and clients use `common::tcp::connect()` API to connect to the listening server.

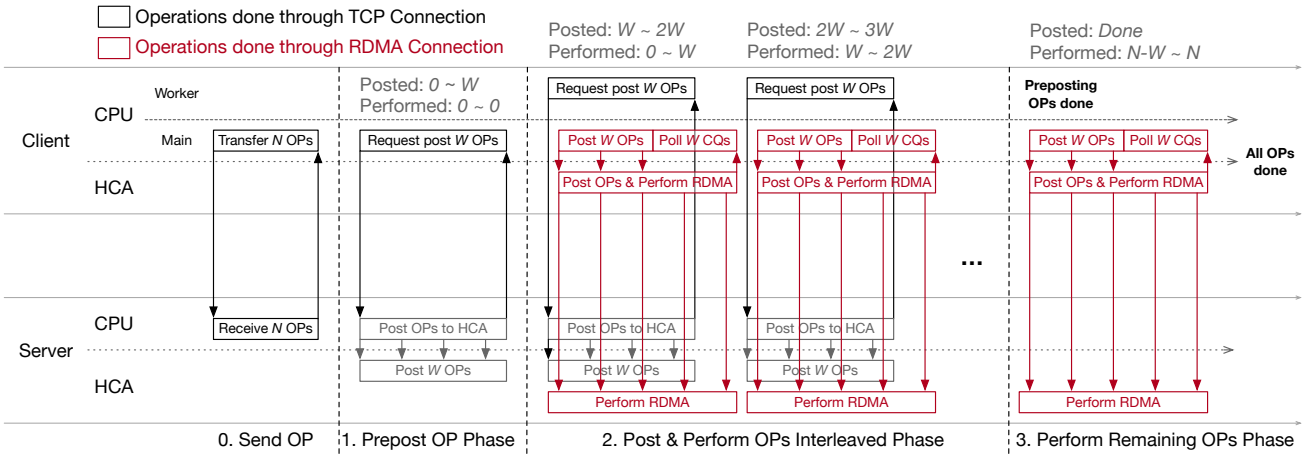
3.1.5.3 Initializing RDMA QP Connection

Once TCP connections are established, a client and server replicas should connect their QPs as well.

```
void Hyperloop::initializeWorker() {
  worker_.reset(new Worker(context_, nvm_, node_index_, peer_nvm_id_));

  auto qp0_id = worker_->getQueuePairIdentifier(0);
  auto qp2_id = worker_->getQueuePairIdentifier(2);

  // Exchange queue pair information first.
  auto identifier =
      previous_node_channel_->receiveMessage<rdma::QueueIdentifier>();
  worker_->connectQueuePair(0, identifier);
  previous_node_channel_->sendMessage(qp0_id);

  worker_->connectQueuePair(1, worker_->getQueuePairIdentifier(1);

  if (next_node_channel_) {
    auto qp2_id = worker_->getQueuePairIdentifier(2);
    next_node_channel_->sendMessage(qp2_id);
    auto identifier =
        next_node_channel_->receiveMessage<rdma::QueueIdentifier>();
    worker_->connectQueuePair(2, identifier);
  }

  // Pass the client's ack QP information to the last node.
  identifier = previous_node_channel_->receiveMessage<rdma::QueueIdentifier>();
  if (next_node_channel_) {
    next_node_channel_->sendMessage(identifier);
    identifier = next_node_channel_->receiveMessage<rdma::QueueIdentifier>();
    previous_node_channel_->sendMessage(identifier);
  } else {
    worker_->connectQueuePair(2, identifier);
    previous_node_channel_->sendMessage(qp2_id);
  }
}
```

Listing 3.19: Source code for QP connection in Hyperloop servers.

Figure 3.7: How the Hyperloop server and clinets interact to perform operations.




```
1  Hyperclient::Hyperclient(const std::string& device_name,
2                           const int device_port,
3                           const std::string& server_ip_port,
4                           const uintptr_t buffer_address,
5                           const size_t buffer_size)
6        : context_(new rdma::Context(device_name, device_port)),
7        channel_(new tcp::Channel(tcp::connect(server_ip_port))),
8        thread_(1) {
9    auto operation_qp = context_->createQueuePair();
10   auto ack_qp = context_->createQueuePair();
11
12   channel_->sendMessage(operation_qp->getQueueIdentifier());
13   auto identifier = channel_->receiveMessage<rdma::QueueIdentifier>();
14   operation_qp->connect(identifier);
15
16   channel_->sendMessage(ack_qp->getQueueIdentifier());
17   identifier = channel_->receiveMessage<rdma::QueueIdentifier>();
18   ack_qp->connect(identifier);
19
20   worker_.reset(new Worker(context_, server_nvm_id_));
21 }
```

Listing 3.20: Source code for QP connection in a Hyperloop client.

The Hyperloop servers and a client exchange data to connect QPs right after establishing TCP connections. The QP information is represented as `common::rdma::QueueIdentifier`, as illustrated in Section 3.1.1.1.

The client has its dedicated *ACK QP* as mentioned in Figure 3.5, which is connected to the last QP of the first Hyperloop server. The connection is established by the code line 23 ∼ 32 in Listing 3.19. After establishing peer QP connection, the client passes its ACK QP information to the server. Intermediate servers (all server replicas except for the first one) just pass the information to the next node, and then the first server receives the information and establishes the connection. Likewise, the first server also send its QP information to the client through the same path in a reverse way. The client receives the QP information and establishes the connection (line 12 ∼ 18 in Listing 3.20).

3.1.5.4 Handling Hyperloop Operations

With the TCP connection and the RDMA connection, the client initiates Hyperloop operations, as explained in Section 3.1.3 and 3.1.4. This section reviews their behaviors and summarizes their interaction. Figure 3.7 illustrates how the Hyperloop servers and the client communicate with each other. In each step, the interaction between the client and the servers is as follows.

1. The client sends Hyperloop operations to be performed in advance to the server replicas to simulate Hyperloop remote work request manipulation. All the server replicas receives the operations and pushes the received operations into the Hyperloop Workers' software queue.

2. Those operations are not posted into the Hyperloop servers' RDMA QPs until the client send a request for posting operations. The client sends a request to prepost some WRs through the TCP channel. The servers receive the request and post the WRs dequeued from their software queue to their RDMA QPs. Yet, the client does not perform RDMA operations.

3. After preposting some Hyperloop operations, the client initiates RDMA operations, and an additional Hyperclient Worker thread keeps sending requests to prepost WRs through the TCP channel.

   The client polls Work Completion events after performing a bunch of WRs from the ACK QP to ensure that all operations are executed normally in all server replicas. If Polling Work Completion is done, it means posted WRs are all consumed in all server replicas, and the next operations can be posted into the servers' QPs.

   While the client uses Work Completion events to check the progress, the servers do nothing regarding Work Completion. The servers leave all RDMA related jobs to their HCAs other than posting WRs.

3.2 Using Hyperloop Implementation

3.2.1 Prerequisites

This section describe required Linux packages to use Hyperloop. The guideline is based on Debian based Linux, such as Ubuntu.

Hyperloop is implementd with C++14, which requires g++ verion 7 or newer to be built. Ubuntu uses g++7 by default since Ubuntu 18.04, however, you need to manually install g++7 if you are using older versions of Ubuntu or any other Linux distributions that do not have g++7 by default.

```
$ sudo apt-add-repository ppa:ubuntu-toolchain-r/test
$ sudo apt update && sudo apt install g++-7 --yes
$ export CXX=/usr/bin/g++7
```

It uses Cmake as its build system. Install it as well.

```
$ sudo apt install cmake --yes
```

Or, manually install the package from `https://cmake.org/download/`.

3.2.2 Compiling the Source Code

Our Hyperloop implementation consists of two libraries, and two executable binaries that use the libraries for the purpose of microbenchmark test.

Cmake supports out-of-source build. Assume you put the source code at `/hyperloop`.

```
$ mkdir /build; cd /build
$ cmake ../hyperloop
$ make -j
```

The compiled libraries and executable binaries will be placed at:

- libhyperloop: `/build/libhyperloop/libhyperloop.so`
- libhyperclient: `/build/libhyperclient/libhyperclient.so`
- client: `/build/apps/client/client`
- server: `/build/apps/server/server`

3.2.3 Using the Applications

The Hyperloop implementation provides default applications that can be used to test Hyperloop working mechanisms: `/build/apps/client/client` and `/build/apps/server/server`.

Assume that each server node *nodeN* has an IPv4 address *10.0.0.N*. The following example establishes 4 connected nodes each of which runs a server process, and runs a client connecting to the chained system. For more information about arguments, use `-help`.

```
node1 $ /build/apps/server/server --port 50000
node2 $ /build/apps/server/server --connect 10.0.0.1:50000 --port 10000
node3 $ /build/apps/server/server --connect 10.0.0.2:10000 --port 45000
node4 $ /build/apps/server/server --connect 10.0.0.3:45000 --port 52000
node5 $ /build/apps/client/client -c 10.0.0.4:52000 -s 65536 -n 100000 -w 1000
```

The first server in node1 does not send a connection request but waits a connection request. node2 establishes a connection to node1 for the chained system, and waits a connection request as well. After the connection is established, node1 does not accept further connection requests until its client node2 is disconnected. Same operations are done in node 3 and 4. While node 4 is able to accept a connection request from either an additional server instance or a client, node5 establishes a connection with node4 *as a client*. The client in node5 will send operation requests to the chained system having 4 nodes: node1, 2, 3, and 4.

Several configurations are provided by both the server:

- `device`: set device name. Default is `mlx5_0`.
- `port`: set TCP listen port. The next server replica or a client should connect to this port.

Current client implementation sends a bunch of gWRITE requests to the server. Following configurations are provided by the client to adjust settings regarding gWRITE operations:

- `size`: set a size of gWRITE requests in bytes. With 65536, for instance, each gWRITE operation writes 64KB data to every server replicas. Default is 1,024 (1KB).
- `number`: set the number of gWRITE requests to be performed. 100000 means it performs 100,000 gWRITE operations. Default is 5,000.
- `window`: set how many requests can be posted at once. With the window size 1000, the client will prepost 1000 operations at first, and keep post 1000 operations after getting 1000 Work Completion events. Default is 1 and the maximum size is 2,000.
- `use-notify`: use solicited event notification instead of polling to wait Work Completion events to reduce CPU usage.

3.2.4 Linking the Library

`/build/apps/server/server` can be used as a Hyperloop server node, however, you many want to implement your own Hyperloop client program. In this case, you build your own and link it with *libhyperclient*. Due to relative include paths that hyperclient internally uses, you need to include the header as follows:

```
#include "libhyperclient/hyperclient.h"

g++ -o <your_program> -lhyperclient -L/build/libhyperclient -I/build <your_code>
```

The implementation provides a compatibility layer for C programs. Use `hyperclient_compat.h` instead of `hyperclient.h`.

```
#include "libhyperclient/hyperclient_compat.h"

gcc -o <your_program> -lhyperclient -L/build/libhyperclient -I/build <your_code>
```

3.2.5 API Functions

The Hyperclient core class provides the following API functions:

```
1  Hyperclient(const std::string& device_name,
2              const int device_port,
3              const std::string& server_ip_port,
4              const uintptr_t buffer_address,
5              const size_t buffer_size);
6
7  bool Hyperclient::sendPrintBuffer(const uint64_t offset, const size_t length);
8  unsigned long Hyperclient::doOperations(
9              const std::vector<rdma::Operation>& ops,
10             unsigned int op_window_size);
```

Listing 3.21: API functions in the Hyperclient core class. Clients must access Hyperloop through the Hyperclient core class.

The compatibility layer provides the following API functions:

```
1  struct compat_hyperclient* compat_hyperclient_connect(
2              const char* device_name,
3              const int device_port,
4              const char* server_ip_port,
5              const void* buffer_address,
6              const size_t buffer_size);
7
8  int compat_hyperclient_destory(struct compat_hyperclient* client);
9
10 int compat_hyperclient_send_print_buffer(
11             struct compat_hyperclient* client,
12             const uint64_t offset,
13             const size_t length);
14
15 unsigned long compat_hyperclient_do_operations(
16             struct compat_hyperclient* client,
17             const struct compat_hyperloop_operation ops[],
18             const unsigned int ops_num,
19             const unsigned int op_window_size);
```

Listing 3.22: API functions in the Hyperclient compatibility layer. Note that all API functions and data structures for C compatibility have a prefix `compat_`.

The APIs for C and C++ are exactly equivalent with a small amount of adjustments for each language. For instance, `std::vector<rdma::Operation>` C++ STL type is replaced with a combination of an array of `struct compat_hyperloop_operation` and the number of the operations for `hyperclient_do_operations`, or `std::string`s are replaced with a traditional string representation: `const char*`.

`hyperclient_do_operations` asks API users to pass their operations through `std::vector<rdma::Operation>&` or `struct compat_hyperloop_operation*`. The following example code illustrates how to build the operations.

```
1  constexpr size_t buffer_size = 0x20;
2  auto buffer = new char[buffer_size];
3  rdma::Operation op_gwrite, op_gmemcpy, op_gcas, op_gflush;
4  op_gwrite.type = rdma::OperationType::GroupWrite;
5  /**
6   * rdma::Operation.op.gWrite has three variables:
7   * 1. (uint64_t) buffer_address: indicates the buffer address
8   *                               in this node to be copied.
9   * 2. (uint64_t) offset: indicates the offset from the buffer address.
10  * 3. (uint64_t) size: indicates the size of data to be copied.
11  *
12  * [buffer_address+offset, buffer buffer_address+offset+size) in this node
13  * will be copied to
14  * [NVM_base_address+offset, NVM_base_address+offset+size) in each replica.
15  *
16  * In this example, buffer[0x10~0x3f] will be copied to
17  * A[0x10~0x3f] (A is the NVM buffer address for each server replica).
18  */
19 op_gwrite.op.gWrite = {reinterpret_cast<uint64_t>(buffer), 0x10, buffer_size};
20
21 op_gmemcpy.type = rdma::OperationType::GroupMemoryCopy;
22 /**
23  * rdma::Operation.op.gMemcpy has three variables:
24  * 1. (uint64_t) source offset: indicates the offset
25  *               from which data should be copied.
26  * 2. (uint64_t) destination offset: indicates the offset
27  *               to which data should \\be copied.
28  * 3. (uint64_t) size: indicates the size of data to be copied.
29  *
30  * In this example, [A+0x0~0x2f] will be copied to [A+0x50~0x7f]
31  * (A is the NVM buffer address for each server replica).
32  */
33 op_gmemcpy.op.gMemcpy = {0x0, 0x50, 0x30};
34
35 op_gcas.type = rdma::OperationType::GroupCompareAndSwap;
36 /**
37  * rdma::Operation.op.gCas has four variables:
38  * 1. (uint64_t) offset: indicates the offset in whch data should be compared.
39  * 2. (uint64_t) old value: checks whether the data in [offset~offset+0x8]
40  *                          equals to this value.
41  * 3. (uint64_t) new value: if the 8-byte data matches the old value,
42  *                          replace it with the new value.
43  * 4. (bool[64]) execute map: indicates whether each node should execute gCAS.
44  *
45  * In this example, A[0x0~0x8] will be compared with
```

```
 * 0x6f57206f6c6c6548ULL (== "Hello Wo").
 * If data matches, this will be replaced to
 * 0x69686968 (== "hihi") in node 0 and 2.
 * (A is the NVM buffer address for each server replica)
 */
op_gcas.op.gCas = {0x0,
                   0x6f57206f6c6c6548ULL,
                   0x69686968,
                   {
                       1,
                       0,
                       1,
                       0,
                   }};

op_gflush.type = rdma::OperationType::GroupFlush;
/**
 * We implemented gFlush to read the buffer through RDMA,
 * so that any data in CPU caches should be flushed to the NVM.
 * rdma::Operation.op.gFlush has two variables:
 * 1. (uint64_t) offset: indicates the offset of the buffer.
 * 2. (uint64_t) size: indicates the size of the buffer.
 * To flush the whole buffer, we flush the entire buffer size that
 * we modified (0x0~0x100).
 */
op_gflush.op.gFlush = {0x0, buffer_size};

/* Pass all four operations at once with STL vector */
std::vector<rdma::Operation> ops{ops_gwrite, ops_gmemcpy, ops_gcas, ops_gflush};
hyperclient::Hyperclient client(...);
client.doOperations(ops);
```

Listing 3.23: An example of implementing Hyperloop operations for a library function call. Source code from `apps/client/main.cpp`.

Similar interface is provided by the C compatibility layer as well. The following code example works exactly same with Listing 3.23:

```
const size_t buffer_size = 0x20;
char* buffer = (char*) malloc(bufer_size);

// Requires ANSI C99 standard for the initialization style
struct compat_hyperloop_operation op_gwrite = {
  .type = OPERATION_TYPE_GWRITE,
  .op = {.gWrite = {0x0, buffer_size}}
};

struct compat_hyperloop_operation op_gmemcpy = {
  .type = OPERATION_TYPE_GMEMCPY,
  .op = {.gMemcpy = {0x0, 0x50, 0x30}}
};

struct compat_hyperloop_operation op_gcas = {
  .type = OPERATION_TYPE_GCAS,
  .op = {.gCas = {0x0,
                  0x6f57206f6c6c6548ULL,
                  0x69686968,
                  {
                      1,
                      0,
```

```
                        1,
                        0,
                    }
                }
        }
};

struct compat_hyperloop_operation op_gflush = {
  .type = OPERATION_TYPE_GFLUSH,
  .op = {.gFlush = {0x0, buffer_size}}
};

struct compat_hyperloop_operation_op ops[4] =
    {op_gwrite, op_gmemcpy, op_gcas, op_gflush};
struct compat_hyperclient* client = compat_hyperclient_connect(...);
compat_hyperclient_do_operations(client, ops, 4, 1);
// Should destroy the client after using it;
// otherwise, another client cannot establish a connection.
compat_hyperclient_destroy(client);
```

Listing 3.24: An example of implementing Hyperloop operations for a library function call for C. Source code from `apps/client/main_c.c`.

Note that `compat_hyperloop_operation` looks exactly same with `rdma::Operation`, hence explanations in Listing 3.23 can also be applied into the compatibility layer APIs. Refer to client programs (`apps/client/main.cpp` for C++ and `apps/client/main_c.c` for C) for more details.

Chapter 4
Limitations

4.1 Feasibility of Implementing Remote Work Request Manipulation

Our implementation has different behaviors in many parts due to lack of *remote work request manipulation*. This section describes why the functionality cannot be implemented, and what would be possible conditions to achieve it.

Remote work request manipulation is a core feature of Hyperloop [21], manipulating metadata of preposted WRs in the server replicas, so that the CPUs are not necessarily involved in the critical path of replicated transactions.

We first review how work requests are posted into the HCA in a node (Refer to Section 2.1.2.2 and 2.1.2.3 for more details):

1. Write WQE to the WQE buffer sequentially to previously-posted WQE.
2. Update Doorbell Record associated with the queue.
3. For send request ring Doorbell by writing to the Doorbell register.

Remote work request manipulation changes the process of posting WRs as follows:

1. Write a bunch of *invalid* WQEs to the *SQ* buffer sequentially to previously-posted WQE. The HCA cannot consume the WQs due to the ownership, or even cannot know whether the WQEs are posted in the WQE buffer, since a doorbell is not rung yet.
2. Update Doorbell Record associated with the queue. Now the ownership is passed to the HCA, however, the HCA still does not recognize whether there exists WQs to be handled, since its doorbell is not rung yet.
3. Write a RECV WR to the RQ buffer; its scatter list contains the SQ buffer in step 1.
4. A peer sends a SEND WR containing metadata for valid WQEs; then the posted RECV WR *overwrites* the WQEs. **Also, the SEND WR sends a 8-byte doorbell register value, which will be written to the node's doorbell register ringing a doorbell, and it notifies the HCA to begin consuming *manipulated WQEs*.**

We could not implement step 4; writing a doorbell register value to the doorbell register via RDMA (so called *remote doorbell ring*) was not possible. We experienced two problems regarding the issue, one solved but the other not solved.

4.1.1 Registering Doorbell Register as a Memory Region (MR)

A HCA's doorbell register is in its PCIe address space; hence it can be accessed via physical memory address, but is not physical memory. Even user processes can access the doorbell register through a virtual memory address mapped to it, the virtual address cannot be used to register a MR as already discussed in Section 2.2.3.1. Registering a MR only takes a memory address that points *physical memory*, and the virtual address pointing to the doorbell register is not in the case, returning `EINVAL`. Using PA-MR returns a MR for the doorbell register, however, reading data from the MR or writing data to the MR did not work.

The reason `ibv_reg_mr()` returns `EINVAL` for doorbell register address is that the corresponding kernel module function `mlx5_ib_reg_user_mr()` uses the kernel API `get_user_pages()`.

```
get_user_pages         (ofed/drivers/infiniband/core/umem.c)
- ib_umem_get          (ofed/drivers/infiniband/core/umem.c)
- mr_umem_get          (ofed/drivers/infiniband/hw/mlx5/mr.c)
- mlx5_ib_reg_user_mr (ofed/drivers/infiniband/hw/mlx5/mr.c)
```

Listing 4.1: Linux kernel function call stack when user calls `mlx5_ib_reg_mr()`.

`get_user_pages()` is used to pin pageable user-space memory buffer to prevent it from eviction. Memory-mapped I/O such as accessing the doorbell register does not use memory to access the corresponding PCIe device register, hence using `get_user_pages` for MMIO address is not a proper use case [13, 31]. Mellanox has been aware of this problem, and implements a new technology called *PeerDirect* (It is also an ancestor of *NVIDIA GPUDirect*) [31, 22, 17, 27].

PeerDirect enables the HCA to directly access to other PCIe devices via their PCIe address space. Considering the fact that the HCA can directly access to other devices' PCIe address space, **we wonder whether it is possible for the HCA to even access to its own PCIe address space using PeerDirect**, though by definition it is not *peer-to-peer* communication. We used p2pmem, an implementation of PeerDirect that reveals `/dev/p2pmemX` device that is mapped to the specified device (in our case the HCA itself) [11, 7]. A virtual address mapped for the device `/dev/p2pmemX` was successfully registered and `ibv_reg_mr()` returned the corresponding MR.

4.1.2 Adopting P2P Communication to Remote Doorbell Ring

Although we have the MR with p2pmem, remote doorbell ring still failed like in PA-MR case. We found other ways for PCIe P2P, namely *P2PDMA* and *iopmem* introduced later than PeerDirect, and tried it as well, but had no luck [28, 9, 10, 6, 8].

While analyzing PCIe Peer-to-Peer communication, we found a commit and an archive of related e-mails, introducing a restriction regarding peer-to-peer transfer [18]:

> *> I'm pretty sure the spec disallows routing-to-self so doing a P2P*
> *> transaction in that sense isn't going to work unless the device*
> *> specifically supports it and intercepts the traffic before it gets to*
> *> the port.*
>
> *This is correct. Unless the device intercepts the TLP before it hits the root-port then this would be considered a "route to self" violation and an error event would occur. The same holds for the downstream port on a PCI switch (unless route-to-self violations are disabled which violates the spec but which I have seen done in certain applications).*
>
> *Stephen*

Figure 4.1: A sample PCIe architecture that a HCA and a SSD are connected to.

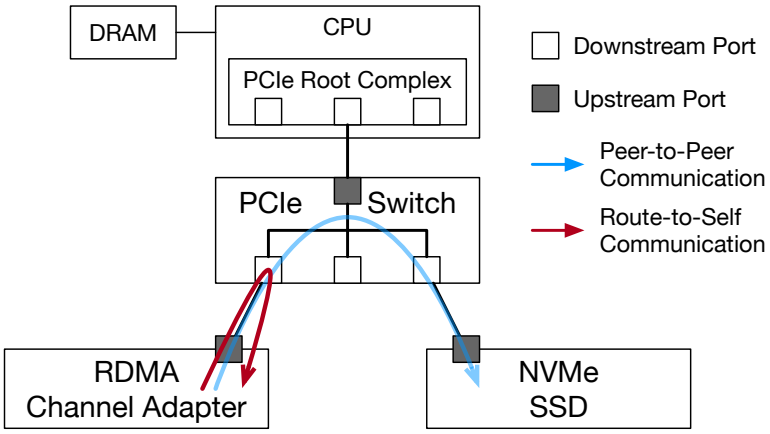


Figure 4.2: A possible solution for remote doorbell ring: using multiple HCAs.

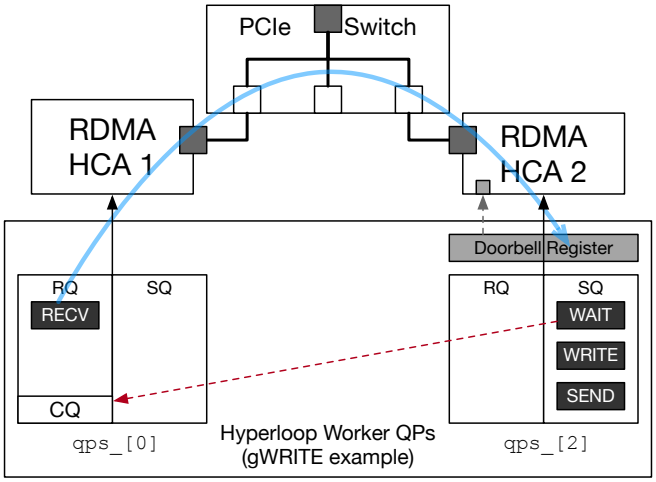


Figure 4.1 illustrates the difference between peer-to-peer communication (blue arrow) and route-to-self communication (red arrow). The PCIe switch (including PCIe Root Complex) determines which port a received TLP should be routed using register values stored in ports' PCI configuration space [25, 29]. *Route-to-self transaction is disallowed* means that all PCIe switch (including PCIe Root Complex) does not route TLPs to the downstream port which they came from. Therefore, if the HCA would consume a generated TLP instead of sending it to its upstream port, it would be possible to implement *remote doorbell ring*, and finally *remote work request manipulation*. With several experiments we have done, however, the HCA seems not support route-to-self, hence implementing *remote doorbell ring* is not possible.

4.1.3 Possible Solution: Using Multiple HCAs

It is **Route-to-self** not allowed in the current PCIe system. Hence, it might be possible to make it work *if we use multiple HCAs for each node*, as illustrated in Figure 4.2. This would be followed by another problem that whether it is possible for a WAIT WR in the SQ of `gwrite_ops_[1]` to wait a CQ gene ration in the CQ of `gwrite_ops_[0]` (red arrow). While CQs are not protected by Protection Domain, we may have no way to make the WAIT WR work since they are associated to different device and contexts.

Verfiying whether it actually does work is a future work.

4.2 Using Vendor-Specific Verb APIs

Hyperloop depends on WAIT WRs, which are only provided by Mellanox OFED driver. RDMA APIs merged into Linux kernel mainline does not have this feature. It means Hyperloop only works on Mellanox devices, which reduces its availability.

Furthermore, Mellanox introduced RDMA-Core device driver (OFED v5.0) at March 2020, which contains core parts of RDMA APIs and **their own private verb APIs are excluded** [36]. All experimental verb APIs, including Cross Channel and WAIT WR, will no longer be supported and provided in the new RDMA-Core device driver, which makes impossible for Hyperloop to run on the newest device driver.

Chapter 5
Evaluation

We evaluate our Hyperloop implementation using microbenchmark and compare it with the results in Section 6.1 of Hyperloop [21].

5.1 Testbed Setup

Our testbed system consists of 3 nodes to simulate Hyperloop with group size 3. The specification of the nodes is summarized in Table 5.1. We are supposed to use a NVM device as a storage such as Intel Optane DIMM, however, we use DRAM as a prototype implementation.

Our implementation does not exactly match the original Hyperloop; it lacks *remote work request manipulation* feature, hence it sends full valid operations in advance to prepost. Evaluation does not include this transfer time, but measure preposting and performing RDMA operations.

Among four Hyperloop operations, gWRITE is the only one that fully uses the network media. In this evaluation, only gWRITE is measured. **Note, however, that a key different between Hyperloop and our implementation is refilling consumed operations. Although Hyperloop preposts empty WRs and uses them after manipulation but it seems not to consider refilling consumed manipulated WRs, we evaluated the performance in consideration of refilling consumed operations**.

Our implementation uses window for burst send, hence latency for each operations cannot be measured; while Hyperloop shows 99th percentile, we have no choice but to show average only.

Several configuration factors can impact the overall RDMA performance: window size or message size, group size, etc. We used various configuration values to observe how performance can vary depending on the configurations. Our setup uses window size (wnd): 20, 50, 100, 200, 500, 1,000 and 2,000 (window size cannot be larger due to hardware limitation. Refer to Section 3.1.3.5), and message size: 1KB, 2KB, 4KB, 8KB, 16KB, 64KB, 256KB, and 1MB. Group size is fixed to 3; in other words, three server replicas are chained and a client connects to the replicated system. Following Assise's underlying insight, we put a client and the last server process in a node together. Figure 5.1 illustrates how they are connected. Node 1, 2, and 3 are chained and run one server replica process each, and the client runs on Node 3 and connects the server replica running on Node 3 [1].

We measure the performance for both cases *with* and *without* CPU workloads. Hyperloop evaluated their implementation in busy CPU circumstance running `stress-ng`. We use `stress-ng --matrix 0 -t 0` to get 100% CPU utilization for all cores in every nodes.

	Node 1	Node 2	Node 3
CPU	Intel Xeon Gold 5218 (16C/32T) × 2		
RAM	128GB DDR4 DRAM	128GB DDR4 DRAM	160GB DDR4 DRAM
HCA	Mellanox MT416842 BlueField Integrated ConnectX-5 Adapter (25Gbps)		
OS	Ubuntu 18.04.4 LTS		
SW	Mellanox OFED 4.7-2.3.9.0		

Table 5.1: The specification of nodes used in evaluation.

Figure 5.1: An illustration of connections between server replicas and a client.

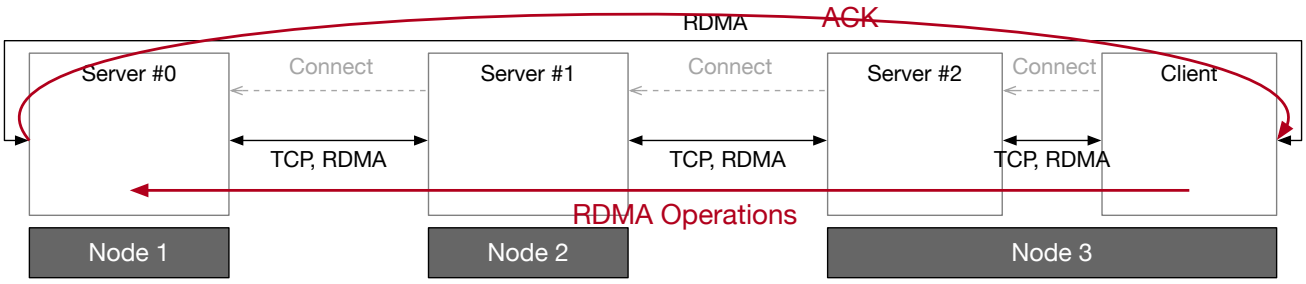
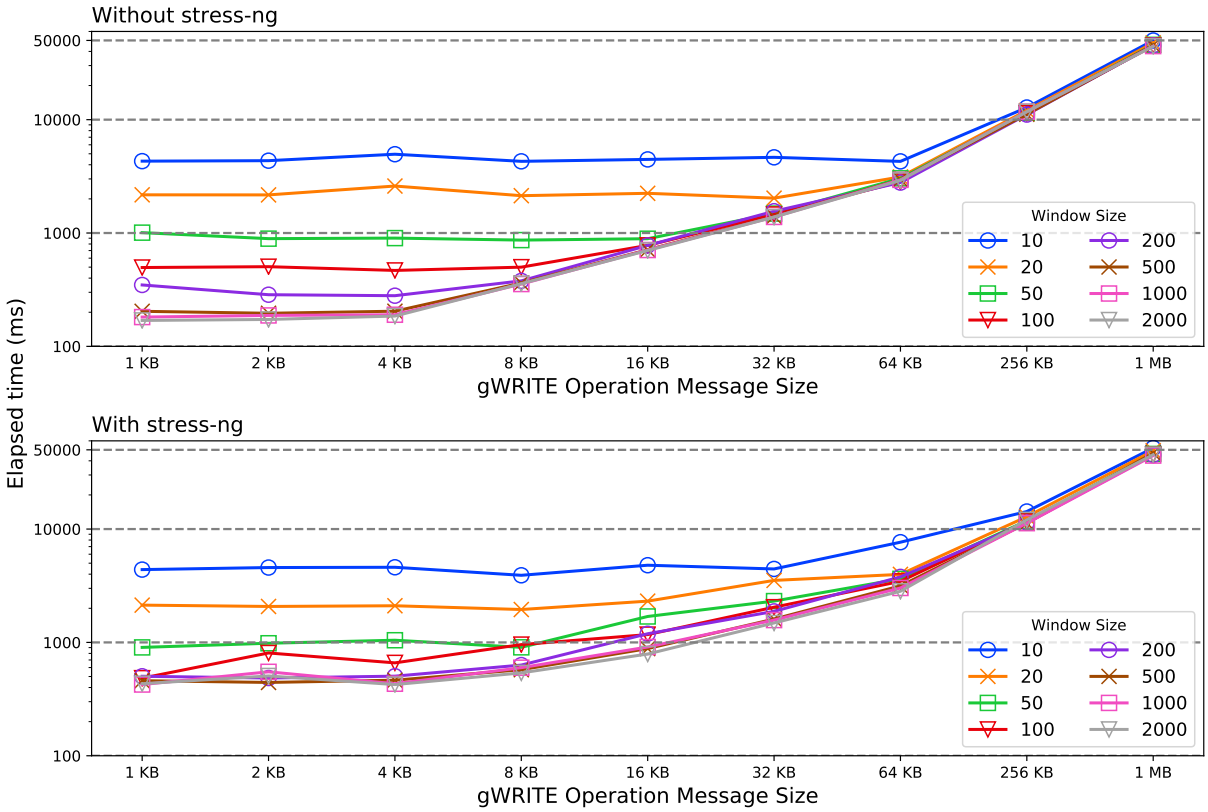


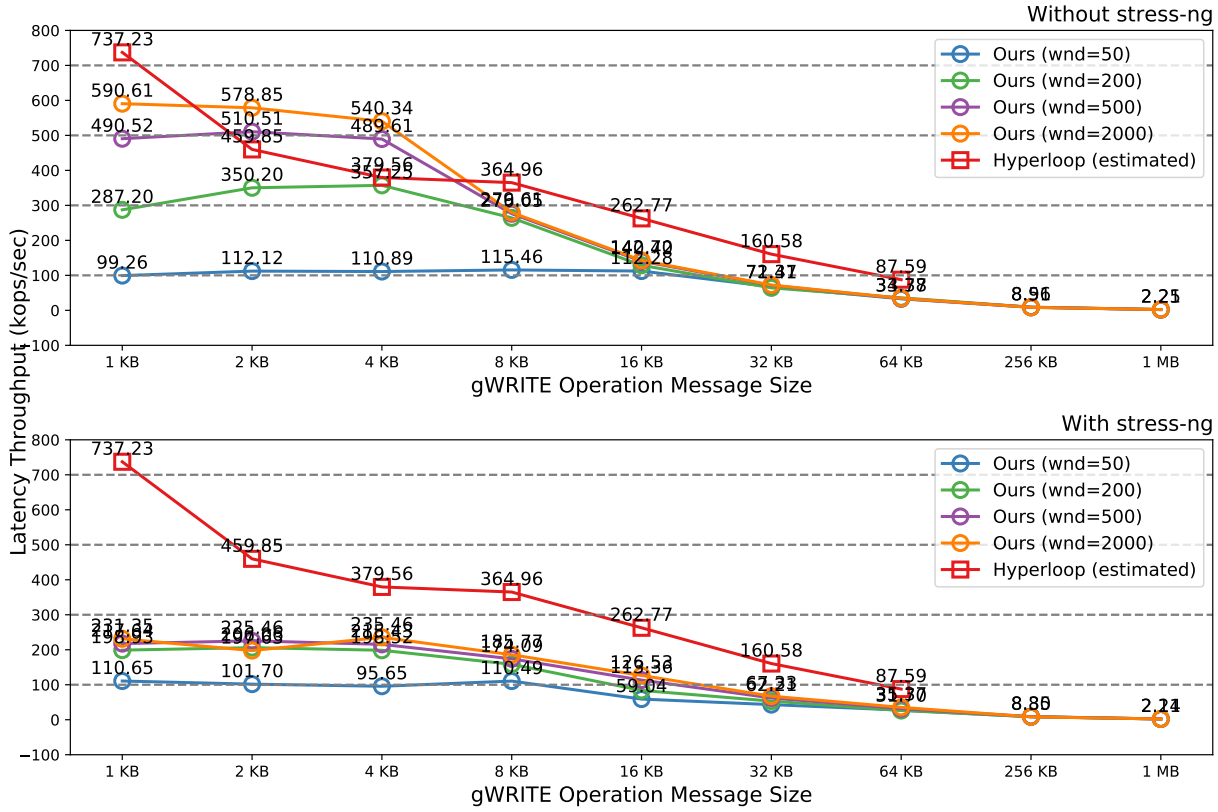
Figure 5.2: Total elapsed time of 100k gWRITE operations.



Time is measured with `std::chrono::system_clock()`, which is one of C++ standard time measurement methods and returns a system-wide wall clock.

Performance is measured for each combination of window size and message size three times, and we peform 100,000 gWRITE in each measurement. The following result is an average of the measurements.

Figure 5.3: Latency throughput of 100k gWRITE operations.



5.2 Results

5.2.1 Elapsed Time

Figure 5.2 illustrates total elapsed time consumed to perform 100k gWRITE operations in various configurations, with and without `stress-ng`. Note that y axis is a logarithmic scale.

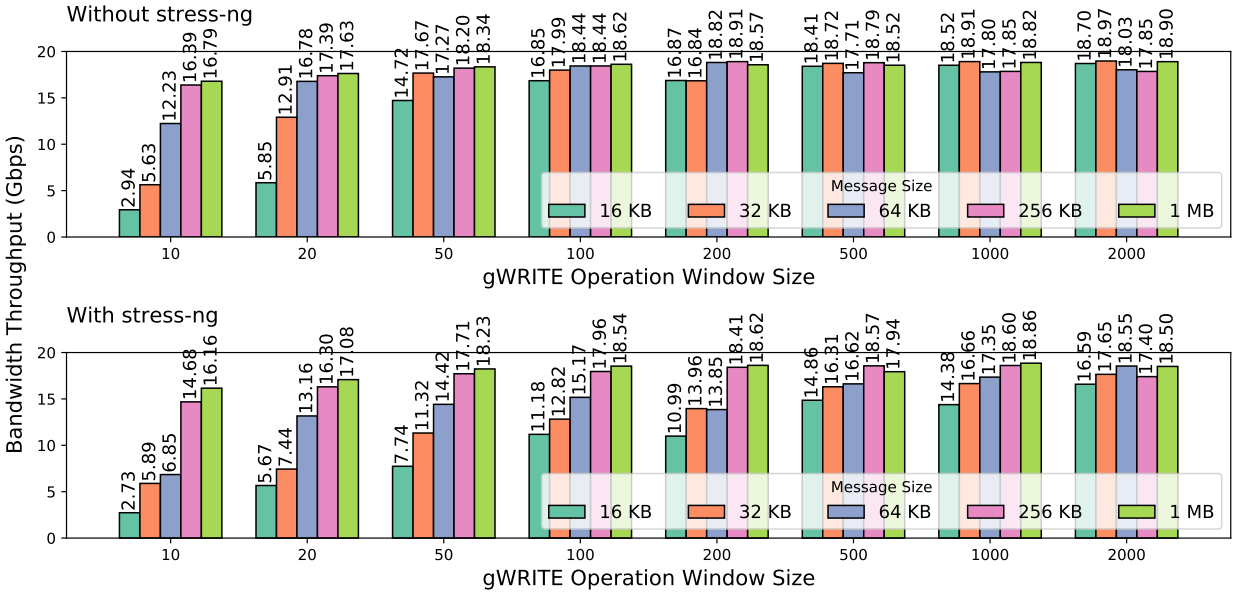
For a set of message size and window size, the elapsed time does not differ and the graphs show horizontal lines, meaning that software overheads are not fully hidden and reposting WRs running in the CPU is in the critical path. By increasing window size, they fully utilize their bandwidth and overheads for reposting WRs are hidden, and the elapsed time increases proportionally to gWRITE message size.

When the CPU is busy, performance for operating small messages is drastically degraded. Seeing the result from a pair of 1KB message size with window size 1000, for instance, the total elapsed time with `stress-ng` is more than two times higher than that without `stress-ng` (425.0ms vs 181.3ms). This is mainly because refilling operations are interfered with high CPU loads coming from `stress-ng`. We further investigate this gap in Section 5.2.4.

5.2.2 Latency Throughput

In terms of latency throughput, we compare our result with Figure 9 from the Hyperloop paper [21] (values for Hyperloop are estimated ones from their figure). Figure 5.3 illustrates the result. Note that direct comparison should be unfair or inappropriate. **We use Mellanox BlueField integrated**

Figure 5.4: Bandwidth throughput of 100k gWRITE operations.



ConnectX-5 25Gbps RoCE adapters, while Hyperloop uses Mellanox ConnectX-3 56Gbps Infiniband adapters. As latency throughput is calculated as $total\_kops\_num/total\_elapsed\_time$, faster adapters would typically show less elapsed time and better latency throughput.

Considering that Hyperloop runs their own implementation with `stress-ng`, our performance may not look good even considering the difference of adapters. However, the difference mainly comes from overheads for refilling operations; In *without stress-ng* case, our implementation show similar performance with Hyperloop despite of using slower adapter, even considering overheads from refilling overheads. Further explanations will be presented in Section 5.2.4.

5.2.3 Bandwidth Throughput

Proper configurations that network can be fully utilized can be found in Figure 5.4. The maximum bandwidth throughput is around 19Gbps. Window size that shows over 17Gbps bandwidth throughput would be a good choice for each message size, e.g., wnd=1000 for 32KB message, in terms of bandwidth throughput. Our implementation shows 76% over the maximum link bandwidth (25Gbps). This may be due to Hyperloop's inborn overheads (overheads from processing WAIT WRs) or ours (refilling operations), etc.

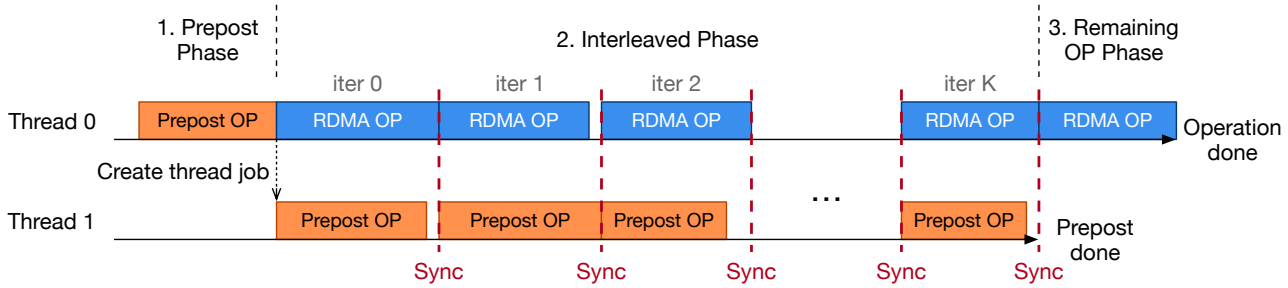
With `stress-ng` again, the maximum bandwidth throughput does not differ: ∼19 Gbps. But only larger messages and larger window size can achieve the peak performance due to increased overheads.

5.2.4 Explaining Performance Gap

In Figure 5.2, 5.3, and 5.4, we observe the decreased performance due to CPU interference coming from `stress-ng`. In this section, we probe that how much responsibility refilling operations has for this performance degradation.

Figure 5.5 reviews how refilling (preposting) operations is implemented in our prototype (refer to Section 3.1.4 and 3.1.5). At each iteration in the interleaved phase, RDMA operations *can be stuck* by

Figure 5.5: An illustration that how Hyperclient uses threads for parellel operations.



delayed operation refill complete response from the servers due to thread sync. At iteration 1 in the interleaved phase from Figure 5.5, thread 0 could not perform the next RDMA operations since it is not guaranteed that corresponding WRs are posted in all server replicas, hence waiting a response from the servers; if refilling operations takes longer, wasted time should increase together and overall performance is decreased.

Thread sync is required to guarantee that corresponding WRs for Hyperloop operations to be posted in all server replicas; hence a thread that performs RDMA operations can only work *after* receiving a response from the servers. Considering this, we measure how many iterations would be delayed by refilling operations. Note that sending refill-operation requests should be negligible, since only 8 bytes are sent to each server through TCP; hence, posting WRs in server CPUs will be a main burden.

We measure refill operation latency (TCP prepost) and RDMA latency (RDMA) respectively for each configuration, and those are illustrated in Figure 5.6. We also calculate in how many windows *refill operations take longer than RDMA operations* among the total operation windows.

In configurations with smaller message size (<32KB) and smaller window size (<50), RDMA operations are usually delayed by TCP refill operations. It becomes worse if message size is less than 4KB and window size is less than 200, since average refill operation latency is higher than average RDMA latency, hence for almost all windows RDMA operations are delayed.

We can also see that usually more time is required to complete both refilling operations and performing RDMA when CPU is busy. Dark bars in Figure 5.6 illustrates average time for job completion without `stress-ng`, and light bars indicates that with `stress-ng`. Light bars are longer than dark bars with matching colors (results with large messages >64KB show similar height but it is due to limited bandwidth throughput); meaning both refilling operations (prepost) and RDMA operations (RDMA) are affected by `stress-ng`.

First, refilling operations requires CPU, which could be interfered by busy CPU workloads without doubt. Second, the interesting observation, RDMA operations take longer as well, can be explained that the client requires CPU to *initiate* RDMA operations, and it seems to be interfered by CPU workloads as well.

5.3 Results: Separating the Client from the Busy Node

Different from Hyperloop, our testbed setup puts the client into the same node that the last node is running together. This may be a reason that RDMA operations also take longer time, not only preposting WRs, because initiating RDMA operations by the client is interfered by CPU workloads.

Due to the limited number of machines available, we use one node dedicated to a client, and the remaining two nodes are chained. We also measure performance with two nodes and a remote client without `stress-ng` to compare the following four results: three nodes without `stress-ng`, three

Figure 5.6: Average prepost and RDMA latency for one window in 100k gWRITE operations with various configurations.

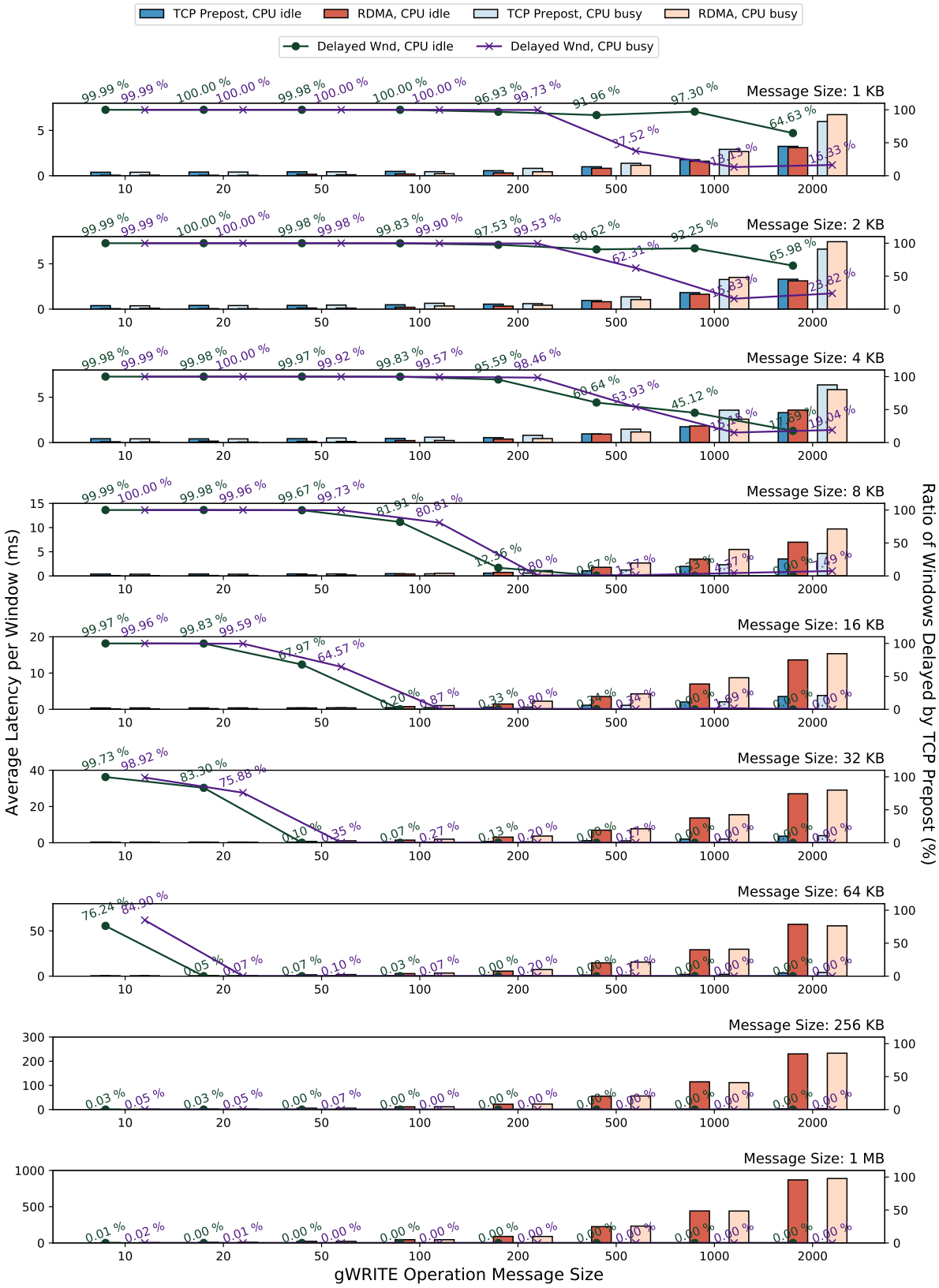
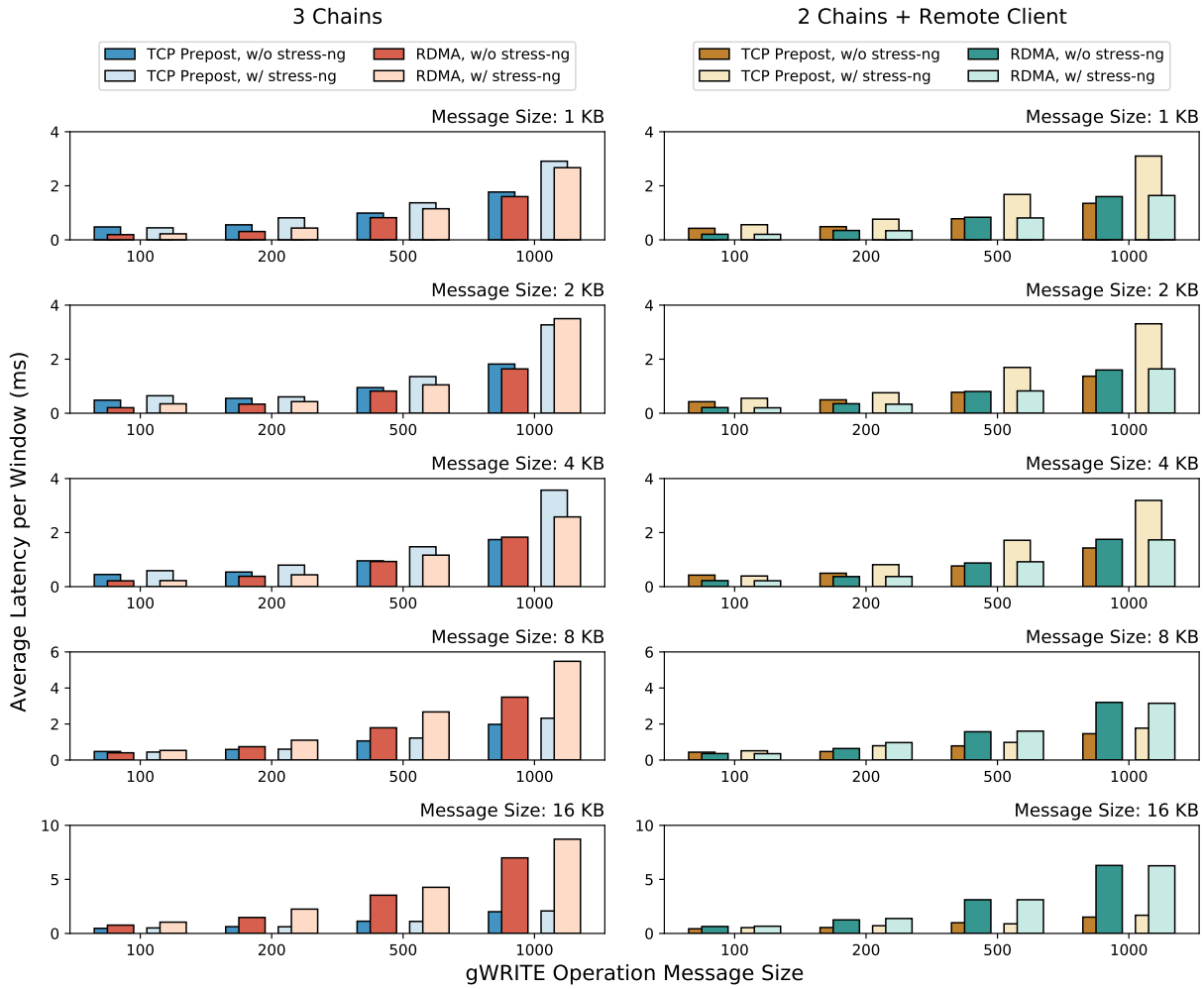


Figure 5.7: Comparing RDMA overheads between three-nodes setup and two-nodes-with-remote-client setup.

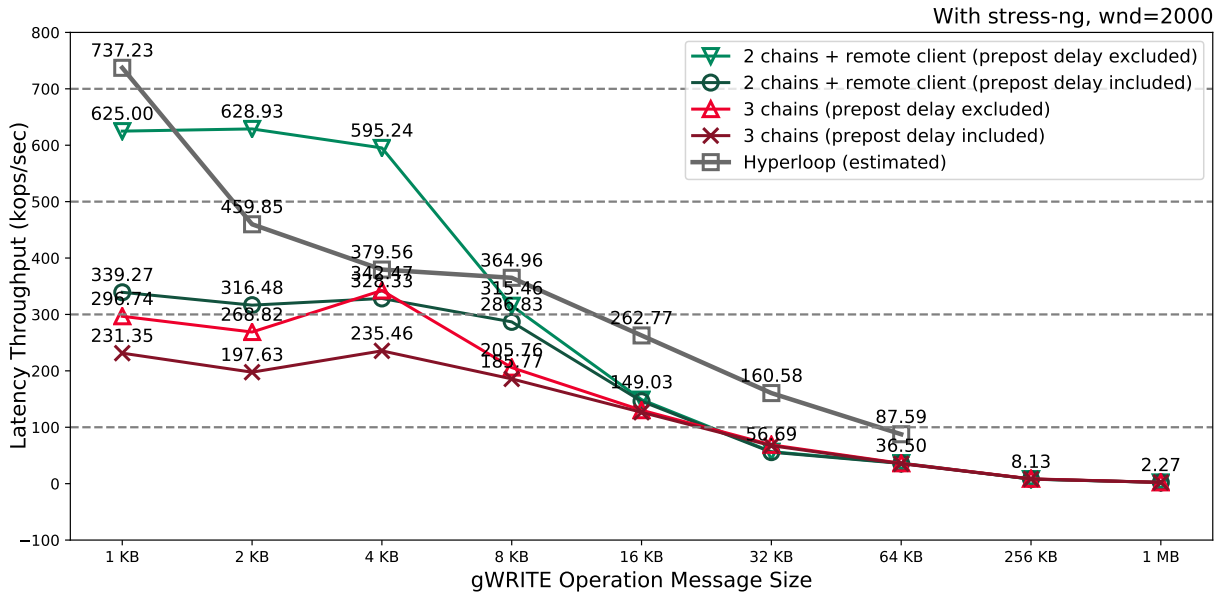


nodes with `stress-ng`, two nodes and a remote client without `stress-ng`, and two nodes and a remote client with `stress-ng`. `stress-ng` runs only in server nodes; hence for *two nodes and a remote client* setup, `stress-ng` does not run on the node that the client runs. From Figure 5.6, increased time for RDMA operations is noticeable when message size is less than 32KB. Therefore, results with messege size < 32KB are shown only.

Figure 5.7 shows difference between three nodes (leftside) and two nodes with a remote client (right-side). Seeing red bars (dark and light) in leftside charts, RDMA operations took longer time when `stress-ng` is in operation. However, green bars in rightside charts do not show any overheads; dark green bars and light green bars show almost the same height, meaning that RDMA operations are not interfered by CPU jobs and not delayed.

With this observation, we again compare latency throughput between three nodes, two nodes with a remote client, and Hyperloop. Figure 5.8 illustrates the compared latency throughput. Even with two nodes and a remote client setup, we still observe overheads for TCP prepost (refilling WRs) in smaller message size (<4KB) in Figure 5.7, since RDMA operations for small amount of data are faster than refilling WRs through the TCP channels. Due to this delay, peak latency throughput cannot be close to Hyperloop's throughput.

Figure 5.8: Comparing latency throughput between two chains plus remote client and three chains.



However, considering our assumption that Hyperloop would be evaluated without refilling WRs in their measurements, comparing it with latency throughput without prepost delay seems to be worthful as well. The result with no refilling WRs (light green plot) closes to the performance of original Hyperloop. The increased throughput can only be observed with smaller message size($<$16KB), because according to Figure 5.7, RDMA operations are delayed only when message size $\leq$ 8KB. With larger message size, overheads for TCP prepost (refilling WRs) are completely hidden by RDMA operations, hence no difference is shown for messages with size $>$8KB in Figure 5.8.

Here, the result of running a client in a busy node (3 chains) without prepost delay is also drawn (light red plot). From the fact that performance difference between the results in 3 chains setup with and without prepost delay is not striking compared to that in 2 chains with a remote client setup, we can put more confidence to our conclusion that initiating RDMA operations from client-side also requires CPU so that is delayed by `stress-ng`, hence can be interfered by heavy CPU workloads.

Chapter 6
Conclusion

In this report, we introduce Infiniband RDMA architecture, illustrate and describe how we implemented Hyperloop, and explore what limitations prevent us from implementing Hyperloop. Hyperloop is conceptually effective, however, seems practically not possible to be deployed. A key contribution of Hyperloop, namely *remote work request manipulation*, would nearly be impossible to be reproduced in our system, hence our implementation is only capable to handle simulated Hyperloop operations with TCP communication. It may be due to lack of our understanding to Infiniband RDMA architecture, so further research regarding deeper analysis of the architecture is necesary. Also, Hyperloop does not mention how they refill consumed WRs for continuous operation. In our evaluation, the job of refilling WRs could be another bottleneck for Hyperloop, reducing its throughput in some cases (small message size, etc).

Reference

[1] Thomas E. Anderson, Marco Canini, Jongyul Kim, Dejan Kostić, Youngjin Kwon, Simon Peter, Waleed Reda, Henry N. Schuh, and Emmett Witchel. Assise: Performance and availability via nvm colocation in a distributed file system, 2019.

[2] Dotan Barak. QP State Machine. https://www.rdmamojo.com/2012/05/05/qp-state-machine/#SQE_state, May 2012. Accessed: April 13, 2020.

[3] Dotan Barak. Tips and tricks to optimize rdma code. https://www.rdmamojo.com/2013/06/08/tips-and-tricks-to-optimize-your-rdma-code/, June 2013.

[4] Dotan Barak. Solicited event. https://www.rdmamojo.com/2014/05/27/solicited-event/, May 2014.

[5] Dotan Barak. Verbs Programming Tutorial. https://www.csm.ornl.gov/workshops/openshmem2013/documents/presentations_and_tutorials/Tutorials/Verbs%20programming%20tutorial-final.pdf, 2014. Accessed: April 13, 2020.

[6] Stephan Bates. iopmem : A block device for PCIe memory. https://lwn.net/Articles/703895/. Accessed: April 20, 2020.

[7] Stephan Bates. p2pmem_pci.c: Add hacky device file for p2pdma. https://github.com/sbates130272/linux-p2pmem/commit/9a5eccff0781f455ac6b2b146007f93c480166ff. Accessed: April 20, 2020.

[8] Stephan Bates. Enabling Remote Access to Persistent Memory on an IO Subsystem using NVM Express and RDMA. In *Proceedings of the 2016 Storage Developer Conference*, SDC'16, USA, 2016. Storage Networking Industry Association.

[9] Stephan Bates. p2pmem: Enabling PCIe Peer-2-Peer in Linux. In *Proceedings of the 2017 Storage Developer Conference*, SDC'17, USA, 2017. Storage Networking Industry Association.

[10] Stephan Bates. Accelerating Storage with NVM Express SSDs and P2PDMA. In *Proceedings of the 2018 Storage Developer Conference*, SDC'18, USA, 2018. Storage Networking Industry Association.

[11] Stephan Bates and Logan Gunthorpe. p2pmem-pci: A (hacky) Linux kernel driver for PCI end points that implement p2pmem on the device. https://github.com/Eideticom/p2pmem-pci. Accessed: April 20, 2020.

[12] Zac Bergquist. hexdump: a header-only utility for writing hexdump-formatted data to C++ steams. https://github.com/zmb3/hexdump, May 2014. Accessed: April 16, 2020.

[13] Jonathan Corbet. The trouble with get_user_pages(). https://lwn.net/Articles/753027/. Accessed: April 19, 2020.

[14] Intel Corporation. *6th Generation Intel Processor Datasheet for U/Y-Platforms Datasheet Volume 2*, February 2016.

[15] Cameron Desrochers. readerwriterqueue: a single-producer, single-consumer lock-free queue for C++. https://github.com/cameron314/readerwriterqueue, January. Accessed: April 16, 2020.

[16] Emil Ernerfeldt. loguru: a lighweight C++ logging library. https://github.com/emilk/loguru, January 2020. Accessed: April 16, 2020.

[17] Daoud Feras and Leon Romanovsky. Asynchronous Peer-to-Peer Device Communication. In *Proceedings of the 13th OpenFabrics Alliance Annual Workshop*, OFA'17, USA, 2017. OpenFabrics Alliance.

[18] Logan Gunthorpe, Bjorn Helgaas, and Stephen Bates. PCI/P2PDMA: Support peer to peer memory. https://lore.kernel.org/patchwork/patch/891005/, February 2018. Accessed: April 20, 2020.

[19] Red Hat Inc. Red Hat Enterprise Linux for Real Time 7 Tuning Guide: TCP_NODELAY and Small Buffer Writes. https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux_for_real_time/7/html/tuning_guide/tcp_nodelay_and_small_buffer_writes, September 2019. Acccesed: April 18, 2020.

[20] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Design guidelines for high performance RDMA systems. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 437–450, Denver, CO, June 2016. USENIX Association.

[21] Daehyeok Kim, Amirsaman Memaripour, Anirudh Badam, Yibo Zhu, Hongqiang Harry Liu, Jitu Padhye, Shachar Raindel, Steven Swanson, Vyas Sekar, and Srinivasan Seshan. Hyperloop: Group-based nic-offloading to accelerate replicated transactions in multi-tenant storage systems. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, page 297–312, New York, NY, USA, 2018. Association for Computing Machinery.

[22] Artemy Kovalyov. Peer Memory Client for IO Memory. https://www.spinics.net/lists/linux-rdma/msg33298.html. Accessed: April 19, 2020.

[23] Mellanox Technologies, Sunnyvale, CA, USA. *RDMA Aware Networks Programming User Manual*, May 2015. Revision 1.7.

[24] Mellanox Technologies, Sunnyvale, CA, USA. *Mellanox Adapters Programmer's Reference Manual (PRM) for ConnectX-4 and ConnectX-4 Lx*, June 2016. Revision 0.40.

[25] PCI-SIG. *PCI Express Base Specification Revision 3.0*, November 2010.

[26] Jakob Progsch and Vaclav Zeman. ThreadPool: a simple C++11 Thread Pool Implementation. https://github.com/progschj/ThreadPool, September 2014. Accessed: April 16, 2020.

[27] Davide Rossetti. GPUDirect: Integrating the GPU with a Network Interface. In *Proceedings of the 2015 GPU Technology Conference*, GTC'15, USA, 2015. Nvidia Corportation.

[28] Marta Rybczyńska. Device-to-Device Memory-Transfer Offload with P2PDMA. https://lwn.net/Articles/767281/, October 2018. Accessed: April 19, 2020.

[29] Darmawan Salihun. System Address Map Initialization in x86/x64 Architecture Part 2: PCI Express-Based Systems. https://resources.infosecinstitute.com/system-address-map-initialization-x86x64-architecture-part-2-pci-express-based-systems/, January 2014. Accessed: April 20, 2020.

[30] Mellanox Technologies. Downloading Mellanox OFED. https://www.mellanox.com/products/infiniband-drivers/linux/mlnx_ofed. Accessed: April 12, 2020.

[31] Mellanox Technologies. Mellanox Article: How To Implement PeerDirect Client using MLNX_OFED. https://community.mellanox.com/s/article/howto-implement-peerdirect-client-using-mlnx-ofed. Accessed: April 19, 2020.

[32] Mellanox Technologies. Mellanox Knowledge Article: Mellanox DPDK. https://community.mellanox.com/s/article/mellanox-dpdk. Accessed: April 12, 2020.

[33] Mellanox Technologies. MLNX_OFED Features Verbs and Capabilities Documentation: Cross Channel. https://docs.mellanox.com/display/rdmacore50/Cross+Channel. Accessed: April 13, 2020.

[34] Mellanox Technologies. Knowledge Article: Physical Address Memory Region (PA-MR). https://community.mellanox.com/s/article/physical-address-memory-region, February 2019. Accessed: Apirl 13, 2020.

[35] Mellanox Technologies. Rocev2 considerations. https://community.mellanox.com/s/article/roce-v2-considerations, April 2019.

[36] Mellanox Technologies. Migration to rdma-core. https://docs.mellanox.com/display/rdmacore50/Migration+to+RDMA-Core, March 2020.

[37] Shin-Yeh Tsai and Yiying Zhang. A double-edged sword: Security threats and opportunities in one-sided network communication. In *Proceedings of the 11th USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'19, page 3, USA, 2019. USENIX Association.