A Design Space for Distributed Producer-Consumer Data Structures Using RDMA

by

Manoj Kumar Sharma

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2020

Author's Declaration

Abstract

Remote Direct Memory Access (RDMA) has become a standard networking technology and is prominently used in high-performance applications. While RDMA can provide both excellent performance and novel capabilities, it can be more difficult than traditional kernel-supported networking to use effectively. RDMA adoption can be simplified through the use of abstractions that target broad classes of applications. In this thesis, we propose *flow structures* as an abstraction targeting distributed applications in which data need to flow directly between producers and consumers. Flow structures are asymmetric distributed producer/consumer data structures in which some servers only produce and others only consume. We present a design space for RDMA implementations of flow structures and illustrate the space using several concrete implementations of a simple flow structure: a FIFO queue. Finally, we present an empirical study that illustrates the design tradeoffs along the dimensions of the design space, and that shows that flow structures abstractions can capture the performance advantages of RDMA while making it much easier to use.

Acknowledgements

I am indebted to my supervisor, Prof. Kenneth Salem without whose guidance and support particularly during the last couple of months, this thesis would not have been possible. I will always be grateful for his invaluable guidance in research and providing me with the progressive look at reading and writing things.

I personally will like to thank Prof. Tim Brecht and Prof. Bernard Wong for being part of early discussions when this project started and later being part of the thesis committee. Also, I thank Benjamin Cassell, Huy Hoang for helping me with all the trivialities.

Over the past two years I have met many people at the University, Data Systems Lab and other places making the list too long to have everyone here. At a general note I will mention a few who were in constant touch with me, beginning with Sai Harsha Singh who knows me since kindergarten and for all his warm talks, Anthony Irudayaraj, Pranjal Gupta for giving me company all the time. Kyle Langendoen, Siddharth Sahu, Amine Mhedhbi for all the small talks in the lab, Hemanth Saxena for clearing the confusion and making me decide to come to Waterloo.

I have the deepest gratitude to my uncle, brother and sisters who were their on the long calls whenever I felt like talking to them. Finally, I will thank my mother, Pramod Devi Sharma for all the sacrifices and pain she had to take and help me with baby steps.

Thank you everyone for being around.

Dedication

This thesis is dedicated to my mother, Late Pramod Devi Sharma without whose support and upbringing, I would not have been the person I am today. It is her motivation and instilled confidence that made me choose this path and the place where I stand.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Over the last decade, Remote Direct Memory Access (RDMA) has seen growing adoption. It provides high network utilization, low latency, and low CPU utilization and overhead because payload buffering, packet processing and dispatch are handled by specialized network interface cards (NICs). Traditionally, a layered hardware/software approach is used for packet transmission, but RDMA enables channel-based communication which involves operating system (OS) intervention only during connection setup and low level context initialization. As a result, RDMA provides kernel-bypass networking. It also offers the ability to access remote memory without any intervention from the remote CPU.

While RDMA can provide both excellent performance and novel capabilities, it can be more difficult than traditional kernel-supported networking to use effectively. RDMA offers a complex, low-level, event-based, asynchronous programming interface, multiple types of operations, and numerous configuration options. For example, an application must choose between reliable and unreliable connection types, must configure event queue pairs, and must manage memory registration. Once communication has been established, the application must choose among one-sided and two-sided operations, coordinate access to remote memory, and handle communication events. These options provide great flexibility but can make it challenging to determine how to get the best performance out of RDMA for a particular class of applications.

Traditional networking is also complex, and this problem is addressed by providing higher-level abstractions, such as stream or datagram sockets, that hide much of the network's complexity behind a relatively simple interface. Ideally, such abstractions should be general enough to accommodate a broad class of applications, but specific enough to enable implementations that make full and effective use of the underlying network technology.

We can use a similar abstraction-based approach to make RDMA easier to use. However, RDMA provides a broader set of capabilities than traditional network interfaces. In particular, it is not clear how to take full advantage of RDMA's remote memory access capabilities behind existing, widely used channel-oriented networking abstractions. But if channel abstractions are too restrictive, what kinds of abstractions *are* appropriate for RDMA?

One body of recent work has focused on *shared state* abstractions, such as shared virtual memories or key/value maps. Applications get a very simple interface, such as Put() and Get(), and RDMA is used to provide a high-performance distributed implementation of shared state across a cluster of machines. Shared state abstractions are well-suited for holding data at rest. They are normally symmetric, meaning that any process can perform any operation supported by the abstraction. Thus, they can support arbitrary indirect communication patterns in distributed applications. This makes them well suited to applications whose communication patterns are not known in advance.

In this thesis, we focus on a different type of shared state abstraction, intended for applications in which data needs to *flow* between known senders and receivers, e.g., in data processing pipelines. Thus, we refer to these as *flow structures.* A flow structure is a producer-consumer abstraction that supports an asymmetric usage pattern across servers. That is, some servers only produce data and others only consume data, while information flows in between them. Of course, flow structures can be emulated using symmetric shared-state abstractions like distributed maps. However, a flow structure implementation can take advantage of its known and restricted usage pattern to optimize its use of the underlying network. Flow structures support *direct* communication among processes in a distributed application.

Many types of flow structures are possible. Perhaps the simplest example is a single-producer, single-consumer FIFO queue, supporting an Enqueue() operation on the producer side and a Dequeue() operation at the consumer. Obvious generalizations would support multiple producers, multiple consumers, or both. Richer generalization can take advantage of the flexibility provided by RDMA. For example, various kinds of *priority queues* allow data to be produced and consumed in different orders. *Aggregation queues* break the one-to-one relationship between Get and Put, and allowing Get operations to return data that depend on multiple Puts. For example, a Get operation might return only the latest item from the queue, discarding the rest, or might return, say, the sum of items currently in the queue. In this sense, flow structures can be thought of as a class of communication abstractions that generalize traditional FIFO channels. These generalizations are made possible by the rich capabilities of RDMA networks.

In this thesis, we take a deeper look at flow structures, and their implementation using RDMA. We argue that there are several broad issues that *any* flow structure implementation must address. The first research contribution of this thesis is the definition of a *design space* for flow structures, based on how those issues are resolved. Unlike other work that considers the impact of low-level RDMA design choices, such as whether to use reliable or unreliable connections, our design space focuses on more abstract architectural questions that arise in RDMA-based implementations. For example, one such issue is how to allow producers and consumers to jointly manage the memory through which the data flows.

After defining the design space, we illustrate it using a case study of alternative implementations of a simple distributed FIFO queue at different points in the design space. Finally, we present an empirical analysis - also based on FIFO queues - of the performance impacts of some of the choices defined by the design space.

The rest of the thesis is organized as follows: Chapter 2 provides a brief outline of RDMA. Chapter 3 presents the context of flow structures and the desrired characteristics of a flow structure. Chapter 4 describes the flow structure design space. Chapter 5 presents the case study, showing two implementations of a simple 1-1 FIFO queue and highlighting the choices defined by the design space. Chapter 6 presents an empirical evaluation of the design choices, using the flow queue implementations. Chapter 7 explores related work pertaining to flow structures and Chapter 8 summarizes the thesis and provides direction for future work.

Chapter 2

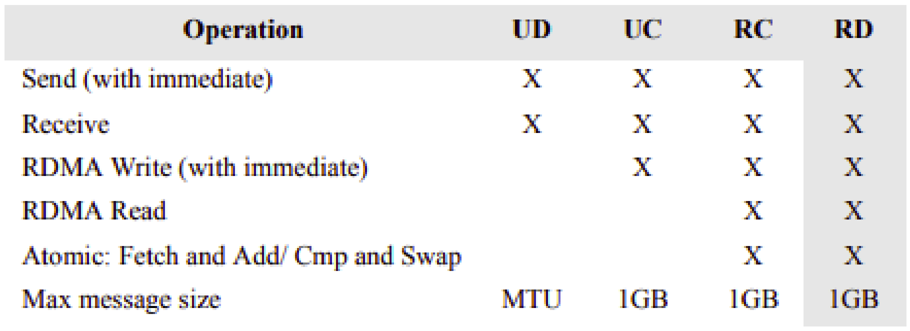
RDMA Background

Infiniband is a widely adopted networking communications standard that provides high throughput with low latencies. It involves setting up channels and using them for communication. Using Infiniband, one can do data transfers without any host CPU involvement. One of the interesting aspects of Infiniband is Remote Direct Memory Access (RDMA), through which a remote machine's memory can be directly accessed.

RDMA is provided by RDMA-enabled NICs called Host Channel Adapters (HCAs) which are specialized to handle network packet related operations like packet transmission, re-transmission, and acknowledgment by themselves. This is possible as the networking stack capability is offloaded to these HCAs with their run-time context spread across memory present on the HCA's die and host's accessible RAM. HCAs work closely with the Direct Memory Access (DMA) controller for accessing local memory. Communication is carried out by means of I/O channels known as Queue Pairs (QP). For any kind of data to be transferred, the source and target addresses (both local and remote, if required) and the transfer size are wrapped into a request which is enqueued to the QP. The hardware accesses the queues and finishes the request without further CPU intervention.

There are many variants [3] of RDMA, including Internet Wide Area RDMA (iWARP), SoftiWARP, and RDMA over Converged Ethernet (RoCE) which is a standard for RDMA over ethernet and provides low latency over ethernet. Each of the variants is chosen based upon the reach of the communication. For example, iWARP is used for wide-area communication and RoCE is used within a data center. Out of all these RoCE is widely used.

RDMA's transport mechanism is exposed to applications through thin APIs known as verbs [3], through which the application directly interacts with the NIC. These verbs



Operation	UD	UC	RC	RD
Send (with immediate)	X	X	X	X
Receive	X	X	X	X
RDMA Write (with immediate)		X	X	X
RDMA Read			X	X
Atomic: Fetch and Add/ Cmp and Swap			X	X
Max message size	MTU	1GB	1GB	1GB

Figure 2.1: RDMA Operations

execute in user space, unlike conventional kernel-implemented message interfaces. Further, these verbs are almost the same across all the variants of RDMA.

2.1 Communication Operations

Like POSIX sockets, RDMA supports different transport modes. Four modes are available:

- Reliable Connection (RC): This is similar to TCP. Packets are delivered reliably and in order.
- Reliable Datagram (RD): This is similar to UDP, except with guarantees for packet delivery. This mode is not supported through the verbs API.
- Unreliable Connected (UC): This provides connection-oriented communication, but packets may be lost.
- Unreliable Datagram (UD): This is analogous to UDP. Packets may be lost or re-ordered.

Figure 2.1 summarizes the verbs (operations) supported over each type of connection. Each operation is initiated at one side of the connection and gets finished by accessing or delivering payload on the other side. For instance, an RDMA write accesses memory at the host that initiates the operation and copies it into memory at a remote server.

The RDMA operations can be classified as *one-sided* and *two-sided*. Send and receive are analogous to socket send and recv operations. They are called two-sided because they

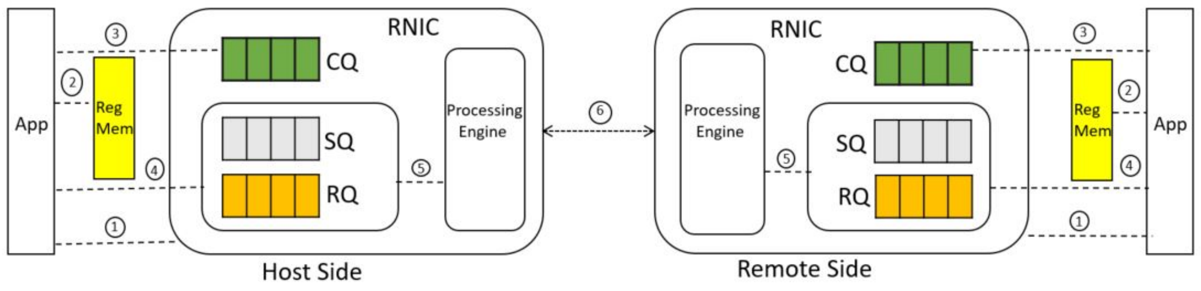


Figure 2.2: Different Resources involved for Connection Setup

involve the participation from the NICs on both sides for communication. That is, the sending side must post a send request and the receiving side must post a receive request before data can be transferred. The other operations, such as read and write, are one-sided as only the initiating side needs to post the operation request.

2.2 Journey of a Byte

This section provides a brief discussion of the steps involved in RDMA communication between machines.

2.2.1 Connection Setup

RDMA communication requires *registered* memory on both sides. Registered pages are pinned in memory so that they do not get paged out by the host's virtual memory system while they are involved in an RDMA operation.

The application's interface to the NIC hardware is by means of a queue pair (QP) and a Completion Queue (CQ). The QP includes a send queue (SQ) and receive queue (RQ) for request dispatch. The completion queue holds NIC generated completion entries known as Completion Queue Entries (CQE) and provides information about completed requests. Queue pairs, memory registration information, and access permissions are associated with a *protection domain (PD)*. A single NIC can be associated with multiple protection domains. Figure 2.2 summarizes these essential components and the numbers indicate the order in which the resources are initialized.

For connection setup, the application determines the available RDMA-enabled devices and their capabilities (Step 1 in Figure 2.2). After confirming the capabilities of the NICs,

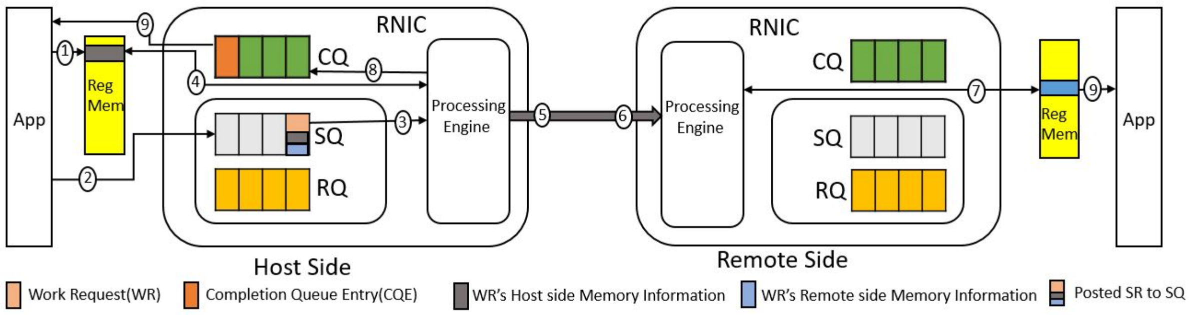


Figure 2.3: RDMA One-sided Write Operation Byte Flow

the application registers memory for use in RDMA operations (Step 2). Next, a CQ pertaining to the connection is created (Step 3) followed by the QP, with specified sizes (Step 4). For establishing a connection between machines, a connection type is chosen and QPs are associated with it and (Step 5). The QPs are then transitioned to a state wherein they are ready to be accessed by hardware implying connection is established (Step 6).

2.2.2 Message Transfer

Once a connection is established, the application communicates by posting *work requests (WR)*. Outbound requests, include write, read, send, write with immediate, and atomics and are posted to the SQ. Inbound operations like receive are posted to the RQ. The NIC retrieves requests from the SQ and RQ and starts acting on them.

Figure 2.3, illustrates how the NIC handles a request for a one-sided write operation. In Step 1, the application modifies (or writes data to be transferred to other side) the local memory as required. Next (step 2), the application will post a WR to SQ. The posted WR will hold the start address of the modified memory and a remote memory address along with access keys(permissions). In step 3, the WR is picked up by the NIC for processing. In step 4, the NIC triggers DMA to fetch the data from local memory, and packetizes and transmits it (step 5) to the remote side. On the remote side, the payload is received in NIC's packet buffers (step 6) along with target address information. Now, the NIC instructs the DMA to write the payload to the memory (step 7). When the data has been transmitted, the sending NIC generates a completion queue event (CQE) and adds it to the CQ (step 8). By polling the CQ (step 9), the application can learn of the request's completion. While the sending is learning of request completion, the receiving application can retrieve the written data from registered memory.

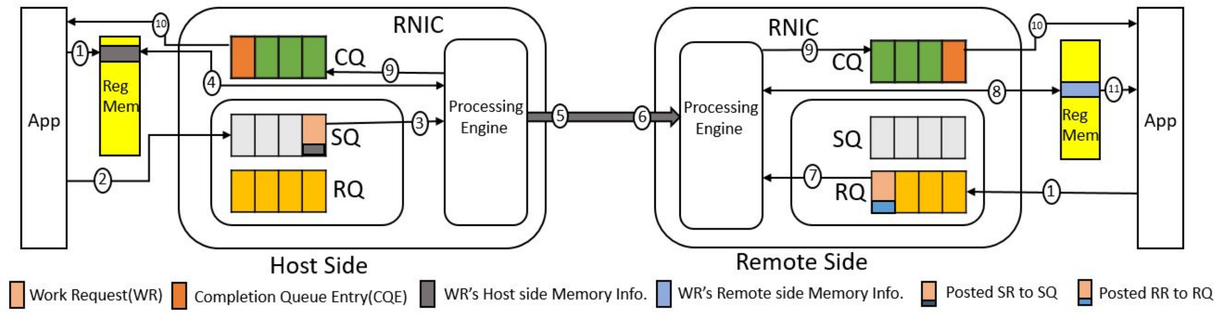


Figure 2.4: RDMA Two-sided Operation Byte Flow

Figure 2.4, illustrates the steps involved in handling a two-sided send-receive operation. On the sender side, the handling is similar to that of the one-sided operations with the exception that the posted WR will not include a target address on the remote machine. To handle the incoming two-sided request, the NIC on the receiving side will consume a receive request from its RQ. The receive request indicates the local target location for the incoming data. When the remote NIC receives the payload it picks a receive request and it initiates a DMA Request to write to the target location provided by the receive request. When the transfer has finished, both the sending and receiving NICs will have added CQEs to their CQs.

RDMA operations can be initiated as *signalled* or *unsignalled* requests. CQEs are generated for signalled requests, but not for unsignalled ones. This allows an application to avoid polling the CQ if it does not care about request completion.

Chapter 3

Flow Structures

As described in Chapter 1, a flow structure is a distributed producer/consumer data structure in which some servers only produce while others only consume. Many kinds of 1-1 flow structures are possible.

In this chapter, we first describe the desired performance characteristics of flow structures in general. We are interested in flow structure implementations that use RDMA to achieve these characteristics. Next, we present some broad assumptions about flow structure implementation which we will use in the discussion in the remainder of thesis. This discussion and the assumptions are intended to be generic in the sense that they apply to any type of flow structure. For concreteness we focus on 1-1 flow structures, but we briefly introduce more general structures at the end of the chapter.

3.1 Desired Properties of Flow Structures

The desired performance characteristics for flow structures include the following.

- **High Throughput:** The maximum throughput of a flow structure should be limited only by the underlying network hardware.
- **Asynchronous Operations:** Flow queues should avoid exposing network latency to produce and consume operations. That is, produce and consume operations should be fast local operations that do not include network delays on their critical paths.

- **Fast Data Transfers:** In a lightly utilized flow structure, the delay between production and consumption of data should be as short as possible, ideally approaching the one-way transit time of the underlying network.

It may not be possible for a flow structure implementation to achieve all of these simultaneously. However, we argue that flow structures are amenable to these requirements because producers and consumers are known in advance. Thus, when new data are produced, their ultimate destination is already known. Thus, a flow structure need not wait before moving data from producers to consumers, provided it has space and bandwidth available to do so.

3.2 Additional Flow Structure Properties

In addition to the three performance characteristics, we also consider two other flow structure properties may be of interest to some applications:

- **Passivity:** We say that a flow structure producer is *passive* if it does not require any CPU involvement except when data are being produced. Similarly, a consumer is passive if it does not require CPU involvement except when data are being consumed. Passive endpoints do not require extra threads or other handlers for events that occur outside of produce/consume operations. Passivity ensures that the flow structure has no impact on the application's (CPU) performance while the flow structure is not being directly used. Endpoints that are not passive are *active*.

- **Network-Awareness:** We say that a flow endpoint (producer or consumer) is *network-aware* if it posts commands to the network interface after the flow is initially established. An endpoint that does not post network commands is *network-oblivious*. Network-oblivious endpoints are possible in flow structures that are implemented using one-sided RDMA operations.

We do not consider fault-tolerant flow structures in this thesis. If the producer or the consumer or the network fails, the flow structure fails and must be re-established. In-flight data will be lost. Flow structures are intended to be used for transient flows between components in a distributed system, like socket connections. Fault-tolerant flows have applications as well, but they are beyond the scope of this work.

3.3 Implementation Assumptions

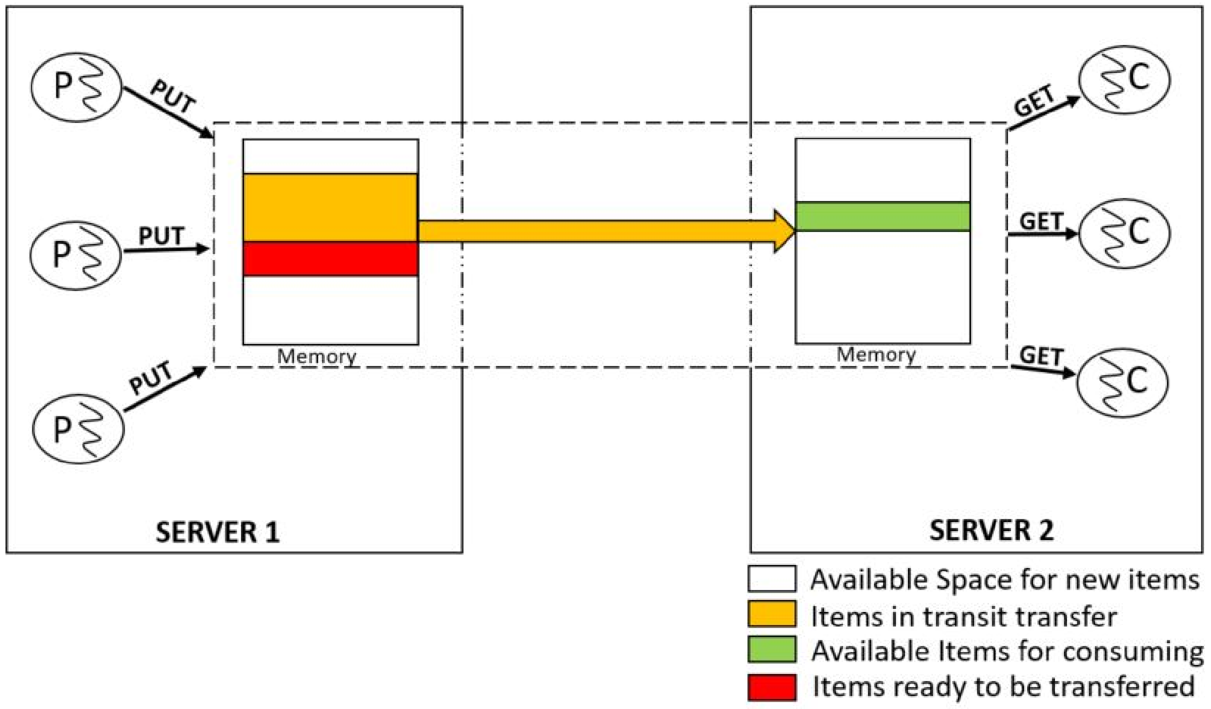

P
P
P
PUT
PUT
PUT
Memory
SERVER 1
Memory
SERVER 2
GET
GET
GET
C
C
C
Available Space for new items
Items in transit transfer
Available Items for consuming
Items ready to be transferred

Figure 3.1: Implementation Assumptions for Flow Structures

Figure 3.1 illustrates our implementation assumptions for 1-1 flow structures where server 1 is the producer machine (or server) where items are produced and server 2 is consumer machine (or server) where items are consumed. Both producer and consumer include a memory buffer. The producer server can support multiple writers, shown as P instances in Figure 3.1. Similarly, the consumer server can have multiple readers, shown as C instances. As shown, writers (1 or more) on the producer side place or Put() items into the producer's buffer. At some point, items are copied directly across the network into the consumer's buffer. From there, readers on the consumer server consume items from the consumer's buffer.

We argue that any flow structure's performance characteristics given in Section 3.1 must fit the assumptions shown in Figure 3.1. The reasons for this are as follows:

- If a flow structure is to achieve fast transfers, it must employ direct communication between producer and consumer, as shown in Figure 3.1. This implies that no intermediate server is involved.

- Unless there is buffer space available on the producer side, producers must wait for data to be transferred to the consumer. Thus, a producer-side buffer enables a flow structure to shield Put() operations from network latency.
- Similarly, without buffer space on the consumer side, consumers must always wait for data to be transferred on demand from the producer. Thus, a consumer-side buffer enables a flow structure to shield Get() operations from network latency.

The availability of buffer space on at least one side also provides a flow structure with the ability to "shape" network traffic to make best use of the available bandwidth. For example, buffering can allow for larger and more efficient data transfers.

3.4 Beyond 1-1

Although Figure 3.1 illustrates a 1-1 flow structure, in general, flow structures can have multiple producers, multiple consumers, or both. For example, Figure 3.2 illustrates a many-to-one flow structure. Such structures can be used, for example, to carry client requests to a shared server, or to fan-in data partitions for joins in a distributed database.

In an N-1 flow structure, each producer maintains its own local buffer and eventually transfers all items to the consumer. In this case, the flow structure must coordinate data transfers to avoid conflicts while writing into the consumer's buffer.

Similarly, one-to-many and many-to-many flow structures are also possible. Possible applications of one-to-many structures are multi-casting information services and log replication. Many-to-many flow structures have applications for join processing or tuple redistribution in distributed parallel database systems.

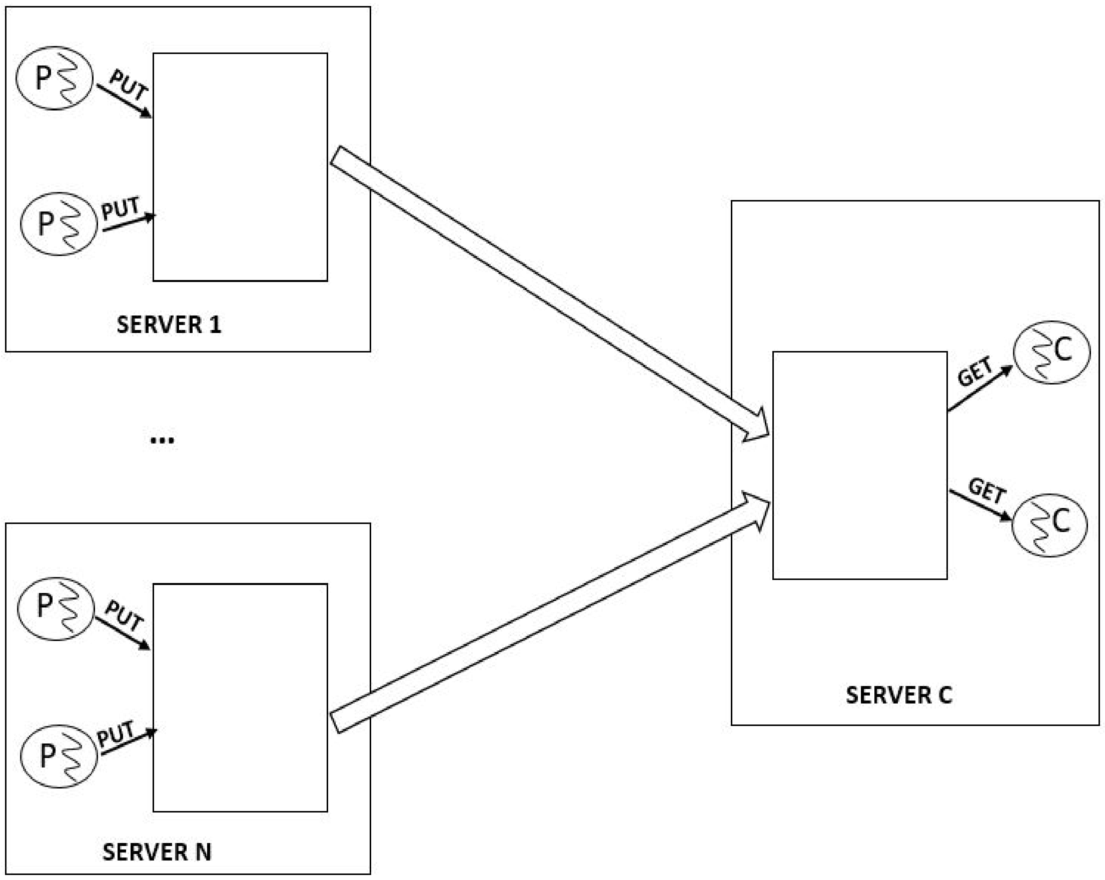


Figure 3.2: A Many-to-One Flow Structure

Chapter 4

Design Space For Flow Structures

Flows are transfers of information that must occur between producer and consumer machines in any flow structure implementation. This chapter will describe three types of flows, corresponding to three different types of information. We define a design space for flow structures based on how RDMA is used to implement each flow.

4.1 Data Flow

The data flow is the primary flow and represents the transfer of application data from the producers to consumers. There are three different ways to use RDMA to implement this flow:

- **Push 1 -** Producers use one-sided RDMA write or write with immediate operations to move data from a producer's buffer to the consumer(s). Thus, data movement is driven by the producer.
- **Push 2 -** Two-sided operations are used to move application data from producers to consumers. Specifically, producers use RDMA send operations and consumers use RDMA recv. Data movement is jointly driven by the producer and consumer, as both must post operations in order for data to move.
- **Pull** - The consumer uses one-sided RDMA reads to move (pull) data from a producer buffer to the consumer buffer. Data movement is driven by the consumer.

Next, we discuss some of the implications of these choices.

Transfer Time: An advantage of the Push-1 and Push-2 approaches is that data transfer can be initiated as soon as data is produced. This is because the data flow is producer-driven, and the producer is aware of newly added data. In principle, this allows transfer time to be as low as the one-way network trip time. In contrast, with pull flows, data transfer cannot be initiated until the consumer is made aware that new data has been produced (We will elaborate on this more in Section 4.2). Furthermore, once the consumer is aware that new data are available, the actual data transfer will require at least the round trip network latency, since the transfer is initiated on the consumer side.

Transfer Size: Transfer size refers to the amount of data moved over the network using RDMA read, write, or send/recv operation. Transfer size can have a significant impact on the transfer bandwidth that the flow queue can achieve. It is easiest to control the transfer granularity with Push-1 and Pull. This is because transfers are controlled unilaterally by one side (producer or consumer), which can choose transfer sizes without coordination. In contrast, transferring data using Push-2 (based on two-sided operations) requires a send operation at the producer and corresponding receive operation at the consumer. Both send and receive must agree on the size of the transfer. Thus, a Push-2 flow structure must either use a fixed (static) transfer size or must implement some mechanism for producer and consumers to agree on transfer size adjustments.

Active Producer: Producer-initiated data flows (Push-1 and Push-2) should have an active producer. Passive producers do no work except when data are being produced and added to the flow structure. If the application stops producing, any data that have been buffered at a passive producer may remain stuck there indefinitely. An active producer, in contrast, can push data even when the application is not producing, allowing it to avoid this problem.

Active Consumer: Similarly, flow structures that use a Pull data flow will require an active consumer. If the application stops consuming, a passive consumer will not transfer new data from the producer. This can result in the producer's buffer filling and preventing new data from being added to the flow even though there is free space available on the consumer side. An active consumer can pull data at any time to avoid this problem.

4.2 Transfer Notification Flow

When a push-based data flow is used, the producer is aware that data have been transferred to the consumer since the producer initiates the transfers. However, there must be some way of notifying the consumer that data have been transferred and are available locally in the consumer buffer. These notifications constitute the *transfer notification flow*.

Conversely, if a pull-based data flow is used, there must be some mechanism for notifying the consumer that data is available in the producer's buffer, so that the consumer can pull it. Thus, for pull-based data flows, transfer notifications tell the data mover (the consumer) that data *is available to be moved*, while for push-based flows transfer notifications tell the consumer that data *has been moved*. In both cases, the transfer notifications flow in the same direction as the data flow, from producers to consumers.

There are different ways to accomplish data transfer notifications:

Watermarking Watermarks are metadata co-located with data to indicate that the location (like a slot in a ring buffer) contains valid data. (These were first used by Liu et al. [25] and they referred to it as piggybacking.) When watermarking is used in a push-based data flow, the producer transfers both data and the watermark to the consumer, effectively piggybacking transfer notification on the data flow. On the remote side, the consumer locally polls for the presence of watermarks in its buffer to learn that a transfer has been completed. For pull-based data flows, the producer adds a watermark when new data is added to its buffer. The consumer can then poll (across the network) for the presence of watermarks to learn whether new data is available for transfer.

Note that for watermarking to work, the consumer must know *where* to poll for the watermark. Thus, it may not be applicable for all types of flow structures.

Push: For push transfer notifications, the producer maintains metadata (e.g., a counter) to track the amount of data that has been transferred. To notify the consumer of transfers, the producer uses one-sided RDMA write operations to update a copy of the metadata at the consumer. The consumer locally polls its local copy of the metadata to learn that data have been transferred (for push-based data flows) or are available to be transferred (for pull-based data flows).

Pull: Pull transfer notifications are similar to Push notifications, except that the consumer pulls the metadata. The producer updates a local copy of the metadata to track the amount of data that has been pushed or is available to be pulled. The consumer uses one-sided RDMA read operations to update its local copy of the metadata and learn about newly available data.

Event-Based: Event-based transfer notifications are possible only for Push-2 data flows. By using signaled RDMA recv operations to receive new data, the consumer can obtain a completion queue event each time a data transfer is completed. Thus, the consumer can poll for completion queue events to learn about data transfers. In effect, transfer notifications are generated directly by the NIC when this approach is used.

Immediate value: This is a variant of event-based transfer notifications that can be used with Push-1 data flows. Instead of using RDMA write operations to implement the data flow, the producer uses write-with-immediate operations. The consumer can poll its local completion queue to obtain the immediate values that are carried by the write-with-immediate operation. (For example, these can encode the amount of data that has been transferred.)

We briefly discuss some of the implications of the different transfer notification techniques.

- Event-based transfer flows (Push-2 and immediate value) generate events on the receiving side. Since the consumer does not know when transfer notifications will occur, this requires that the consumer post adequate number of extra receive requests to RQ and maintain the "pool" of extra receive requests. If this pool is depleted, the producer will be unable to deliver notifications.
- Using pull data flows with watermarked data items requires that the consumer poll across the network, to identify watermarks in the the producer's buffer.
- By combining a Push-1 data flow with either watermarking or push transfer notifications, the consumer side of the flow structure can be designed to be completely passive and network oblivious.
- Similarly, by combining a Pull data flow with Pull transfer notifications, the producer side of the flow structure can be made passive and network oblivious.

4.3 Space Notification Flow

In a flow structure, only the consumer is aware of data being consumed and removed from the structure. Thus, in any flow structure, there must be some way for the consumer to notify the producer about the availability of free space. These notifications constitute the *space notification flow*. This flow carries information *against* the direction of the data flow, from the consumer to the producer.

For flow structures that use push-based data flows, the space notification flow carries information about space availability in the *consumer's* buffer. The producer needs this information so that it knows how much space is available for it to push data into. For pull-based data flows, the space notification flow carries information about space availability in the *producer's* buffer, which is created when the consumer pulls data. The producer needs this information so that it knows how much local buffer space it has available for newly-produced data.

We consider three types of space notification flow:

- **Push:** Under this approach, the consumer maintains metadata to track free space, e.g., counts of data consumed, or data pulled from the producer. It uses one-sided RDMA write operations to push these metadata to the producer, which uses its local copy to determine the amount of space available.
- **Pull:** This is similar to the Push approach, except that the producer updates its local copy of the space metadata by performing one-sided RDMA reads of the producer's copy.
- **Push-2:** This is similar to the Push approach, but the consumer uses send operations to push space meta-data to the producer. The producer must post receive operations to receive the space notification flow.

4.4 Design Space

Every flow structure must implement all three flows. With the exception of event-based transfer notifications, which can only be used with Push-2 data flows, the implementation choices for each flow are orthogonal. Thus, the full design space defines 33 different design categories, differing in how producers and consumers use RDMA to communicate. As part of this thesis we do not iterate through every design rather we pick two categories for demonstrating and use few other designs for evaluating the implications of design choices.

Further, each point in the design space can be implemented in multiple other ways. For instance in case of push data flows, adjacent items can be batched together as a single write known as the batched data flow . Another way could be to use scatter gather to push or pull items. Using any of the design choices will depend upon the requirements of the design.

Chapter 5

Case Study: FIFO Flow Queues

We can have many types of flow structures and each of them can be implemented in a variety of ways, as described in Chapter 4. Here, we will illustrate the design space by presenting two alternative designs of a simple type of flow structure: a FIFO queue. The two designs illustrate two points in the flow structure design space.

A (FIFO) flow queue is a flow structure that allows items to be enqueued on one machine and dequeued on another. Since it is a first-in-first-out (FIFO) queue, items are dequeued in the same order in which they are enqueued. We have focused on FIFO queues because they are widely used and relatively simple to implement, making it easier to illustrate different design choices.

We consider two designs based upon the network awareness property so as to distinguish between having a network-aware versus network oblivious side. The first uses a push-based (Push-1) data flow with watermarking for data transfer notification. Space notifications are pulled by the producer. In this design, the producer is network-aware, but the consumer is network-oblivious. The second design is similar but uses push-based space notifications. Thus, both sides are network-aware.

5.1 Design #1: Pushed Data with Network-Oblivious Consumer

This design uses a Push-1 data flow, with watermarking. Space notifications are pulled by the producer. Both the producer and the consumer maintain local ring buffers, with capacity for S queue items in each ring. The ring buffers are located in pinned registered memory. The producer and consumer are connected through a reliable RDMA connection, which is used for all flows.

The producer and consumer manage the ring buffers using four counters. The counters are:

`consumed`: This counts the number of items consumed from the queue i.e., from the consumer buffer on consumer side. The counter is maintained on the consumer side, where it is incremented whenever an item is dequeued. A potentially stale copy of this counter, referred to as `consumed_p`, is present on the producer side, where it is used to determine the amount of free space available in the consumer's ring.

`produced`: This counter is located on the producer side and counts the number of items produced.

`intransit`: This counter is located on the producer side and tracks the number of produced items for which (asynchronous) transfer to the consumer has been initiated.

`transferred`: This counter is located on the producer side and tracks the number of items for which transfer to the consumer side has been completed.

The flow queue implementation maintains these invariants over the counters:

$$\texttt{consumed} \leq \texttt{transferred} \leq \texttt{intransit} \leq \texttt{produced} \tag{5.1}$$

The difference between `intransit` and `consumed` indicates the number of items that are either in the consumer buffer or in transit. As the producer pushes data, it ensures that this difference is no more than `S`, which is the capacity of the consumer's buffer. Similarly, the difference between `produced` and `transferred` indicates the number of items in the producer ring. The producer ensures that this total is no larger than its ring's capacity, `S`. If the producer's ring is full, further attempts to enqueue new items return "queue full" errors until space becomes available.

Out of the four counters, one of them, `consumed`, is maintained by the consumer but used by the producer. Before the producer pushes new data, it must check the availability of space in the consumer's ring. To do this it uses its local copy of `consumed` and `consumed_p`, which may be stale. Staleness may cause the producer to underestimate the amount of free space available in the consumer's ring. This may delay the transfer of some data, but it will never result in overwritten data in the consumer's ring.

At any given point in time, the producer's ring has room for at most `S - (produced - transferred)` additional items. Thus, the value of the counter `transferred` can be interpreted as "credits" which must be "spent" to produce new items. When the items are transferred to the consumer side, the producer obtains new credits, making room for new items. If the consumer stops consuming then no items are transferred, eventually resulting in the producer's ring becoming full, implying all the credits are exhausted. Similarly, the consumer side has space for `S - (intransit - consumed)` items. Based on this value, the producer side pushes items to the consumer side. Any extra pushes beyond this limit are not done to avoid memory overwrites. Here, the value of the counter `consumed` acts as "credits". As a result of the space notification flow, the value of `consumed` is reflected on the producer side so that the producer can use it for accounting for space availability in the producer's ring.

5.1.1 Implementing the Flows

Figure 5.1 shows the producer side of the flow queue, which implements the enqueue operation. This implementation is purely passive, meaning that the producer's CPU does no work outside of the enqueue operations. Figure 5.2 shows the consumer side, which implements dequeue. Here, the consumer implementation is both passive and network-oblivious. That is, once the queue has been initiated, the consumer does not issue any RDMA commands.

In the following, we describe out how these implementations provide each of the three information flows that define our design space. The implementations use one-sided RDMA read and write operations. We use left arrows to define read operations and right arrows for write operations. For example, at line 20 in Figure 5.1, the notation

$$\texttt{consumed\_p} \Leftarrow \texttt{consumed}$$

represents a one-sided RDMA read operation that reads the value of the remote `consumed` variable into the producer's `consumed_p` variable. Double arrows ($\Leftarrow$ or $\Rightarrow$) like the one shown above represent synchronous operations, while single arrows ($\leftarrow$ or $\rightarrow$) represent asynchronous operations. Thus,

$$\texttt{Qp [intransit \% S]} \rightarrow \texttt{Qc[intransit \% S]}$$

at line 30 represents an asynchronous write from the producer's ring (`Qp`) into the consumer's ring (`Qc`). Note that the right-hand sides of these operations will refer to remote variables.

```
1  int produced = 0; // number of items produced
2  int intransit = 0; // number of items sent to consumer
3  int transferred = 0; // number of items received by consumer
4  int consumed_p = 0; // mirrors consumed at consumer
5  Qp[S]; // bounded item buffer, capacity S
6
7  def ispfull(): return produced - transferred == S
8  def iscfull(): return intransit - consumed_p == S
9  def ispempty(): return produced == intransit
10 def chk_lazyrefresh():
11    return ((transferred - consumed_p) / S) * 100 > THRESHOLD_FACTOR
12
13 def enqueue(item):
14    var status = SUCCESS
15    while (e = get_event() != NULL):
16       if (event_t == WRITE): transferred++
17    if (ispfull()): // producer at capacity
18       status = QUEUE_FULL
19       if (iscfull()): // consumer at capacity
20          consumed_p ⇐ consumed // forced refresh
21    else if (chk_lazyrefresh()):
22       consumed_p ← consumed // lazy refresh
23    while (!ispempty() and !iscfull()):
24       Qp [intransit % S] → Qc[intransit % S]
25       intransit++
26    if (!ispfull()):
27       Qp[produced % S] = Watermark(Item)
28       produced++
29    if (!iscfull()):
30       Qp [intransit % S] → Qc[intransit % S]
31       intransit++
32    return status
```

Figure 5.1: Design #1, Producer Side

Data Flow: This design uses a push-based data flow. Thus, the producer initiates data transfers. Whenever a new item is enqueued, the producer checks whether there is room on the consumer side for any untransferred items in the producer's ring. If so, it pushes as many of those items as possible to the consumer, at line 24 in Figure 5.1. After

```
1  int consumed = 0; // number of items consumed
2  Qc[S] // bounded item buffer, capacity S
3
4  def dequeue(item):
5    if (Qc[consumed % S] is Watermarked): // Watermark Check
6       item = Qc[consumed % S]
7       Qc[consumed % S] = NULL
8       consumed++
9       return SUCCESS
10   return QUEUE_EMPTY
```

Figure 5.2: Design #1, Consumer Side

adding the new item to its ring, the producer immediately transfers that item (line 30), if space is available at the consumer. Otherwise, the new item remains in the producer's ring until it can be transferred during a subsequent enqueue operation.

This data flow implementation uses a separate RDMA write operation to push each data item. When items are small, this will lead to inefficient use of the network. A more network-efficient queue implementation can instead push items in batches. In Chapter 6, we consider both batched and unbatched versions of this queue.

In a simple and purely passive push-based design like this, an item can potentially remain in the producer's ring indefinitely, even after the consumer's ring has been drained. This illustrates why queues with push-based data flows should have active producers. With an active design, items left in the producer's ring could be pushed without waiting for the next enqueue operation, e.g., as soon as space becomes available in the consumer's ring.

The progress of the data flow is tracked by the `intransit` and `transferred` counters. `intransit` is incremented whenever an item transfer is initiated. `transferred` is incremented when an item transfer is completed. So that it can learn about transfer completions, the producer uses signaled RDMA write operations to push the data. These result in completion events. The producer checks the completion queue for these events at the beginning of every enqueue request (line 15), and increments `transferred` accordingly.

Transfer Notification Flow: This design uses watermarking for transfer notification. When a new item is enqueued, the producer watermarks it (Figure 5.1, line 27). Thus, when items are pushed to the consumer, their watermarks are carried along with the items. On the consumer side, the dequeue operation polls for the presence of a watermark in the next slot in the consumer's ring to determine whether it contains a pushed item (Figure 5.2, line 5). Once the item has been consumed, the producer clears its watermark from the ring.

Watermarking has been used in a variety of previous systems that use RDMA [12, 25, 21]. It works in this case because the consumer's buffer is organized as a ring. Thus, the consumer knows where in its ring the next data item should appear, and it can poll that location for a watermark.

If watermarks were not used, the consumer could determine how many items were available in its ring buffer by comparing the `transferred` and `consumed` counters. However, this would require a copy of the `transferred` counter to be maintained on the consumer side, and its value would need to be synchronized with the producer's copy. This would correspond to push- or pull-based transfer notifications in our design space. As a result of using watermarks, these extra counter synchronization operations are avoided.

Space Notification Flow: The space notification flow provides information to the producer about the amount of free space available in the consumer's ring. This design uses Pull space notifications, meaning that the producer pulls the information it needs from the consumer side, using one-sided RDMA read operations. In this case, the relevant information is the value of the consumer's `consumed` counter.

The producer determines the amount of space available in the consumer's ring by comparing its `consumed_p` and `transferred` counters. `consumed_p` is the producer's copy of the `consumed` counter, which is maintained at the consumer. Since `consumed_p` may be stale, the producer determines a conservative lower bound on the actual amount of space available in the consumer's ring.

To minimize the number of times it needs to synchronize `consumed_p` the producer synchronizes only when its conservative estimate tells it that, space available on the consumer may be getting low. Specifically, it synchronizes lazily (Figure 5.1, line 22) when it estimates that space utilization in the consumer's ring rises above a threshold, U_{max}:

$$\frac{\texttt{transferred} - \texttt{consumed\_p}}{S} \geq U_{max} \tag{5.2}$$

Lazy synchronization means that the producer does not wait for the counter to be synchronized, to avoid stalling the enqueue operation that triggered synchronization. Note that if space utilization in the consumers ring is actually high and remains high, the producer will synchronize its counter frequently, potentially on every enqueue operation.

If the producer determines that the consumer's ring is full, and its own ring is also full, the producer performs an additional *synchronous* synchronization of `consumed_p` in an attempt to avoid rejecting an enqueue operation for lack of space. This occurs at line 20 in Figure 5.1

5.2 Design #2: Pushed Data with Network-Aware Consumer

In the previous design, the RDMA operations for all three flows are initiated by the producer. Next, we consider a second design which is similar to the first, except that space notifications are pushed by the consumer, rather than pulled by the producer. In this design, the consumer becomes network aware, and one-sided writes are the only RDMA operations that are used. Figure 5.3 shows the implementation of the enqueue operation at the producer, and Figure 5.4 shows the consumer's dequeue operation.

Since the data flow and transfer notifications are the same as in Design#1, we focus our discussion on the space notification flow. For space notifications, the counter `consumed` is pushed from the consumer side to producer.

One advantage of push based space notifications is that the consumer knows when new space becomes available, since it implements the dequeue operation. Thus, it can avoid pushing the `consumed` counter if its value has not changed. However, when an item is dequeued and the consumer's `consumed` count increases, the consumer must decide how aggressively to synchronize that change to the producer. By synchronizing infrequently, the number of synchronization operations can be reduced. Unfortunately, the consumer does not know how urgently the producer *needs* to know about the availability of new space.

In our design, the consumer synchronizes `consumed` fairly aggressively. Specifically, it lazily synchronizes `consumed` during every dequeue operation, unless there is still an in-flight synchronization from a previous dequeue operation. Synchronization occurs at line 20 in Figure 5.4. The rationale for aggressive synchronization is to minimize the risk to enqueue operations being rejected at the producer due to a perceived lack of space. Futhermore, since this design combines consumer-pushed space notifications with producer-pushed data flow, the RDMA operations for space notification do not compete with data flow operations at the producer's NIC.

In this chapter, we have seen two FIFO flow queue designs, corresponding to two points in the flow structure design space. In Chapter 6 we will consider the performance of these two designs, as well as some additional variations that are not explcitly illustrated here.

```
int produced = 0; // number of items produced
int intransit = 0; // number of items sent to consumer
int transferred = 0; // number of items received by consumer
int consumed_p = 0; // mirrors consumed at consumer
Qp[S]; // bounded item buffer, capacity s

def ispfull():
  return produced - transferred == S

def iscfull():
  return transferred - consumed_p == S

def ispempty():
   return produced == intransit

def enqueue(item):
   var status = SUCCESS
   while (e = get_event() != NULL):
      if (e == WRITE): // consume any completion events
         transferred++
   if (ispfull()): // producer at capacity
      status = QUEUE_FULL

   while (!ispempty() and !iscfull()):
      Qp [intransit % S] → Qc[intransit % S]
      intransit++

   if (!ispfull()):
      Qp[produced % S] = Watermark(Item)
      produced++
   if (!iscfull()):
      Qp [intransit % S] → Qc[intransit % S]
      intransit++
   return status
```

Figure 5.3: Design #2, Producer Side

```
int consumed = 0; // number of items consumed
int consumed_p = 0; // Mirror of producer side consumed
var write_in_progress = false; // Metadata write in progress status
Qc[S] // bounded item buffer, capacity S

def dequeue(item):
  var status = QUEUE_EMPTY

  while (e = get_event() != NULL):
    if (event_t == WRITE):
      write_in_progress = false

  if (Qc[consumed % S] is Watermarked): // watermark check
    item = Qc[consumed % S]
    Qc[consumed % S] = NULL // clear the watermark
    consumed++
    status = SUCCESS

    if (write_in_progress is false):
        consumed → consumed_p
        write_in_progress = true

  return status
```

Figure 5.4: Design #2, Consumer Side

Chapter 6

Evaluation

In this section, we empirically address two broad questions about flow structures.

First, what are the implications of the three dimensions of the flow structure design space for performance? Because many types of flow structures are possible, it is difficult to thoroughly answer this question. To provide some insights, we focus on comparing simple flow queue implementations from different points in the design space, including those illustrated in Chapter 5. We use these comparisons to illustrate general tradeoffs that exist along the dimensions of the design space. We expect that these tradeoffs will occur in some form in any flow structure design.

Second, how does the performance of FIFO flow queues compare to direct communication via sockets? Flow structures are intended to provide easy-to-use abstractions for direct communication between processes, while allowing applications to benefit from the flexibility and performance of RDMA. To quantify the performance benefits, we can compare a flow structure against a similar baseline abstraction that is not implemented with RDMA. Specifically, we compare FIFO flow queues with the Linux implementation of stream sockets, as both provide a simple FIFO channel abstraction. We expect FIFO flow queues to perform at least as well as that baseline.

Most of our experiments were conducted using a pair of Supermicro SYS-6017R-TDF servers, each with one Mellanox 10 Gbps SFP port, 64 GB of RAM, and two Intel E5-2620v2 CPUs each having 6 cores with a frequency of 2.6 GHz. The servers are connected to a Mellanox SX1012 10/40 GbE switch. Both servers run an Ubuntu 18.04.2 server distribution with Linux kernel version 4.18.0.

A few experiments were conducted using a faster 40 GbE network (can extend up till 56 Gbps). For those experiments, each server has single 2.0 GHz Intel Xeon D-1540 CPU with 8 cores, 64 GB of RAM, and one Mellanox ConnectX-3 40 Gbps port. The servers are connected through a Mellanox SX1012 10/40 GbE switch. For the 40 Gbps experiments, the servers run an Ubuntu 18.04.2 server distribution with Linux kernel version 4.18.0.

6.1 Transfer Notification

In this section, we consider the impact of data transfer notifications on the performance of flow queues. We focus on flow queue implementations in which data are pushed from the producer to the consumer using one-sided RDMA write operations. These are called Push-1 data flows, in our design space terminology.

As discussed in 4.2, there are three alternatives for transfer notifications for Push-1 data flows: watermarking, producer-pushed notifications, and consumer pulled notifications. Thus, our experiments compare three FIFO flow queue implementations, each one using a different type of transfer notification. All three flow queues use a Push-1 data flow. All three use Pull space notifications, like the design illustrated in Section 5.1. Thus, they differ only in the way they handle transfer notifications.

Next, we describe each of these flow queue implementations in more detail.

- **Watermark (TN-WM):** This is the flow queue implementation that was presented in Section 5.1. We term it as TN-WM as it uses watermarks for transfer notifications.

- **Push (TN-Push):** This flow queue is similar to the TN-WM queue, but instead of watermarking the producer asynchronously pushes the value of its `transferred` counter to a copy (called `transferred_c`) maintained on the consumer side to notify the consumer that new data has been transferred. The consumer uses `transferred_c` to know how many items are available for consumption from its ring buffer. Specifically, the quantity `(transferred_c - consumed)` is a lower bound on the number of transferred items available for consumption from the consumer's ring.

  A key question for TN-Push designs is how frequently to notify the consumer. Infrequent notification may delay the consumption of transferred data, while frequent notification results in more RDMA operations. Our implementation pushes the updated `transferred` counter after every data push has finished, unless a previous counter push is still in-flight. Thus, at most one transfer notification request will be in flight at any point in time.

- **Pull(TN-Pull)** - This flow queue is similar to TN-Push, in that the `transferred` counter is replicated on the consumer side. However, instead of the producer pushing its counter value to the consumer, the consumer pulls the producer's copy to update its local count.

  Since the consumer always has a stale version of the transferred counter, it can locally determine a lower bound on the number of items present in its local buffer. Our TN-Pull flow queue is designed to asynchronously pull (using a one-sided RDMA read) a counter update when the consumer's lower bound on the number of items in its buffer falls below a pre-defined threshold, unless there is already an in-flight counter update. In our experiments, the threshold is set to 10% of the capacity of the consumer's buffer. In addition, the consumer *synchronously* pulls a counter update if a dequeue operation finds that the consumer's buffer appears to be completely empty.

For each experiment, we initialize a flow queue with buffers of capacity 8192 items at the producer and consumer, connected over the 10GbE network. We choose a fixed item size for each experiment, in the range from 4 bytes to 64 KB. A single driver thread on the producer side sequentially enqueues 1048576 (2^{20}) items, with no sleeps between the enqueues. Similarly, on the consumer side, a single thread sequentially dequeues the 2^{20} items, spin-polling if it finds the queue empty.

On each server, we measure the total time required to enqueue or dequeue all of the items, and record counts of different types of RDMA operations performed at each server. We report (application) *throughput* as the ratio of the total size of the items sent to the total time time (as measured at the producer). Finally, we measure latency and return status of each enqueue and dequeue operation. The return status is either SUCCESS, to indicate that the operation has completed successfully, or FULL (enqueue) or EMPTY (dequeue) to indicate that the operation failed because the queue is full (in the case of enqueue) or empty (in the case of dequeue).

Each experiment is independently repeated ten times, and the reported values (e.g., throughputs) are averages over all runs. Over the 10 runs, each of the reported values e.g., throughput did not vary much and differed by at the most 2% so variability is not shown in the graph.

Figure 6.1 shows the throughput of each flow queue implementation as a function of the item size. For item sizes up to 4 KB, throughput is limited by the rate at which the sequential driver thread can enqueue items into the queue. For larger items, the driver thread is able to saturate the network.

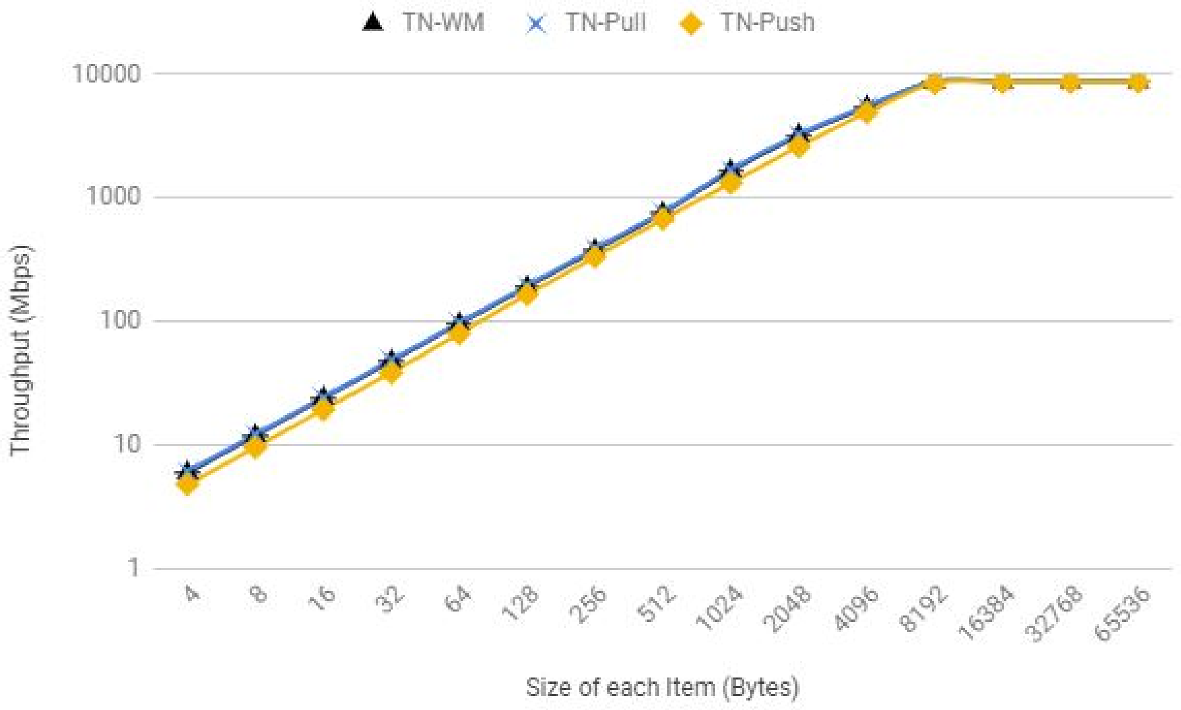


Figure 6.1: Throughput with Different Types of Transfer Notifications

Figure 6.2 shows the latency for successful enqueue operations for small item sizes. Enqueue time is slightly higher for the TN-Push queue than for the other two. This is because enqueues to TN-Push queue need to post additional write operations to the producer NIC for transfer notifications. However, the impact on enqueue time is very small. Enqueues for both TN-WM and TN-Pull have similar latencies as both the designs are the same on the producer side.

Another measure of the overhead of transfer notifications is the *transfer notification ratio*. The transfer notification ratio is the number of RDMA operations performed for transfer notifications divided by the number of items sent through the queue. Figure 6.3 shows the transfer notification ratio for the TN-Push and TN-Pull queues, as a function of item size. The TN-WM queue is not shown, as watermarking does not require separate transfer notification operations.

A flow structure using TN-Push could reasonably have a notification ratio as high as one, by pushing a notification for every item sent. This would double operation load on the NIC. Our implementation has a ratio of about 0.5 for small items, which is still very

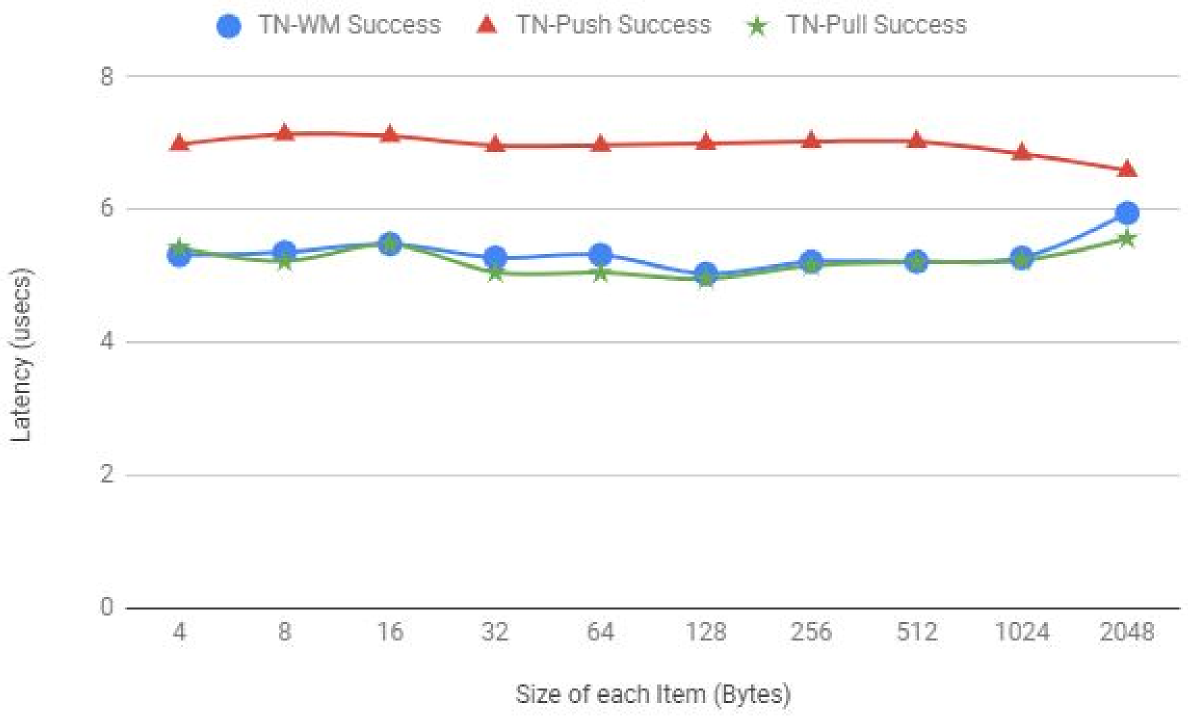


Figure 6.2: Enqueue Latency

aggressive, i.e., one notification for every two items. As we have shown, these extra operations have only a small impact on enqueue latency and a negligible impact on throughput in our experiments. However, our experiments use only a single driver thread, which does not saturate the network unless the items are very large. These overheads could conceivably limit throughput of a more performant concurrent flow structure, since notifications could potentially double the number of operations that must be handled by the producer's NIC.

It is worth noting that the transfer notification ratio for TN-Push drops as the item sizes get larger. This is because notification operations take longer to complete, since they share a request queue with data transfer operations. Our TN-Push queue allows only one in-flight notification operation at a time. This makes notifications operations more efficient (each carries "news" of multiple item transfers), but means that there is a delay before the consumer is informed that new data are available it its buffer. To avoid this problem and maximize the timeliness of transfer notifications, flow structures using TN-Push can employ separate RDMA queue pairs for data flow and transfer notification operations.

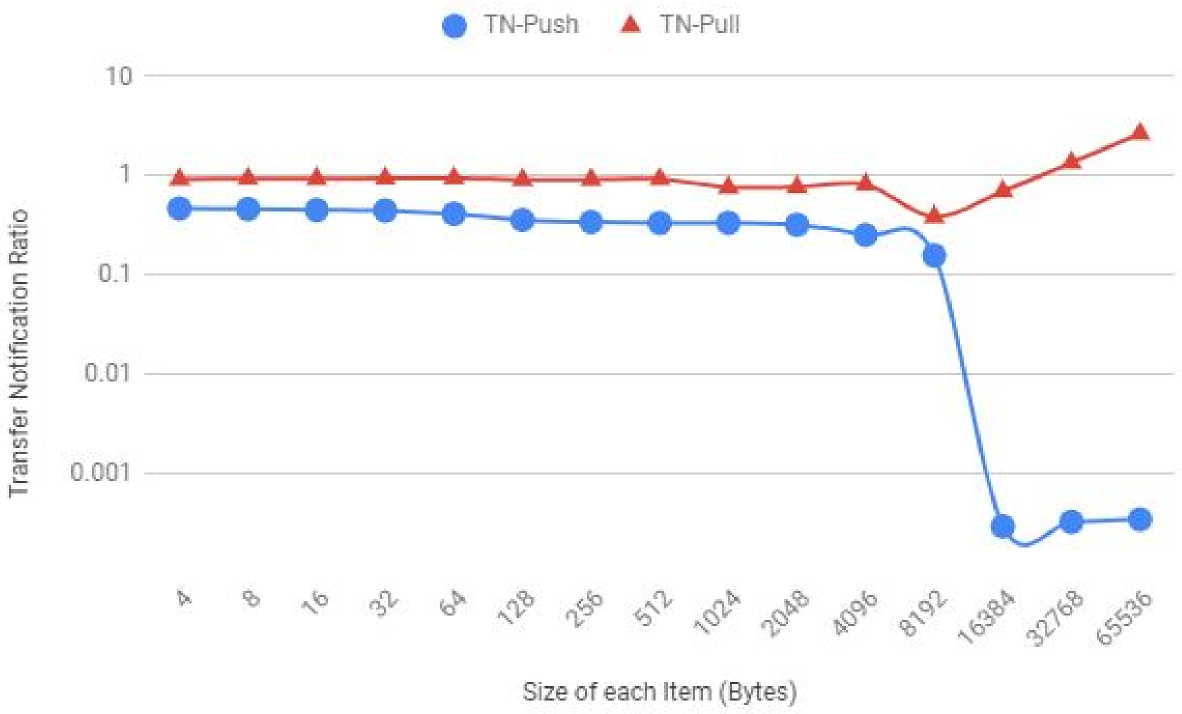


Figure 6.3: Transfer Notification Ratio vs. Item Size

However, we do not try this as part of the experiments and our implementation uses a single QP on both the sides.

Our TN-Pull queue generally has an even higher notification ratio than TN-Push. In this experiment, the throughput bottleneck is the producer or the network, so the consumer buffer is often empty. The TN-Pull queue synchronously reads to check for transfers in that case. If transfers take a long time, as is the case for large items, this might occur multiple times, which is why the ratio increases beyond one for the largest item sizes.

Although TN-Pull results in more transfer notification operations than TN-Push in our setting, those operations are less problematic because they are performed by the consumer's NIC. Thus, they do not compete with data transfer operations at the producer, and do not impact enqueue latency or throughput. Their primary impact is on the latency of dequeue operations at the consumer.

On the consumer side, both successful and unsuccessful (Queue Empty) dequeue operations are slower for TN-Pull than for the other queues. On an average the dequeue latency for TN-Pull is 5 μsecs for item sizes ranging from 4 to 2048. This is because the RDMA read operations for transfer notifications are performed by dequeues. If the consumer's buffer is not empty, reads can be asynchronous, so network latency is not exposed to the dequeue operation. However, if its buffer appears to be empty, a dequeue waits for a synchronous read so that it does not fail to return data that are actually present (but unnotified) in its buffer. On the other hand, for small item sizes ($\leq$2048) the dequeue latencies for TN-WM and TN-Pull are at a sub microsecond level (nanoseconds) and are difficult to measure.

In summary, we can draw several conclusions regarding transfer notifications:

- Watermarking eliminates the need for separate transfer notification operations, and is an ideal choice for flow structures that can take advantage of it.
- TN-Push transfer notifications need to be used carefully if they are combined with push-based data flows, as they compete with data flow operations for NIC capacity.
- TN-Pull offloads notification overhead from the producer's NIC, which may be valuable in combination with push-based data flows. However, care is needed to avoid exposing network latency to dequeue operations when the consumer's buffer utilization is low. One way to avoid this would be to move to an active consumer, which obtains transfer notifications off of the critical path of dequeue operations.

6.2 Data Flows

The data flow is the major flow, as it carries the application data. In Section 4.1, we identified three types of data flow: producer-pushed (Push-1), consumer-pulled (Pull) and two-sided (Push-2). We focus here on comparing the two producer-driven alternatives, Push-1 and Push-2. Although Pull queues are feasible, a Pull consumer cannot start transferring new data until it has been notified of its arrival, and pulling the data will require a full network round trip. Thus, we expect Pull queues to have higher data transit times and high dequeue latencies, at least when the queue is not full. Push-based designs avoid these problems.

Thus, the two queues we consider here are:

- **Push-1:** This is the flow queue implementation that was presented in Section 5.1.

- **Push-2:** This is similar to the Push-1 queue, except that sends and receives are used to move the application data. Transfer notification is event-based, so explicit transfer notification operations are not needed.

  In a Push-2 design, each send operation performed by the producer must be matched by a receive operation posted by the consumer. If there is no receive operation, the producer's NIC is forced to retry the send until a receive operation has been posted by the consumer (or fail the operation). To avoid this problem, the consumer side should maintain a pool of receive requests. The number of receives in this pool is a parameter of our Push-2 implementation. The default value is 64, but we also experiment with pool sizes of 48, 32, and 16 to show the performance impact of an inadequate pool. Note that the maximum pool size is limited by (a) the capacity of the NIC's receive queue, and (b) the amount of memory available in the consumer's buffer. The latter constraint arises from the fact that each receive request in the pool must specify a target location in the consumer's buffer for the incoming data.

  Another potential limitation of the Push-2 design is that each receive request must specify the amount of data to be received, which must match the corresponding send operation. This is not a significant constraint for our simple design, since all data transfers are the size of a single queue item. However, more sophisticated flow structures, such as those that would employ dynamic transfer sizes to improve network efficiency, would need a mechanism for coordinating request sizes between the producer and consumer.

Finally, we note that both the Push-1 and Push-2 queues use Pull space notifications.

Figure 6.4 shows the results of the same throughput comparison experiment we used in Section 6.1. For Push-2 with a reserve pool of 16 we have plotted the error bars showing the maximum and minimum throughput achieved as throughput varies while in the other cases the deviation is within 2% and so have not been reported.

For the Push-2 queue, we show results for experiments with several receive request pool sizes. As was the case in Section 6.1, throughput is limited by the single driver thread at the producer unless the items are large, in which case the 10 GbE network is the bottleneck. Both queues have similar performance when items are small, indicating that the cost of their enqueue operations is similar (about 5 μsec per operation). This is not surprising, since they perform similar operations: Push-1 posts a one-sided write operation on each enqueue, while Push-2 posts a two-sided send operation on each enqueue.

This experiment also shows that an inadequately sized receive request pool at the consumer can limit the peak throughput achievable by the queue. If incoming send packets

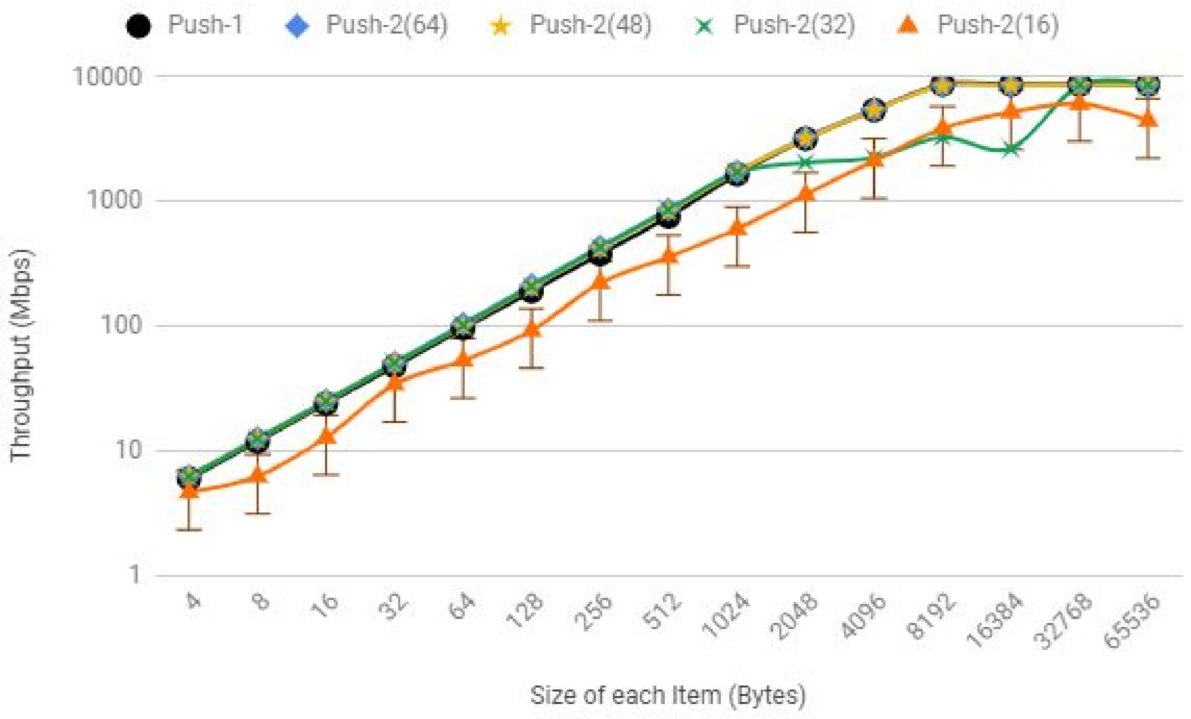


Figure 6.4: Throughput of Push-1 and Push-2 Data Flow Queues

find no receive operations at the consumer's NIC, they will be discarded and the producer has to retransmit, both of which waste bandwidth. We confirmed that sent items were being dropped by measuring the *vport_rx_dropped* [2] counter at the consumer's NIC. It tracks the number of received packets discarded due to the lack of receive buffers. Further, the packet drops and re-transmits are not uniform across all the runs for same size which can give us varying throughput results. Figure 6.4 shows Push-2(16)'s average throughput and minimum, maximum achieved throughput by means of error bars.

The dequeues are more expensive for Push-2 in comparison to the Push-1 design. On average, for item sizes ranging from 4 to 2048, Push-2's successful dequeue operation's latency is 2.5 μsecs whereas for Push-1 dequeue operations are fast and have latency values at sub microseconds level(nanoseconds). This is because the Push-1 consumer is network-oblivious, while the Push-2 consumer must post receive events and handle completion events from the consumer's completion queue. Even a dequeue request that returns EMPTY is relatively expensive, since it must check the completion queue to confirm that no new data have recently arrived.

To sum up, we draw the following conclusions about push-based data flows:

- Provided the receive operations are managed properly, we can expect similar throughput from Push-1 and Push-2 designs. Enqueue latencies should be similar in both cases, but dequeue latencies will be higher for Push-2 designs.
- Using two-sided operations adds CPU overhead on the consumer side, since the consumer must post RDMA requests and handle completion events.
- Push-2 designs are more complicated, as they require management of a receive request pool at the consumer, and coordination of data transfer sizes between the two sides.
- It may be easier to implement a blocking dequeue operation using a Push-2 design, since Push-2 designs result in receive events being generated by the consumer's NIC. These events can be used to unblock waiting consumers – although our Push-2 queue does not do this. In contrast, a Push-1 consumer must poll for notifications of data transfers.

6.3 Space Notification Flow

As discussed in Section 4.3, we have three ways of doing space flows – producer-pulled (Pull), consumer-pushed (Push) and two-sided (Push-2). In this section, we compare them.

We implemented queues using each of the three space notification techniques. All three queues use Push-1 data flows and watermarking for transfer notifications.

- **Pull:** This is the queue that was described in Section 5.1. Space notifications are pulled by the producer side.
- **Push:** This design, is like the Pull design, except the consumer asynchronously pushes space notifications to the producer side. As the case for the Pull design, each space notification updates the producer's copy of the `consumed` counter, thereby allowing the producer to determine the availability of space on the consumer side. In our Push queue, the consumer pushes the value of `consumed` each time it consumes another item, unless a previous space notification push is already in-flight.

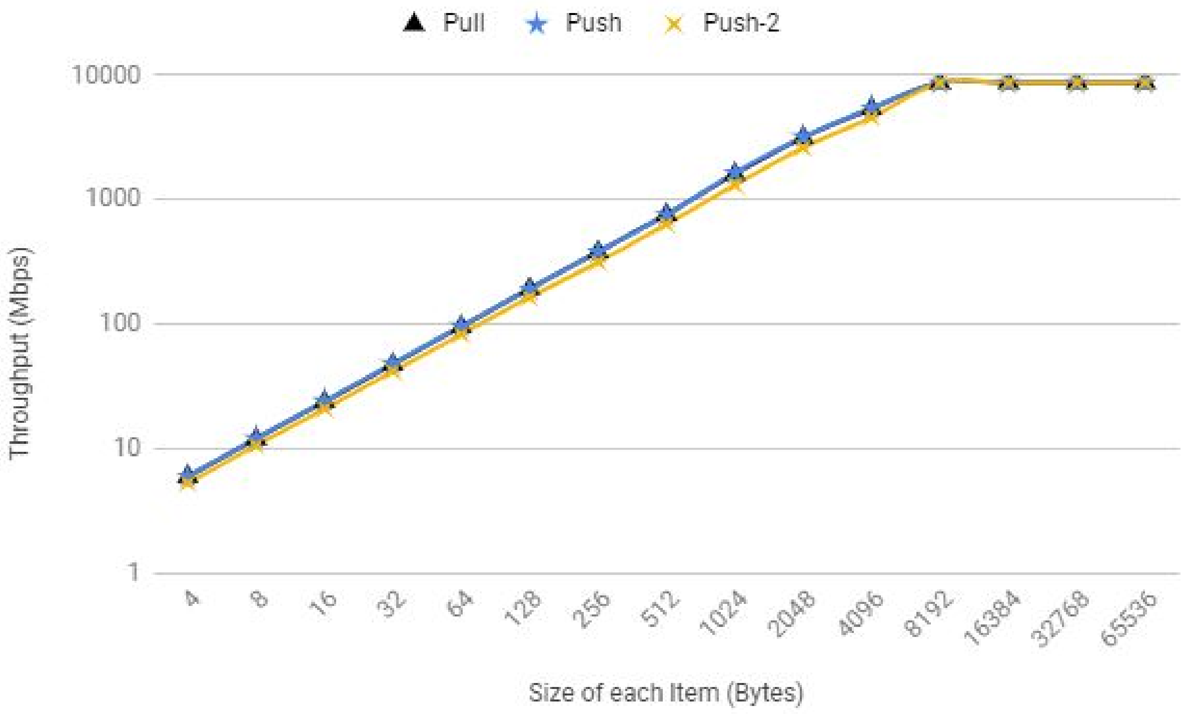


Figure 6.5: Throughput with Different Types of Space Notifications

- **Push-2:** This is similar to the Push-1 design, except that the consumer uses two-sided send operations to push the updated value of its `consumed` counter to the producer side. The producer must post corresponding receive operations in its QP in order to get these updates. In our design, the producer maintains a pool of 64 receive requests in its QP, for space notifications. The design of this Push-2 is similar to the design proposed by Liu et al. [25] for RDMA-enhanced MPI. However, that design extends the producer's ring buffer with non-registered reserve buffers in case of overflows.

Once again, we first ran the throughput experiment described in Section 6.1 to compare these three designs. Figures 6.5 and 6.6 show the throughput and enqueue latency for each of the three queues.

The Push-2 design has somewhat higher enqueue latency than the other two, and hence slightly lower throughput for small items, when throughput is limited by the enqueue rate of a single thread. In the Push-2 design, the producer must post receive operations and also

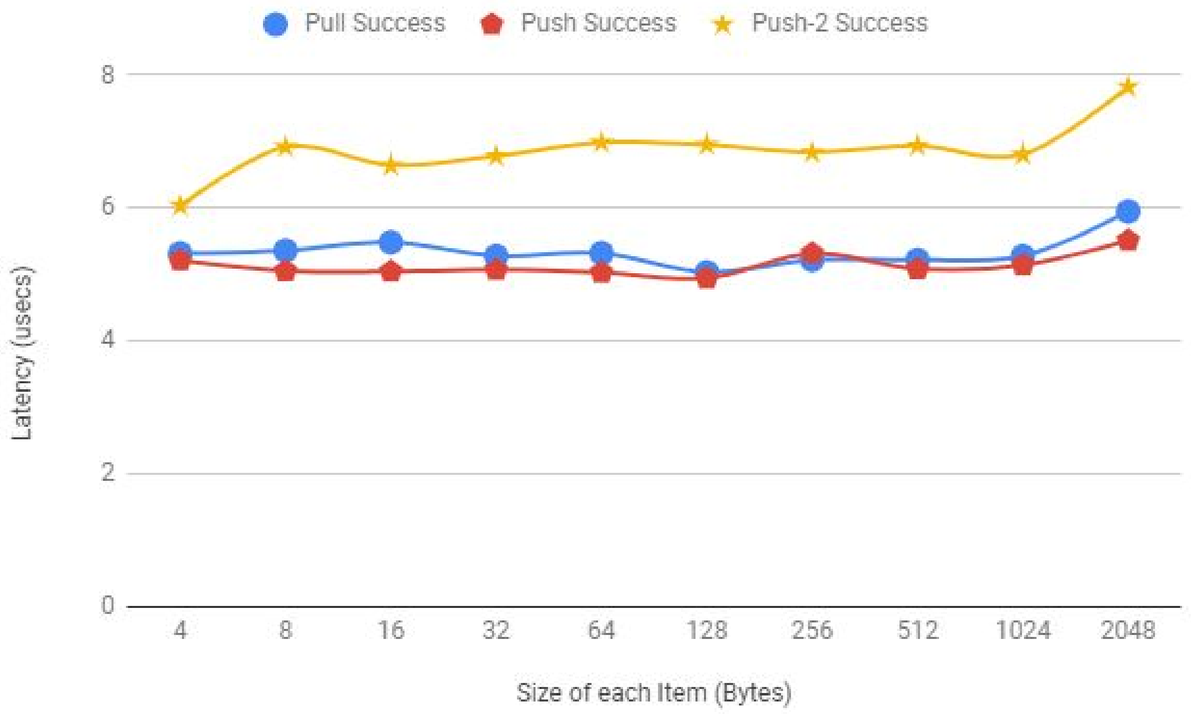


Figure 6.6: Latency for Enqueue Operations with Different Space Notification Techniques

poll the completion queues for the completion of those events, in addition to the RDMA operations it performs for the data flow. There are many space notification events because the consumer sends space notifications aggressively.

The Pull design also requires the producer to post extra RDMA operations for space notification, beyond those required for the data flow. For Pull, these operations are RDMA reads. However, there are many fewer of these for Pull than for Push-2, so the impact on enqueue latency is negligible. The Pull queue pulls new space information only when it thinks the consumer's buffer is close to full, which occurs only after the producer has transferred almost enough data to fill the consumer's ring buffer, i.e., infrequently.

Figure 6.7 shows the *space notification ratio* for each of the queues. This is the total number of space notification pushes (or pulls) divided by the number of items transferred. Both Push and Push-2 have much higher ratios than Pull. They push aggressively because they do not know how full the producer thinks the consumer's buffer is.

Figure 6.8 shows the dequeue latencies for the Push and Push-2 queues. For the pull design, the latency values are at sub microsecond level(nanosecond) and are not shown

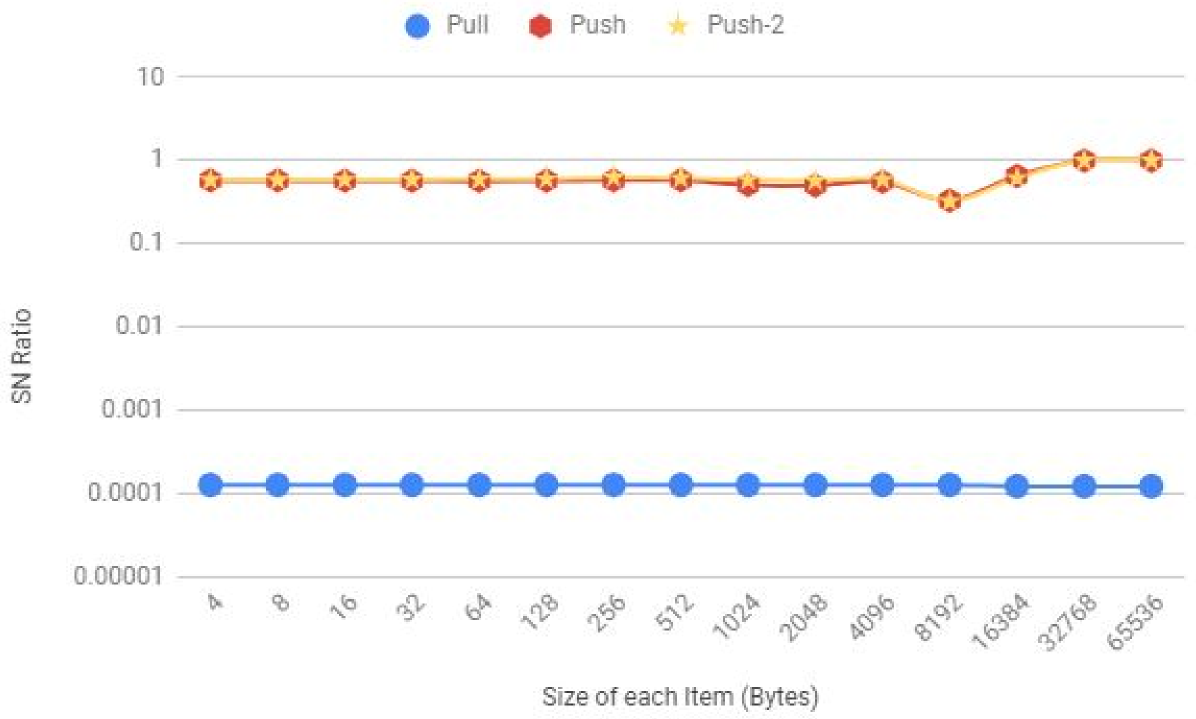


Figure 6.7: Space Notification Ratio for Different Space Notification Techniques

in the figure. Both Push and Push-2 have higher latency because the dequeues perform RDMA operations to send space notifications. In contrast, in the Pull design the consumer is network-oblivious and very fast.

We summarize our conclusions about space notification as follows:

- The Pull design (when combined with a push-based data flow) offers very fast network-oblivious dequeue operations. Space notification operations do compete with the data flow at the producer's NIC, but the producer only needs to pull space information when it thinks the consumer's buffer is nearly close to full. Unless the consumer is the bottleneck and the queue is very full, these pulls will be infrequent. If the queue is very full, pulls could become more frequent and could compete with data transfers, but this will occur only if the consumer, rather than the network, is the bottleneck for the flow structure.
- Push designs avoid any competition with the data flow at the producer side, thus

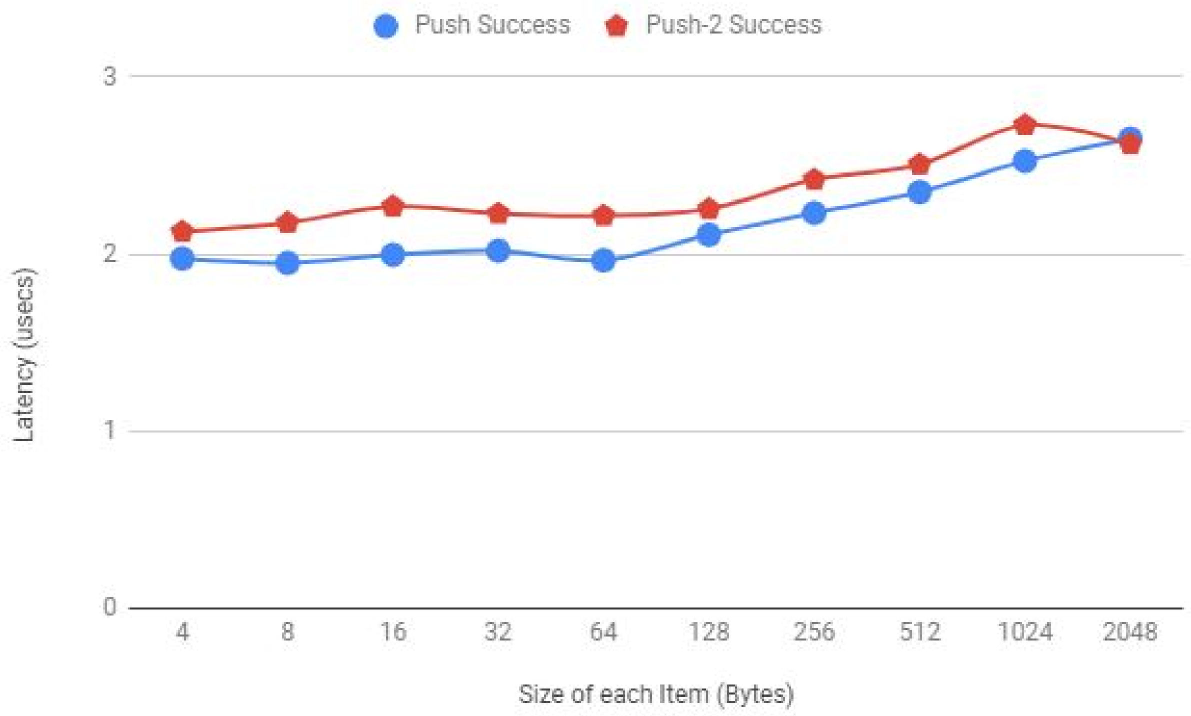


Figure 6.8: Latency for Dequeue Operation for Different Space Notification Techniques

they could be a good choice for flow structures for which high throughput is critical. However, they result in relatively expensive dequeue operations.

- Push-2 designs introduce complexity and offer no performance advantages. Hence, they should be avoided.

6.4 Comparing Flow Queues and Sockets

In this section, we compare the performance of RDMA flow queues with that of Linux stream sockets, which provide a kernel-based abstraction similar to flow queues. We consider the efficiency and throughput of both approaches, and also compare their transit times, i.e., the minimum time between enqueuing a message at the producer and dequeuing it at the consumer.

Specifically, we compare sockets to the following three RDMA queue implementations:

- **Basic Flow Queue (Basic):** This is the flow queue design that is presented in Section 5.1. The producer pushes data to the consumer, watermarking is used for transfer notifications, and space notifications are pulled by the producer. This flow queue is completely passive – all CPU work occurs during enqueue and dequeue operations.

- **Batching Flow Queue (Batching):** This queue is similar to the basic queue, except that items are pushed to the consumer in batches. The batch size is determined dynamically. Specifically, the batch queue keeps at most one RDMA write operation (for pushing data) in flight at a time. Any items that are enqueued while a data push is in flight are retained in the producer's buffer, and are pushed as part of the next batch. The push of the next batch is initiated by the first enqueue operation that occurs after the previous batch's push has completed. Like the basic queue, the batching queue is passive.

- **Active Batching Flow Queue (Data Mover):** The data mover queue is an active version of the batching queue. In the data mover queue, all RDMA operations at the producer, including posting RDMA operations and polling for completion events, are offloaded to a dedicated thread. When an RDMA operation completes, the thread updates state variables and potentially issues new RDMA operations, e.g, it will trigger a push of the next batch when the previous batch push completes. After each enqueue operation, the thread also determines whether any new RDMA operations are needed, e.g., it will push a newly enqueued item immediately if there is not already a push in flight. Enqueue operations are very fast, because all RDMA operations are offloaded to the "data mover" thread. In these design, only the producer is active. The consumer remains passive and network-oblivious.

We compare these three flow queues to a baseline implementation based on TCP stream sockets. Each enqueue operation is implemented as a socket send operation. Each dequeue operation is implemented as a socket recv. The Maximum Transmission Unit(MTU) size for the stream is set to 1024 bytes, which matches RDMA's MTU.

Our first experiment uses the same design described in Section 6.1, with a single enqueueing thread and a single dequeueing thread. Figure 6.9 shows the throughput of each queue on a log scale. The results are also tabulated in Table 6.1. Figure 6.10 shows the mean latency of enqueue operations for basic, batching and sockets while data mover is not shown as it is very fast and has values less than a microsecond.

Throughput of all of the queues is limited by the enqueueing rate of the single producer thread until the items are large enough to saturate the 10 GbE network. The throughput

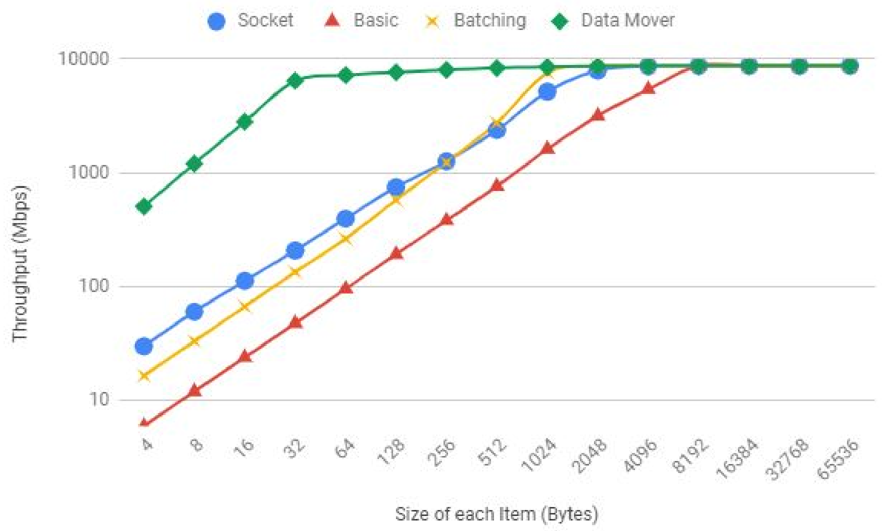


Figure 6.9: Throughput of Flow Queues and Socket Baseline

for sockets varies by 10% from the mean values whereas for the other designs it is within 2% of deviation from the mean value.

As shown in Figure 6.10, the cost of enqueue operations varies substantially across the queues. Enqueue operations are fastest for the data mover queue and has enqueue latency less than 0.5 μsecs (it requires only a few nanoseconds and is not shown in the graph), since it runs entirely in user space, and all RDMA operations are offloaded to a separate thread. Thus, each enqueue only copies the item into the producer's buffer and updates state variables. As a result, a single thread is able to saturate the network with items as small as 32 bytes. Like the data mover queue, the socket baseline can batch items and move network processing off of the critical path of the enqueue operation. However, each enqueue involves at least one kernel call, which the data mover queue avoids. This cost could be removed from the socket baseline by buffering items at the user level and sending less frequently, but this will come at the cost of extra data copying and increased transit times.

Both the batching queue and the basic queue have more expensive enqueues than the socket baseline. This is because both of these implementations obtain RDMA completion events through a Linux file descriptor, meaning that a kernel read() call is required to poll for completion events. In general, there may be multiple completion events (on average) per enqueue operation, increasing the cost of enqueue. These costs can be avoided, with some extra complexity, by polling the NIC completion queue directly, in user space. Doing

Size of Item	Socket	Basic	Batch	Data Mover
4	29	5	16	506
8	60	12	33	1202
16	112	23	66	2801
32	207	47	134	6430
64	396	95	263	7185
128	745	192	577	7632
256	1260	381	1235	7975
512	2369	760	2736	8303
1024	5153	1610	7522	8469
2048	7907	3155	8649	8616
4096	8618	5392	8677	8647
8192	8691	8701	8694	8666
16384	8699	8709	8706	8680
32768	8697	8714	8712	8684
65536	8700	8716	8715	8706

Table 6.1: Throughput in Mbps of Flow Queues and Socket Baseline (10 Gbps network)

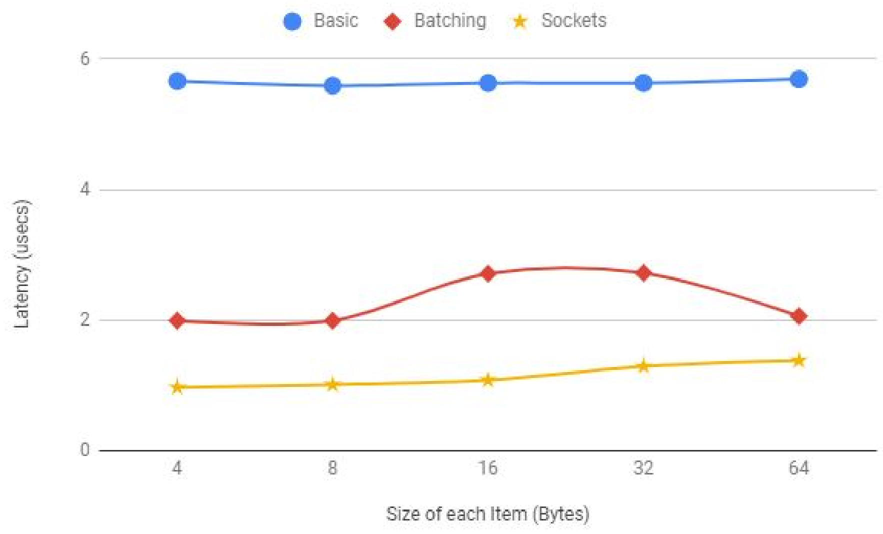


Figure 6.10: Enqueue Latency for Flow Queues and Socket Baseline

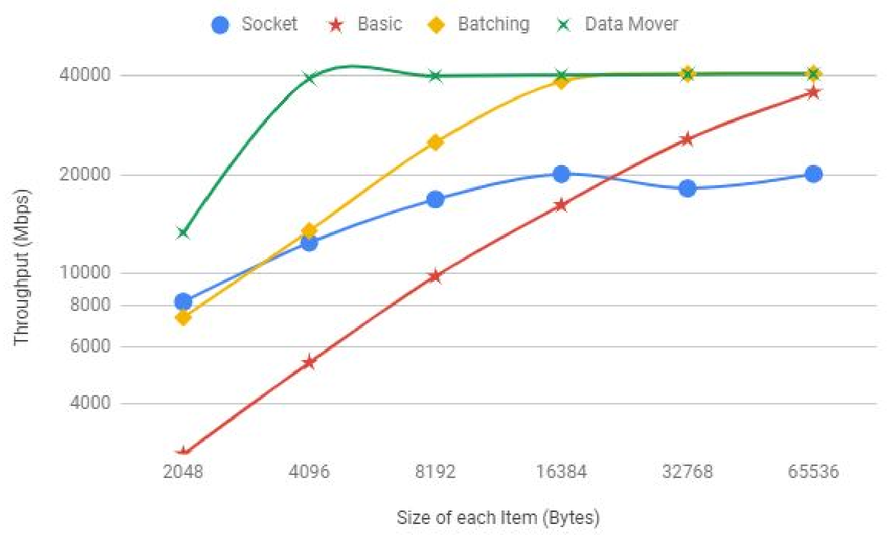


Figure 6.11: Throughput of Flow Queues and Socket Baseline (40 Gbps network)

Size of Item	Socket	Basic	Batch	Data Mover
2048	8195	2814	7346	13298
4096	12417	5346	13479	39262
8192	16819	9809	25088	39962
16384	20092	16170	38503	40257
32768	18172	25652	40590	40456
65536	20103	35677	40635	40547

Table 6.2: Throughput (in Mbps) of Flow Queues and Socket Baseline (40 Gbps network)

so should move enqueue performance closer to that of the data mover queue, which does not perform any system calls during enqueue operations.

We repeated the throughput experiment using a 40 GbE network (which can have an extended line up to 56 Gbps). Figure 6.11 shows the results on a log scale, for larger item sizes. Table 6.2 tabulates the same results. (For smaller sizes, the results are similar to those for the 10 GbE network.) Even with a single driver thread, the data mover queue can saturate the network for items sizes of 4 KB and above. Throughput for the socket baseline peaked at about 20 Gbps. The throughput for the socket baseline varies and deviated by up to 30% of the mean value.

We do not plot the latency for dequeue operations as the latency values are less than a microsecond for queue designs. The mean latency for dequeue operations for each of the queues is less than 0.5 μsecs for item sizes ranging from 4 to 2048. On the other hand, the average latency of socket recv call is 1 μsec. Here, all of the RDMA flow queues have lower latency than the socket baseline, since the latter involves a system call. All three flow queues are network-oblivious on the consumer side. Thus, they perform no system calls and no RDMA operations

6.4.1 Transfer Times

We performed a second experiment to measure the item transfer time of the RDMA queues and the socket baseline. The transfer time is the time between the enqueue of an item at the producer and the dequeue of that item at the consumer. Since this will depend on how full the queue is, we measure transfer times with an empty queue to determine the minimum transfer time.

This experiment involves two queues, Q_1 and Q_2 going in opposite direction. A single driver on one side enqueues an item into Q_1 and then attempts to dequeue from Q_2. On the other side, a single driver thread attempts to dequeue from Q_1. Once it succeeds, it immediately enqueues an item into Q_2. This handshake is repeated 131072 (2^{17}) times, and we measure the total time required at the producer. The mean ping pong time is calculated as the total time divided by the number of handshakes (2^{17}), and the transfer time is reported as half of the mean ping pong time. The size of the exchanged items is a parameter of the experiment. This experiment was performed on the 10 GbE network.

Figure 6.12 shows the results of this experiment. We show transfer times for the basic queue and the socket baseline. For comparison the graph shows the raw RDMA write time. This is half the time for a bidirectional exchange of RDMA write operations (of the size of one item) between the two sides, without the additional costs of memory management and notifications imposed by the queue. The Transfer time for the RDMA queue is about 7 μsecs for the smallest items, which is a little more than double the raw write time. In contrast, the socket baseline requires about 15 μsecs for the smallest items, which includes kernel call overheads on both sides.

The purpose of flow structures is to use the flexibility of RDMA to enable a rich variety of easy-to-use communication abstractions through which applications can utilize the performance advantages offered by RDMA. By comparing FIFO flow queues and sockets, which provide similar abstractions, we've shown that the RDMA implementation can indeed expose performance advantages to applications, without the complexities of the low-level RDMA interface. RDMA-based abstractions can offer lower-overhead enqueue

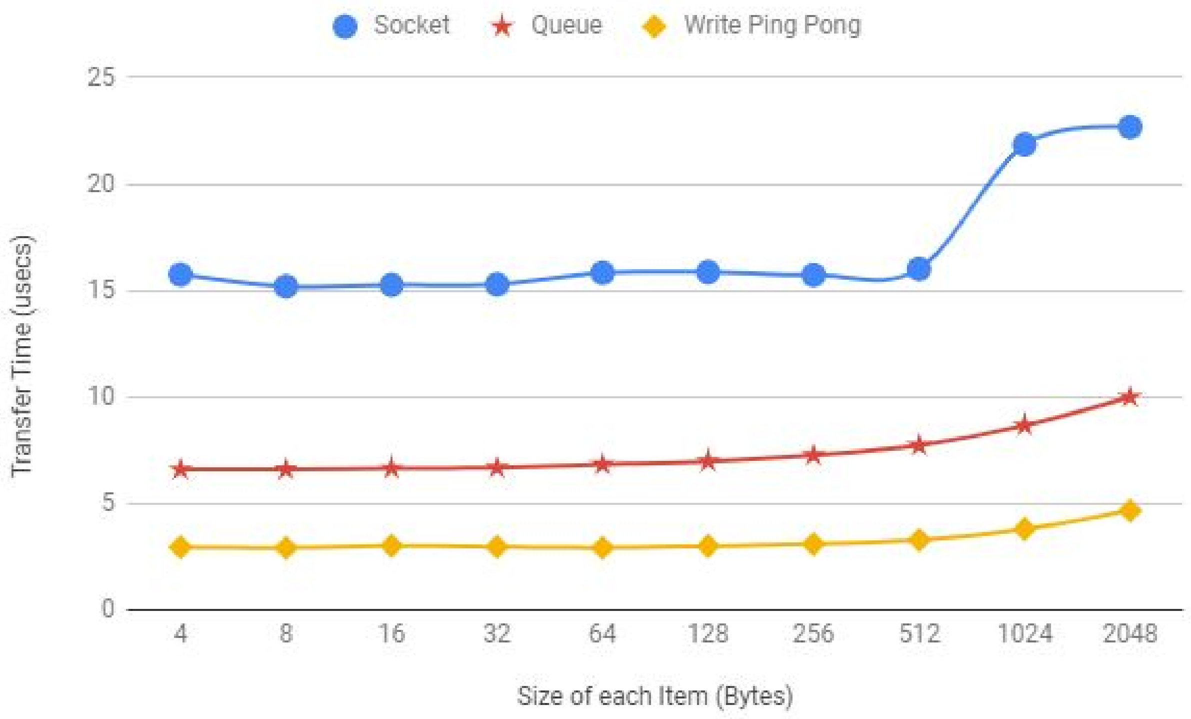


Figure 6.12: Transfer Time for RDMA Queues and Socket Baseline

and dequeue operations, greater throughput, and faster transfer times than kernel-based abstractions.

Chapter 7

Related Work

Flow structures are distributed data structures using RDMA for information exchange. We discuss related work in two areas related to flow structures. The first area, presented in Section 7.1, covers work on using and abstracting RDMA effectively. The second, presented in Section 7.2, covers work on distributed data structures.

7.1 RDMA Systems

Earlier work [17], [8] dissected the traditional networking stack and highlighted the advantages offered by RDMA. Frey et al,[17] pointed out that the performance drop when using the IP stack is because of the use of intermediate buffers and memory copies involved, while Balaji et al, [8] studied the impact of traffic on the memory bus while data is being transferred. As a result, over the last few years RDMA is seeing increasing adoption. However, a challenge for RDMA is that it is not easy to adopt. Many papers [30], [4], [35], [6] have highlighted the difficulties such as low level interface, asynchronous request handling by hardware, different transport modes, the wide design space from which to select transfer operations, and hardware limitations like queue sizes.

We classify work on making effective use of RDMA into two categories. The first category covers work on abstractions intended to simplify the use of RDMA while retaining its performance advantages. The second category covers work on how to make effective use of RDMA so that its performance potential can be realized. In the next sections we will look at each of these.

7.1.1 Abstractions For RDMA

Remote Regions [4] provides a file-based abstraction for managing remote memory. Regions provide operations for sharing and accessing remote memory using RDMA, using the file system namespace. Regions allow any process or machine to export its memory as a file to other process.

RStore [34] presents a pool of remotely accessible memory that can be allocated and shared using RDMA. It provides a distributed access *namespace* and APIs for managing the pool and attaching memory to remote servers. Further, to improve performance it separates resource allocation from the data path making it easier to use and managing resources.

Storm [30] provides a transactional API which dynamically determines the use of one sided read operations or Remote Procedure Calls (using write with immediate based RPC) operations for accessing remote data structures. Further, the transactions are based on optimistic concurrency control and can be used to build different kinds of remote structures. It considers the recent developments in networking hardware (Mellanox CX4 and CX5 NICs).

RDMAvisor [35], provides a simple API based on *send-recv* semantics that enables the use of RDMA as a service in a cloud setting. For deploying RDMA in a data center, DemiKernel [39] proposed a new operating system (OS) architecture for kernel bypass devices, like RDMA-capable NICs, that are loosely coupled with the host running the OS. The proposed DemiKernel assists applications by providing a queue-based abstraction for network I/O operations that are performed solely in user space.

For streaming services using RDMA, RocketStreams [12] presents a framework for streaming content from one machine to many others. It provides an easy to use design with an event driven user space library that makes use of write with immediate operation for data transfers. This framework can be used for building custom applications as well.

The Data Processing Interface (DPI) [6] proposes flow-based abstraction for many-to-many data flows between machines. It also proposes a low-level interface for simplifying some aspects of the RDMA interface, such as remote memory allocation. Further, it explores the possibility of incorporating in-network processing in the flow abstraction.

Among the mentioned systems most of them [34, 35, 30, 4] address memories as references and require indirect communication to access remote memory while flow structures refer data as items and have direct communication between a producer and consumer machine. As a result, flow structures can be easily substituted for or by other communication primitives whereas using these abstractions requires application specific implementations.

7.1.2 Using RDMA Effectively

FASST, [20] provides a distributed in-memory transaction processing system. It makes extensive use of unreliable connections for scaling the number of connections as a workaround for limited RNIC memory. FASST, uses RPC-style communication with two-sided operations instead of one sided operations.

Another transaction engine, GAM [11], provides a distributed memory management platform with flexible consistency for memory access. It uses synchronous reads and asynchronous writes with pipelined request handing for improved performance and latency hiding. Further, GAM uses a distributed cache to try to avoid remote accesses.

The Remote Fetch Paradigm (RFP) [33] considers the disparity between the inbound and outbound endpoints in one-sided RDMA operations. The latter requires CPU involvement, while the former does not. In RFP, clients push requests to the server and pull their results, so that the server is always the inbound endpoint. This distinction is similar to the network-awareness property described in this thesis.

Other work [38, 37, 16, 32, 9] considers the effective use of RDMA in the context of database systems. Yoon et al. [38] presents a decentralized lock manager using RDMA atomic primitives with retries and backoff mechanism in the case of lock failures. Barthels et al. [9] use RDMA for join processing. They make use of pre-allocated pinned buffers for partitioning and distributing the data between servers, each of which is computing a part of the join. Frey et al. [16] suggests a new architecture for transferring messages between servers and evaluate their approach for different distributed join processing algorithms. Flow Join [32] proposes an adaptive technique to identify heavily hit hash keys at runtime, and uses RDMA for data distribution and communication to minimize latency and avoid host-side CPU usage.

For implementation purposes each of the above systems considers a set of RDMA operations and follows an RPC or RFP approach whereas flow structures are flexible and can be easily designed with a combination of different flows so as to meet the changing requirements of application behaviour.

7.2 Distributed Data Structures

With the advent of RDMA's one-sided operations, many researchers have explored the idea of using them to implement high-performance distributed data structures [23, 5, 18, 25].

Larkins et al. [23] present distributed linked data structures based on a *get/compute/put* model in a global address space. Their proposed tree structures use directory structures to hold metadata. They divide the space into chunks and use one-sided operations through the Message Passing Interface(MPI) to access data. They support tree node data access by means of chunks and support multi chunk transfers for efficient transfer of data.

DASH [18] is a C++ library that implements a distributed array, following the C++ STL style with different data distribution methods. It supports item access via references and uses one sided operations underneath.

Aguilera et al. [5], proposed "far memory" which is quite near to our work and relies on one sided operations for developing data structure based abstractions for accessing remote memory. They emphasize the need for hardware primitives to realize indirect addressing of remote memory and propose certain primitives to achieve the same. Further, they classify memory as far (remote) and local (present on same machine) and provide a runtime cache for holding items or information that can be repeatedly used with relaxed consistency.

RDMA-based key value(KV) stores have been quite extensively explored [27, 20, 21, 13, 14, 36]. Some of these [20, 21, 27] use two-sided operations to implement the *get* and *put* operations. Pilaf [27] provides a distributed key value store using one sided operations for gets while updates and puts are sent via messages using two sided operations to the server so as to avoid race conditions. In Pilaf, each item is provided with a checksum to self verify the contents after a fetch is performed. In HERD [20, 21], clients use one-sided write operations to write requests into server memory. The server returns responses using User Datagram(UD) based two sided operations. Further, the server employs key prefetching with pipelined requests to overlap key fetch and data processing to enhance performance.

Others [13, 14, 36] use only one sided operations with network-oblivious servers. Nessie [13] is a client-driven design using only one-sided operations for doing puts and gets. FARM [14] uses one sided reads for gets, while puts are handled using one-sided writes performed in order by means of a circular buffer on server side. Also, DrTM [36] uses one sided operations for *get and put* operations of a KV Store. Their design is based on a transactions model aided by transactional memory.

The mentioned distributed structures [23, 18] are symmetric as each of the clients can perform any kind of operation on the data whereas flow structures are orthogonal with an asymmetric model. With flow structures, data flows from producers to consumers and is aided by other flows to assist in managing the distributed structures design. These flows are based upon RDMA operations wherein we do not exclude one type of operation in favour of another based upon any performance metric. Here, we intend to generalize the design possibilities considering research advancements be it hardware [28, 29, 24, 30] or application [37] based. These work [28, 29, 24, 30] highlight and indicate that functionality of RDMA will evolve. For instance, Mittal et al. [29] and Timely [28] propose support for congestion control and ack support for unreliable connections for sound delivery of packets while Rogue [24] and Storm [30] emphasize the fact that the hardware is evolving with time. Based upon these observations we believe that hardware support is going to get enhanced with the same set of primitive operations that RDMA offers. Hence, we do not rule out certain designs as they may perform well in future. Rather, the notion is to be open to all possible design alternatives while simplifying the complexity in building distributed structures which we illustrate by means of a flow queue.

7.2.1 Distributed Queues

Phothilimthana et al, [31] propose Floem, a language and compiler for NIC-accelerated applications. They motivates the use of distributed data structures, like flow structures, that can effectively provide a mapping of operations to physical queues.

Liu et al. [26] analyze the use of RDMA for implementing MPI. They propose a hybrid design that uses one sided operations for data transfers and two sided operations for metadata transfer. It handles overflow cases by using two-sided operations and reserve buffers. Further, based upon the observations of Balaji et al. [7] for an asynchronous zero copy design for message transfers, Liu et al. [25] extended their earlier hybrid design to support large messages with a zero copy variant for the MPICH2 library. The flow queue designs presented in Chapter 5 of this thesis are similar to these MPI designs, although they use lazy refresh of counter metadata. These lazy operations are done on a demand basis and are mostly non-blocking and interleaved with actual data flow operations.

Quite recently, Fent et al. [15] propose L5, a unified interface for communication covering RDMA, RoCE and shared memory. They use a similar three-counter design similar to the one used for flow queues in this thesis, although the L5 design is for shared memory message transfer. For remote communication, L5 considers asymmetric associations between server and clients. For client to the server direction, they consider N-1 settings and use chained requests of two writes, where first one writes the message and the second toggles a single-byte mailbox which is polled by the server for incoming messages. On the other hand, for the server-to-client direction (1-1) they use a single write operation with a length field and watermark.

Chapter 8

Conclusions and Future Work

Flow structures are asymmetric distributed data structures with working memory distributed across machines. They use RDMA for direct information transfer between machines. Flow structures provide rich communication abstractions that ideally provide excellent performance while hiding the complexity of the low-level RDMA interface.

We defined a design space for flow structures based on three information flows that any flow structure must implement. Each point in the design space will have different performance and implementation characteristics. Thus, implementations can target the requirements of specific applications. We illustrate the design space using two implementations of a 1-1 FIFO flow queue, corresponding to different points in the design space. In one implementation, both the producer and consumer are network-aware. In the other, the producer is network-aware but the consumer is not.

We also present a series of experiments with a variety of different FIFO flow queue implementations to illustrate the performance implications of different choices in the design space. The results provide some insights into these choices, such as the benefits of watermarking over other methods of transfer notification. We also compare RDMA flow queues with a socket baseline, showing that RDMA flow queues can provide better throughput, lower enqueue/dequeue latency and shorter transfer times behind a high-level interface. Thus, flow structures can provide easy-to-use communication abstractions while offering performance advantages over non-RDMA alternatives.

Flow structures have a large potential for future work. Many other types of flow structures, such as priority queues and N-to-1 queues, are possible, and the performance achievable for such structures remains to be determined. Also, we can develop flow structures based upon other networking standards, such as DPDK or MPI. Finally, flow structure abstractions can be adapted to incorporate the offloading of computation into the network. For instance, flow queues could be adapted to apply application-specific filters or transformations to enqueued data items.

References

[1] *At The Network's Edge.* https://www.dell.com/community/s/vjauj58549/attachments/vjauj58549/PowerEdge-Rack-HW/4918/1/At%20The%20Network's%20Edge%20-%20Hendrich%20Hernandez.pdf.

[2] *MLNX_EN for Linux User Manual.* https://www.mellanox.com/related-docs/prod_software/Mellanox_EN_for_Linux_User_Manual_v4_0.pdf.

[3] *RDMA Aware Networks Programming User Manual.* http://www.mellanox.com/related-docs/prod_software/RDMA_Aware_Programming_user_manual.pdf.

[4] Marcos K Aguilera, Nadav Amit, Irina Calciu, Xavier Deguillard, Jayneel Gandhi, Stanko Novakovic, Arun Ramanathan, Pratap Subrahmanyam, Lalith Suresh, Kiran Tati, et al. Remote regions: a simple abstraction for remote memory. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 775–787, 2018.

[5] Marcos K Aguilera, Kimberly Keeton, Stanko Novakovic, and Sharad Singhal. Designing Far Memory Data Structures: Think Outside the Box. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, pages 120–126. ACM, 2019.

[6] Gustavo Alonso, Carsten Binnig, Ippokratis Pandis, Kenneth Salem, Jan Skrzypczak, Ryan Stutsman, Lasse Thostrup, Tianzheng Wang, Zeke Wang, and Tobias Ziegler. DPI: The Data Processing Interface for Modern Networks. In *CIDR*, 2019.

[7] Pavan Balaji, Sitha Bhagvat, H-W Jin, and Dhabaleswar K Panda. Asynchronous zero-copy communication for synchronous sockets in the sockets direct protocol (SDP) over InfiniBand. In *Proceedings 20th IEEE International Parallel & Distributed Processing Symposium*, pages 8–pp. IEEE, 2006.

[8] Pavan Balaji, Hemal V Shah, and Dhabaleswar K Panda. Sockets vs rdma interface over 10-gigabit networks: An in-depth analysis of the memory traffic bottleneck. In *In RAIT workshop*, volume 4, page 2004, 2004.

[9] Claude Barthels, Simon Loesing, Gustavo Alonso, and Donald Kossmann. Rack-scale in-memory join processing using RDMA. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1463–1475, 2015.

[10] Benjamin Brock, Yuxin Chen, Jiakun Yan, John Owens, Aydın Buluç, and Katherine Yelick. RDMA vs. RPC for Implementing Distributed Data Structures. *arXiv preprint arXiv:1910.02158*, 2019.

[11] Qingchao Cai, Wentian Guo, Hao Zhang, Divyakant Agrawal, Gang Chen, Beng Chin Ooi, Kian-Lee Tan, Yong Meng Teo, and Sheng Wang. Efficient distributed memory management with RDMA and caching. *Proceedings of the VLDB Endowment*, 11(11):1604–1617, 2018.

[12] Benjamin Cassell, Huy Hoang, and Tim Brecht. RocketStreams: A Framework for the Efficient Dissemination of Live Streaming Video. In *Proceedings of the 10th ACM SIGOPS Asia-Pacific Workshop on Systems*, pages 84–90. ACM, 2019.

[13] Benjamin Cassell, Tyler Szepesi, Bernard Wong, Tim Brecht, Jonathan Ma, and Xiaoyi Liu. Nessie: A decoupled, client-driven key-value store using RDMA. *IEEE Transactions on Parallel and Distributed Systems*, 28(12):3537–3552, 2017.

[14] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. FaRM: Fast remote memory. In *11th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 14)*, pages 401–414, 2014.

[15] Philipp Fent, Alexander van Renen, Andreas Kipf, Viktor Leis, Thomas Neumann, and Alfons Kemper. Low-Latency Communication for Fast DBMS Using RDMA and Shared Memory. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*, pages 1477–1488. IEEE, 2020.

[16] Philip W Frey, Romulo Goncalves, Martin Kersten, and Jens Teubner. A spinning join that does not get dizzy. In *2010 IEEE 30th International Conference on Distributed Computing Systems*, pages 283–292. IEEE, 2010.

[17] Philip Werner Frey and Gustavo Alonso. Minimizing the hidden cost of RDMA. In *2009 29th IEEE International Conference on Distributed Computing Systems*, pages 553–560. IEEE, 2009.

[18] Karl Fürlinger, Tobias Fuchs, and Roger Kowalewski. DASH: A C++ PGAS library for distributed data structures and parallel algorithms. In *2016 IEEE 18th International Conference on High Performance Computing and Communications; IEEE 14th*

International Conference on Smart City; IEEE 2nd International Conference on Data Science and Systems (HPCC/SmartCity/DSS), pages 983–990. Ieee, 2016.

[19] Anuj Kalia, Michael Kaminsky, and David Andersen. Datacenter RPCs can be general and fast. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 1–16, 2019.

[20] Anuj Kalia, Michael Kaminsky, and David G Andersen. Using RDMA efficiently for key-value services. In *ACM SIGCOMM Computer Communication Review*, volume 44, pages 295–306. ACM, 2014.

[21] 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, 2016.

[22] Anuj Kalia, Michael Kaminsky, and David G Andersen. FaSST: Fast, Scalable and Simple Distributed Transactions with Two-Sided (RDMA) Datagram RPCs. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 185–201, 2016.

[23] D Brian Larkins, James Dinan, Sriram Krishnamoorthy, Srinivasan Parthasarathy, Atanas Rountev, and Ponnuswamy Sadayappan. Global trees: a framework for linked data structures on distributed memory parallel systems. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, page 57. IEEE Press, 2008.

[24] Yanfang Le, Brent Stephens, Arjun Singhvi, Aditya Akella, and Michael M Swift. RoGUE: RDMA over Generic Unconverged Ethernet. In *SoCC*, pages 225–236, 2018.

[25] Jiuxing Liu, Weihang Jiang, Pete Wyckoff, Dhabaleswar K Panda, David Ashton, Darius Buntinas, William Gropp, and Brian Toonen. Design and Implementation of MPICH2 over InfiniBand with RDMA Support. In *18th International Parallel and Distributed Processing Symposium, 2004. Proceedings.*, page 16. IEEE, 2004.

[26] Jiuxing Liu, Jiesheng Wu, and Dhabaleswar K Panda. High performance RDMA-based MPI implementation over InfiniBand. *International Journal of Parallel Programming*, 32(3):167–198, 2004.

[27] Christopher Mitchell, Yifeng Geng, and Jinyang Li. Using One-Sided RDMA Reads to Build a Fast, CPU-Efficient Key-Value Store. In *Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 103–114, 2013.

[28] Radhika Mittal, Nandita Dukkipati, Emily Blem, Hassan Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, David Zats, et al. TIMELY: RTT-based Congestion Control for the Datacenter. In *ACM SIGCOMM Computer Communication Review*, volume 45, pages 537–550. ACM, 2015.

[29] Radhika Mittal, Alexander Shpiner, Aurojit Panda, Eitan Zahavi, Arvind Krishnamurthy, Sylvia Ratnasamy, and Scott Shenker. Revisiting network support for rdma. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 313–326. ACM, 2018.

[30] Stanko Novakovic, Yizhou Shan, Aasheesh Kolli, Michael Cui, Yiying Zhang, Haggai Eran, Boris Pismenny, Liran Liss, Michael Wei, Dan Tsafrir, et al. Storm: a fast transactional dataplane for remote data structures. In *Proceedings of the 12th ACM International Conference on Systems and Storage*, pages 97–108. ACM, 2019.

[31] Phitchaya Mangpo Phothilimthana, Ming Liu, Antoine Kaufmann, Simon Peter, Rastislav Bodik, and Thomas Anderson. Floem: a programming system for NIC-accelerated network applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 663–679, 2018.

[32] Wolf Rödiger, Sam Idicula, Alfons Kemper, and Thomas Neumann. Flow-join: Adaptive skew handling for distributed joins over high-speed networks. In *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*, pages 1194–1205. IEEE, 2016.

[33] Maomeng Su, Mingxing Zhang, Kang Chen, Zhenyu Guo, and Yongwei Wu. Rfp: When rpc is faster than server-bypass with rdma. In *Proceedings of the Twelfth European Conference on Computer Systems*, pages 1–15. ACM, 2017.

[34] Animesh Trivedi, Patrick Stuedi, Bernard Metzler, Clemens Lutz, Martin Schmatz, and Thomas R Gross. Rstore: A direct-access DRAM-based data store. In *2015 IEEE 35th International Conference on Distributed Computing Systems*, pages 674–685. IEEE, 2015.

[35] Zhi Wang, Xiaoliang Wang, Zhuzhong Qian, Baoliu Ye, and Sanglu Lu. RDMAvisor: Toward Deploying Scalable and Simple RDMA as a Service in Datacenters. *arXiv preprint arXiv:1802.01870*, 2018.

[36] Xingda Wei, Jiaxin Shi, Yanzhe Chen, Rong Chen, and Haibo Chen. Fast in-memory transaction processing using RDMA and HTM. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 87–104, 2015.

[37] Bairen Yi, Jiacheng Xia, Li Chen, and Kai Chen. Towards zero copy dataflows using rdma. In *Proceedings of the SIGCOMM Posters and Demos*, pages 28–30. ACM, 2017.

[38] Dong Young Yoon, Mosharaf Chowdhury, and Barzan Mozafari. Distributed lock management with rdma: Decentralization without starvation. In *Proceedings of the 2018 International Conference on Management of Data*, pages 1571–1586. ACM, 2018.

[39] Irene Zhang, Jing Liu, Amanda Austin, Michael Lowell Roberts, and Anirudh Badam. I'm Not Dead Yet!: The Role of the Operating System in a Kernel-Bypass Era. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, pages 73–80. ACM, 2019.