DFI: The Data Flow Interface for High-Speed Networks

Lasse Thostrup, TU Darmstadt
Jan Skrzypczak, Zuse Institue, Berlin
Matthias Jasny, TU Darmstadt
Tobias Ziegler, TU Darmstadt
Carsten Binnig, TU Darmstadt

Abstract

In this paper, we propose the Data Flow Interface (DFI) as a way to make it easier for data processing systems to exploit high-speed networks without the need to deal with the complexity of RDMA. By lifting the level of abstraction, DFI factors out much of the complexity of network communication and makes it easier for developers to declaratively express how data should be efficiently routed to accomplish a given distributed data processing task. As we show in our experiments, DFI is able to support a wide variety of data-centric applications with high performance at a low complexity for the applications.

ACM Reference Format:
Lasse Thostrup, Jan Skrzypczak, Matthias Jasny, Tobias Ziegler, and Carsten Binnig. 2021. DFI: The Data Flow Interface for High-Speed Networks. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD '21), June 20–25, 2021, Virtual Event, China.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3448016.3452816

1 Introduction

Motivation: Scale-out data processing systems are the typical architecture used today by many systems to process large data volumes since they allow applications to increase compute and memory capacities by simply adding further processing nodes. However, a typical bottleneck in scale-out systems is the network which often slows down the speed of data processing if communication is in the critical path. For distributed in-memory systems this might lead to degraded performance when adding more nodes [29].

However, this changed with the advent of high-speed networks such as InfiniBand. Network bandwidth increased almost up to the speed of main memory and latencies dropped by orders of magnitude [4], making scale-out solutions more competitive. However, blindly upgrading to faster networks does often not directly translate into performance gains, as there is a plenitude of aspects to consider to achieve a good performance for distributed data processing systems.

One particular important aspect to efficiently use high-speed networks is to redesign data processing systems to leverage remote direct memory access (RDMA) as a low overhead communication protocol. RDMA provides kernel bypass and zero-copy making data transfers less expensive than classical network stacks such as TCP/IP [12]. In recent years, industry and academia have thus started to adapt scale-out data processing systems in order to make use of RDMA. As a result, significant speed-ups have been shown for a wide range of data processing systems ranging from key-value stores [17, 21, 26], over distributed DBMSs (for OLTP and OLAP) [4, 18, 34, 37–39] to Big Data systems and Distributed Machine Learning [15, 25, 36].

However, using RDMA is complicated because it provides only low-level abstractions (called RDMA verbs) for data processing [8]. Hence, redesigning data processing systems for RDMA often requires significant efforts to take care of many low-level detail choices [3, 4, 16, 18, 41] regarding remote memory and connection management as well as other decisions such as which RDMA verbs to use for which type of workload.

Contribution: In this paper, we propose the Data Flow Interface (DFI) as a way to make it easier for data processing systems to exploit high-speed networks. Accordingly, DFI defines abstractions and interfaces suited to a broad class of data-intensive applications, yet simple enough for practical implementation with predictable performance and low overhead relative to "hand-tuned", ad hoc alternatives. In designing a high-level interface tailored to data processing, we adopt the approach taken by the high-performance community for MPI [14] to provide a simple yet effective interface for high-speed networks. However, since MPI has been designed for computation-intensive workloads such as large-scale simulations, it comes with many design choices that are not optimal for data-intensive workloads [19]. Consequently, MPI has seen only very limited adoption for data processing systems [2].

In brief, the main idea of DFI is that data movements are represented as *flows*. DFI flows are an abstraction providing primitives for efficient network communication. These primitives are intended to be used as a foundation for building data-intensive systems and provide many benefits over MPI (e.g., thread-centricity, pipelined communication). By lifting the level of abstraction, DFI flows not only hide much of the low-level complexity of network communication but also allow developers to declaratively express how data should be efficiently routed to accomplish a given distributed data processing task. Moreover, DFI flows allow developers to specify *optimization hints*; e.g., to maximize bandwidth-utilization or minimize network latency of transfers. By using flows as the main abstraction, DFI supports a wide variety of data-centric applications ranging from bandwidth-sensitive distributed OLAP to more latency-sensitive workloads such as distributed OLTP or replication with consensus protocols.

Recently, the need for better interfaces to high-speed networks has also been discussed in a vision paper [1]. We, however, are the first paper that provide a concrete suggestion and a full implementation for an interface that can enable a broad class of data-centric applications to make efficient use of modern networks. Moreover, there have also been several other attempts to build libraries for data processing over high-speed networks [5, 9, 11]. For example, FaRM [9] and GAM [5] provide a programming model based on a shared address space which focuses on supporting latency-sensitive

SIGMOD '21, June 20–25, 2021, Virtual Event, China.

ACM ISBN 978-1-4503-8343-1/21/06…$15.00
https://doi.org/10.1145/3448016.3452816

workloads (e.g., such as distributed transactions). Another example is L5 [11], which target the communication between clients and servers to replace traditional client-centric communication libraries such as ODBC. Different from those libraries, as mentioned before, DFI flows aim to be a much more general abstraction that can support a broader class of data-centric applications.

Finally, like MPI and different from the approaches mentioned before (such as FaRM, GAM and L5), DFI only defines an interface for communication. Hence, different vendors can provide an efficient implementation of DFI for their network technology. As a result, this increases the compatibility and portability of data processing systems across different networking technologies including RDMA-capable networks (such as InfiniBand as well as RoCE).

In summary, this paper makes the following contributions:

- First, we present the design of DFI based on the general abstraction of flows that allow developers to declaratively specify the communication behavior of distributed systems by defining its topology (1:1, N:1, 1:N and N:M) as well as providing other properties for execution.
- Second, we provide a first implementation of DFI[1] for an InfiniBand-based networking stack and discuss how the high-level abstractions of DFI are being mapped to the low-level implementations using RDMA.
- Third, we provide an exhaustive evaluation of our DFI implementation and demonstrate that DFI does not only provide many benefits over MPI for data processing but also showcase that DFI can provide high performance for different data-centric applications.

Outline: The remainder of this paper is structured as follows: In Section 2, we first give an overview of two existing interfaces, RDMA verbs and MPI. Moreover, we analyze MPI as the direction taken by the high-performance community to provide a simple yet effective interface for high-speed networks and discuss the limitations of MPI for distributed data processing. Afterwards, in Section 3 we present an overview of DFI before we discuss details of the programming model in Section 4 as well as our implementation for InfiniBand in Section 5. Finally, we conclude with our evaluation in Section 6 and a summary in Section 7.

2 Existing Interfaces

In this section we aim to give an overview of existing interfaces namely the standard RDMA verbs interface native to the InfiniBand network stack and the Message Passing Interface (MPI), the de facto standard in the HPC community.

2.1 RDMA Verbs

The InfiniBand RDMA verb interface is a low-level interface providing low latency and high bandwidth communication. The interface exposes one-sided verbs (*write*, *read* & *atomics*) and two-sided verbs (*send* & *receive*) which refer to the involvement of end-points (i.e., one-sided verbs only involves the CPU of the sender). The high performance of RDMA is in general achieved by the asynchronous nature of RDMA, making it possible to pipeline computation and communication such that the CPU is not busy idling during network communication. To issue RDMA verbs (one- or two-sided), the application has to register a memory region in which the RNIC can directly access memory, leaving communication related memory-management a responsibility of the application. Moreover, due to the RDMA verb interface's very low abstraction level it provides also a huge design space. This requires, however, that applications need to carefully explore this design space and to optimally make use the available low-level options [11, 17, 40, 41].

2.2 Message Passing Interface

The Message Passing Interface (MPI) is widely used by the HPC community as a high-level abstraction for high-speed networks, and has through many years of development reached a mature and industrial-strength quality. One could now argue that MPI is already good enough for data-centric applications as well. In the following, we first aim to provide a better understanding of the programming and execution model of MPI and next we discuss the shortcomings of MPI for data-centric applications.

Programming Model: For programming distributed applications, MPI provides different primitives. These primitives can be categorized into point-to-point and collective communication.

- *Point-to-point communication:* The *MPI point-to-point* primitives provide communication operations to exchange data between a sender and a receiver. Point-to-point primitives, similar to native RDMA verbs, support two-sided communication with *send* and *receive*, as well as one-sided primitives with *put* and *get*. For the two-sided primitives, the *send* request has to be matched with a *receive* request, whereas the one-sided primitives transfer data without involving the remote side into the communication. While the point-to-point primitives offer high flexibly in how data is exchanged between two nodes, they still leave many low-level details to the application and thus only raise the level of abstraction minimally compared to native RDMA verbs (RDMA read/write or RDMA send/receive). For example, the remote memory management for the one-sided MPI primitives (*put* and *get*) is still up to the application to handle, which in a setup with many writers, involves considerable amounts of engineering efforts to coordinate the concurrent memory accesses as shown in [2].
- *Collective communication:* Different from the point-to-point primitives, the so-called *MPI collectives* targets many-to-many communication between multiple (sender and receiver) nodes and provide a higher-level abstraction to exchange data between nodes. Examples of MPI collectives are *scatter*, *gather*, *broadcast* or *reduce* and *all to all* that transfer bulks of data (i.e., vectors of elements) between multiple nodes. Hence, collectives seem to be a perfect candidate for many data processing tasks such as data shuffling or even to implement replication protocols. However, while the collectives provide a convenient way to exchange data between multiple nodes, all these primitives use a bulk synchronous (i.e., blocking) communication model where all data needs to be available on the sender side before the collective is being executed. This limits the efficiency of MPI collectives for data processing [19] since it hinders overlapping of compute and communication.

[1] https://github.com/DataManagementLab/DFI-public

Execution Model: MPI programs follow a process-centric execution model, where parallelism is achieved by running the multiple processes of the same program in parallel on multiple nodes. A new MPI program is started by running *mpirun* which launches the same program on all specified processes spread across the specified cluster nodes. Again, this process-centric parallelization model is not ideal for data processing systems as we discuss next.

2.3 Shortcomings of MPI

In the following, we give a brief overview of the main limitations of MPI for distributed data processing systems. Many of these shortcomings are evaluated further in Section 6.2.

Compute- and not Data-centric: MPI was designed towards supporting compute-intensive applications such as distributed simulations. However, the communication behavior in these types of distributed applications is very different from the needs of data-intensive applications. While distributed simulations exchange data in a bulk synchronous manner (i.e., after every iteration of a simulation), many data-centric applications are often dominated by data transfers (i.e., data shuffling).

Hence, for many data-centric applications it is important that applications can overlap computation and communication efficiently such that compute resources do not get idle [3]. This also holds for more latency sensitive operations such as distributed transactions. As shown in [33], overlapping does not only help to increase the overall throughput but it also reduces the end-to-end latency of distributed transactions.

Therefore, various types of data-centric applications would benefit from a pipelined (i.e., overlapping) communication model that provides a more loose coupling between senders and receivers. While a pipelined communication model is available for MPI's point-to-point primitives such as non-blocking send or one-sided put and get primitives, using these primitives in a non-blocking manner often results in more complex application code similar to using RDMA verbs directly [2]. In addition to point-to-point primitives, MPI collectives provide a higher-level of abstraction for communication between multiple nodes. However, as mentioned before, MPI collectives use a bulk-synchronous communication model.Extensions of MPI collectives to support pipelining [35] have unfortunately not made their way into today's MPI distributions. Hence collectives as they are available today do not only lack the the support for overlapping of computation and communication but also are thus sensitive to stragglers and skew which can both limit the performance in data processing systems significantly [6, 31].

Process-centric and not Thread-centric: As mentioned before, MPI has been designed for process-level parallelism. As such, the communication primitives of MPI (point-to-point and collectives) were designed for single-threaded usage; i.e., only one dedicated communication thread of an MPI process can call the communicating primitives. This, however, is very different from designs of modern data-centric systems for high-speed networks where multiple worker threads are often required to saturate the network [4, 38].

While recent papers [2] have shown that multi-process parallelism can be used in MPI to saturate the network, it comes with other downsides.For example, when using multi-process parallelism within a node, global data structures (e.g., an aggregation hash table) need to be accessed by different processes through shared memory.

Finally, in the recent years, many MPI distributions have added multi-threading support. However, as multi-threading support was only added as an afterthought, it lacks an efficient implementation in MPI as we show in our evaluation.

3 DFI Overview

In this section, we first highlight the central design goals of DFI before we discuss the flow-based programming model, as well as the high-level idea of the execution model behind flows.

3.1 Key Design Principles

The aim of DFI is to provide a high-level abstraction that provides efficient support for a broad set of data processing systems. In the following, we present the key design principles of DFI to ideally support the needs of these systems:

(1) Pipelining: Different from MPI, which targets compute-centric applications such as distributed simulations, many data-centric applications are often dominated by data transfers (i.e., data shuffling). For this reason, it is shown to be crucial that computation and communication can be overlapped [3].

(2) Thread-centricity: Multi-threading is essential not only in achieving high degrees of parallelism in modern data-centric architectures but also to saturate the network as mentioned before. Hence, different from MPI, DFI should be designed from ground up to enable a thread-centric execution and communication model.

(3) Low-overhead synchronization: Another important aspect that goes along with thread-centricity is that DFI aims to provide low-overhead synchronization between sender and receiver threads as well as between sender threads that target the same receiver. By providing low-overhead synchronization, DFI thus should enable scalability to a high number of sender and receiver threads.

(4) Declarative optimization: A last important goal is that DFI exposes parameters as a handle for applications to declare what optimizations are desired. Examples of such optimizations are whether applications are bandwidth or latency sensitive, but also other guarantees such as global ordering of messages when data is send across flows (which is important, for example, for data replication protocols).

3.2 Flow-based Programming Model

At the center of the abstraction are DFI's flows. Flows encapsulate the movement of data between end-points in a distributed application, by exposing *sources* and *targets* as data entry and exit points on a per thread-level. This simple abstraction allows applications to compose potentially complex communication topologies, including both point-to-point, one-to-many, many-to-one and many-to-many communications between worker threads of multiple nodes. As we show later in this section, the flow abstraction is powerful enough to support a wide range of data processing use-cases such as distributed join algorithms, but also consensus protocols.

In the following, we provide an example of a concrete many-to-many flow type in DFI, which is one out of multiple other flow types as we discuss later. The most common many-to-many communication in data processing systems is arguably key-based shuffling

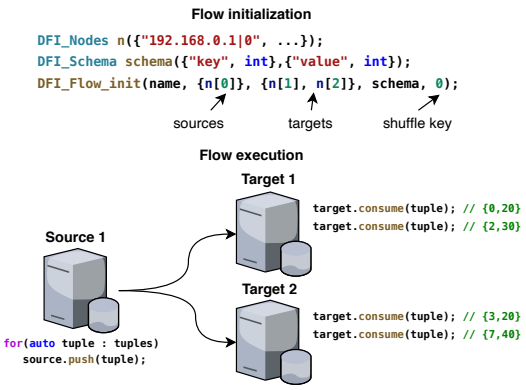


Figure 1: DFI's Programming and Execution Model. Example of flow initialization for setting up a shuffle based flow. The flow execution exemplifies the tuple-based push and consume primitives on DFI.

of data across multiple sources and targets. An example of such a shuffle flow in DFI is illustrated in Figure 1.

As we see in the example, before a flow can be used it first has to be initialized by specifying a unique flow name identifier, location of source and target threads identified with the node address and a thread ID in DFI_Nodes, the schema of tuples that are transferred and on which key the tuples should be shuffled (see upper part of Figure 1). Note, it is also possible for applications to specify application-specific partition functions, but as default a simple key-based hash function is used to partition the tuples across receivers.

To make the flow available for other nodes, its metadata is published in a central registry upon initialization (e.g., a master node in a distributed system). For using a flow, sources and targets first need to retrieve the flow metadata from the central registry. The source nodes can then use a flow by pushing tuples into the flow and the target to consume tuples out of the flow by pulling from the flow (see lower part of Figure 1).

In addition to shuffle flows between N senders and M receivers, DFI provides many other flow types (i.e., a combiner and a replicate flow) and topologies (i.e., 1:1, N:1, 1:N and N:M) to support various data processing applications. More details about the full programming model of DFI will be explained in Section 4.

3.3 High-level Flow Execution

Key to the execution model of DFI's flows are the design principles discussed above. We achieve these design principles by implementing an execution model where each thread with a source or target has a private send/receive buffer that not only decouples sender from receiver threads but also uses a new memory layout for remote data transfer between sender/receiver threads with only minimal synchronization overhead as we discuss next.

In the following, we present the high level execution of flows by following the example of shuffling tuples shown in Figure 1. The push primitive on sources is asynchronous and returns immediately after the tuple to be transferred is copied into the internal send buffer. This non-blocking behavior allows applications to interleave the computation and communication, i.e., pipeline, and thus utilize both CPU and network resources. Moreover, internally the flow execution heavily uses the available one-sided RDMA primitives to reduce the CPU involvement of the targets, and thus decouples the sources and targets as much as possible. To enable one-sided network communication, as mentioned before, a receive buffer must be in place in which the tuples of one or multiple sending threads are written to. Details about the buffer design and their low-overhead synchronization model are discussed further in Section 5.

Once a tuple has been pushed into the flow, a routing decision will be made by the flow based on the provided shuffling key. Depending on the chosen optimization goal (bandwidth or latency), the execution of the flow will transport tuples across the network. For bandwidth optimization, flows batch tuples together destined for the same target in order to achieve a better bandwidth utilization through larger messages. On the other hand if a latency optimization is chosen, the flow execution will prioritize transferring the tuple as soon as possible. The details of these optimizations are discussed further in Section 5 as well.

4 Programming Model

In the following, we present a more detailed view on the programming model of DFI and its main abstractions by detailing the opportunities for setting up various communication flow types. In addition, the programming model will be demonstrated through a set of concrete use cases.

4.1 DFI Tuples

To allow processing of application-specific tuples between different end-points (i.e., threads) of DFI flows, DFI receives the tuple types through the passed schema on flow initialization. The schema can be constructed of various data types, that each mirrors the size of C++ types, specifically the LP64 data model (default data model in most Unix-based systems). The types, however, can be extended by the application to meet the need for other user-defined types.

DFI's type system enables efficient data processing: (1) Avoiding any type interpretation overhead is key to high-speed networks since every additional overhead can significantly reduce bandwidth or increase latency of the overall distributed algorithm [12, 30]. Tuple types are parameters of flows that are defined at flow initialization; i.e., no type interpretation happens at flow execution. Instead, efficient offset computation can be used to access attributes of a tuple (e.g., to make routing decisions). (2) The type system of DFI also allows applications to push down the processing to devices in the network, such that the interface is extensible towards leveraging the future generations of SmartNICs and programmable switches. For example, data aggregation of a DFI combiner flow (which is another DFI flow type) could be pushed into InfiniBand switches as we discuss below.

4.2 DFI Flows

So far we have only presented a concrete example of constructing a flow for shuffling tuples between a set of source and target threads. However, DFI defines flows with different characteristics to support the wide demands of data processing systems. Table 1 shows the three flow types in DFI, the communication topologies supported by the corresponding flows, as well as their declarative flow options.

The flow abstraction also offers easy adaptability of application algorithms, since different types of flows can be trivially exchanged to offer different behaviors. For instance, to change a symmetric repartition join algorithm into a fragment-and-replicate join, instead of using a shuffle flow that routes tuples based on the join key, use a replicate flow to replicate the inner table. Performing such

Flow type	*Communication topology*	*Flow options*
Shuffle flow	1:1, N:1, 1:N, N:M	Bandwidth/latency
Replicate flow	1:N, N:M	Bandwidth/latency + ordering guarantees
Combiner flow	N:1	Bandwidth/latency + various aggregations

Table 1: DFI flow types for a wide range of data-centric applications. Communication topologies and flow options further allow applications to adjust the behavior of flows based on application requirements.

algorithmic changes on typical solutions leveraging the RDMA verb interface would infer a significant rewrite of the communication relevant parts of the solution.

In the following, we discuss the different flow types and their potential use in data processing systems.

4.2.1 Shuffle Flow: The shuffle flow is a central abstraction of DFI, where various different communication patterns and routing options can be specified. The communication pattern is indirectly defined by declaring the participating sources and targets in the flow initialization, and can therefore follow 1:1, N:1, 1:N and N:M communication patterns between sending and receiving threads.

The routing of tuples from sources to targets can be defined in three ways in a shuffle flow: (1) The application specifies the shuffle key and let DFI handle the routing. (2) A routing function can be supplied for more control, e.g., to realize different partition functions such as range-partitioning or radix hash partitioning. (3) Lastly, it is also possible to directly specify the node identifier of a target thread on each push into the flow.

4.2.2 Replicate Flow: Another flow type that DFI provides is a so called replicate flow, which targets data processing tasks involving data duplication, such as replicated state machines, fragment-and-replicate join operators or data duplication for stream processing.

The performance of a naïve replication of tuples which uses multiple RDMA operations (i.e., one for each target), will quickly become limited by the outgoing link-speed of the source node; e.g., a replicate flow with 1 source and 8 targets, will have to divide the available network bandwidth at the source, if messages are replicated to all 8 targets on the source node. In DFI, we instead make use of RDMA multicast such that when enabled, messages are replicated in the network as to prevent the outgoing link of the source(s) from becoming a bottleneck.

For some applications using replication, ordering of messages plays an important role. An example of this is state machine replication, where the correctness depends on all replicas processing the incoming operations in the same order. Since many networks (including InfiniBand) do not provide this guarantee if multiple receivers are involved [22] (even not for simple networks with only one switch), replicate flows can be initialized to provide global ordering guarantees, such that all targets consume tuples out of the flow in the same order. Details on how global ordering is implemented for replicate flows in DFI are explained in Section 5.

4.2.3 Combiner Flow: The third flow type supported by DFI is the combiner flow. The focus of the combiner flow is many-to-one communication patterns which is typically used in aggregation scenarios, such as a SQL aggregation or a parameter server [23] for distributed machine learning. The combiner flow supports various different aggregations (e.g., SUM, COUNT, MIN, MAX) to be performed on the tuples.

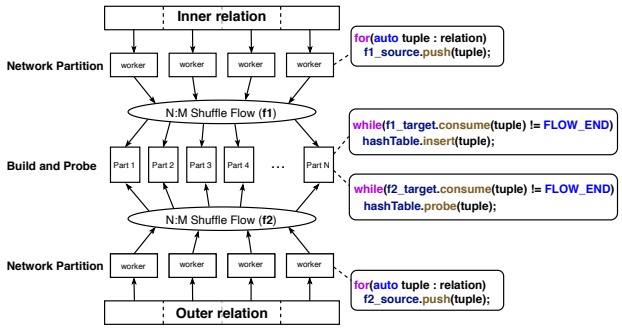


Figure 2: Distributed Radix Hash Join with DFI flows. Two shuffle flows are used to partition tuples across network, one for each relation.

Again while a naïve implementation would implement the reduction at the target node, the network can be used to accelerate the reduction. For example, InfiniBand offers the SHARP protocol [13], that enables in-network aggregations for high-speed InfiniBand networks and thus could help to mitigate when the in-bound network of the receiver becomes a bottleneck.

4.3 Use Cases

In the following we present two distributed data processing use cases and how they are realized through DFI: First, we discuss distributed joins for OLAP where the aim is to reduce the runtime by making efficient use of the available network bandwidth. Second, we present a distributed consensus use case where the performance criteria is low latency and high message throughput.

4.3.1 Distributed Radix Join: The distributed radix hash join is a popular join operator due to its dominating performance [2, 3]. The idea behind the radix hash join is to partition the input relations into such small partitions that the resulting hash tables fit into the CPU caches to reduce cache-misses.

In its original form the distributed radix join has a high level of complexity since multiple sender and receiver threads need to coordinate. For example, in [2, 3], histograms of buckets are precomputed in a first pass on each input table to allocate private memory buffers for each thread on the receiver node and then use coordination-free one-sided communication in a second pass to shuffle the data of each input table.

We argue that with DFI, the design of a distributed radix join is simpler while the performance is on par (and sometimes even better) with the latest distributed radix join implementations (as will be shown in Section 6.3). To realize the join with DFI, two bandwidth optimized shuffle flows are used as shown in Figure 2, one for shuffling each relation. Figure 2 also shows the pseudo-code how tuples can be pushed into the flows during network partitioning, and consumed at the target (i.e., receiver node) out of the flows for the relations to either build the hash table (for the inner relation) or probe the hash table (for the outer relation).

The shuffle flows for the join are initialized with one source per sender thread and one target per output partition. That way

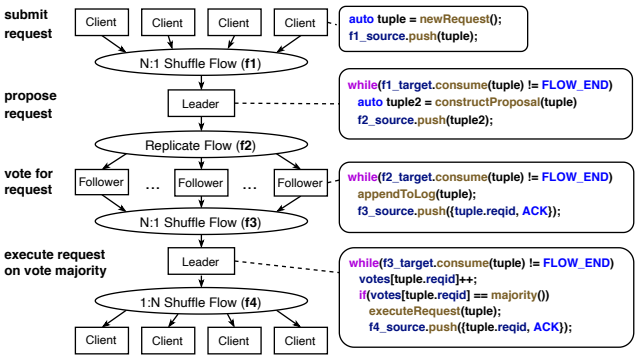


Figure 3: Leader-based consensus with DFI flows. Four flows are used to realize consensus between replicas.

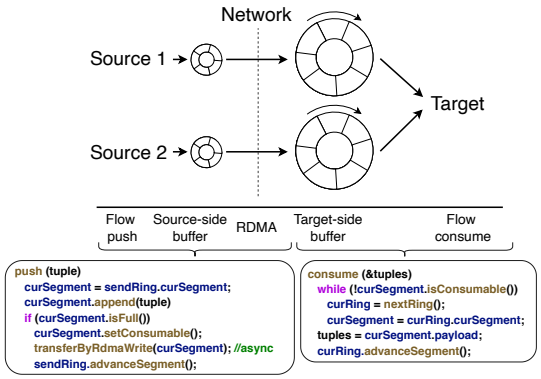


Figure 4: DFI flow implementation using ring buffers. In DFI flows, each source allocates a private target-side ring buffer to minimize coordination overhead.

the flow can used for achieving the desired partition fan-out. The routing of tuples to the partition-specific *targets* is done on a per thread level by passing a radix hash function to DFI as the routing function. This also leads to a noticeable reduction of complexity of the DFI join compared to the original RDMA-based distributed radix join since the histogram computation can be completely omitted. Moreover, the memory management of local and remote buffers is handled in DFI.

4.3.2 Distributed Consensus: Consensus in a distributed system describes the agreement of multiple (often asynchronous) participants on a single value, or a sequence of values, while tolerating the presence of faulty participants. It is a fundamental primitive in distributed computing which is needed, for example, for the reliable implementation of replicated state machines, leader election, or system reconfiguration.

Classical consensus protocols [20, 27] are centered around a centralized coordinator, called leader. The leader orders concurrently arriving requests of participants (i.e., clients) and forwards them to a set of so called followers. The followers vote for requests that they receive from the leader. Once the leader has received a majority of votes (itself included), the leader can notify the corresponding client that its request was agreed-upon. The high-level message flow of a leader-based consensus implementation using DFI can be modeled directly with the flows provided by DFI and is depicted in Figure 3. Figure 3 additionally shows pseudo-code of how these flows are used for the communication which we explain in the following.

Clients initially send their vote with an N:1 shuffle flow to the leader. The replicate flow is ideal to handle the communication from the leader to its followers, as all followers receive identical messages. The use of the RDMA multicast verbs built into DFI alleviates load placed on the leader compared to the naïve replication of messages. This is an interesting optimization, as the leader is typically a major bottleneck in consensus-based systems. Once followers received the request and voted for a result, they send the outcome back to the leader, again using a shuffle flow. In a last step the leader distributes the consensus-outcome to the client using the client IDs as the shuffle key.

An interesting optimization that DFI provides is to use the optimization option for global ordered multicast (also referred to as ordered unreliable multicast - OUM). In particular, Li et al. [22] propose a single round-trip consensus protocol based on OUM. While this work focuses on Ethernet-based systems, to our knowledge, DFI is the first system that can provide these semantics in the context of InfiniBand.

As we show in Section 6.3.2, using the ordered multicast significantly improves both throughput and latency compared to conventional consensus protocol designs using native RDMA that follow more classical consensus designs.

5 Flow Implementation

In this section, we discuss our implementation of DFI for a recent RDMA-capable InfiniBand EDR hardware stack to showcase the design choices of how to enable the key design principles discussed in Section 3. In future, we envision that different vendors can provide an efficient implementation of DFI for their network technology.

5.1 Flow Execution

The key design principles listed in Section 3 impose challenges for how the data transfer between the sources and targets is realized which are pivotal for distributed data processing. In the following, we give an overview of the flow execution (Section 5.1) and the buffer design (Section 5.2) for bandwidth optimized shuffle flows. Optimizations for latency-optimized flows and other flow types will be discussed at the end of this section.

On a high-level, to achieve the design goals listed in Section 3 for shuffle flows, DFI uses a private send/receive buffer for each pair of source and target threads as illustrated in Figure 4. The design of source- and target-side buffers follows a ring-based design where each ring is composed of a configurable number of segments and is allocated as one consecutive region in memory. The segment itself can be sized to contain a single tuple up to a batch of tuples. Therefore, the segment size is a tuning parameter that allows DFI to either optimize for bandwidth or latency independent of the tuple sizes used by the application.

One key question is how such a segmented ring design enables pipelining of tuples with low-overhead synchronization. In order to achieve pipelined data transfer between buffers (i.e., a decoupling of senders and receivers), one-sided RDMA writes are used to copy data asynchronously from sources to targets. This asynchronous data transfer using RDMA writes is implemented by the `transferByRdmaWrite` call in Figure 4. This method also implements the synchronization with the target buffer to not overwrite

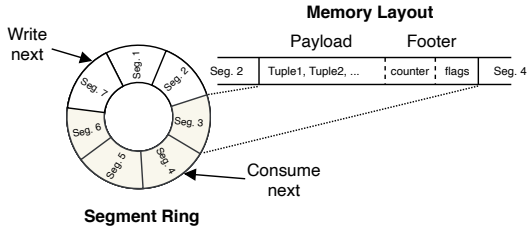


Figure 5: Target-side buffer structure. The segment ring data structure is a densely allocated memory region split up into segments. Segments are appended with small footers to handle coordination and fill grade of each segment.

any segments that has not been consumed yet. The synchronization is based on the metadata of each segment as we discuss next in Section 5.2. For latency-optimized flows (see Section 5.3), we instead use a credit-based approach to further reduce the overhead.

In setups with a very high number of sources and targets, a design with private buffers for each source / target combination can by first sight lead to high memory consumption. In DFI, however, applications can effectively reduce the memory consumption by reducing the number of segments per ring, since only a few segments are needed to achieve good pipelining and source / target decoupling. As we show in our evaluation, this can efficiently reduce the memory overhead while only affecting performance minimally.

Other approaches also employ a circular buffer for communication over RDMA, e.g., such as FaRM [9, 10]. While our buffer design shares some similarities with the buffer design of FaRM (e.g., using one-sided writes for data transfer), there are noticeable differences: (1) The aim of the FaRM design is only latency sensitive message-passing and hence does not provide a bandwidth optimized communication primitive. (2) The buffer design of FaRM targets only shuffle flows but no other flow types such as replication flows or combiner flows, which require additional optimizations as we discuss in Section 5.4.

5.2 Buffer Design

In the following we present further details of the buffer design for the bandwidth optimized setting. We first explain the design of the outgoing buffer on the source side before we then discuss the design of target-side buffers.

Source-side Buffer: The main difference is that source-side buffers use much fewer segments than target-side buffers to reduce the memory overhead of buffers. Smaller send buffers do not violate our goal to decouple sources and targets since data is sent directly once it is available. However, since data out of source buffers is transferred asynchronously we need more than one segment. For this reason, we need to ensure that an RDMA write of a segment has been carried out before the segment can be reused to avoid data loss. For achieving this, the source-side buffers use signaled RDMA writes, a technique to check if the asynchronous transfer is completed. However, in order to reduce the number of these checks, as they are quite expensive, the source only issues a signaled write once it wraps around the buffer.

Target-side Buffer: Target-side buffers use a slightly different design since sources and targets need to synchronize; i.e., targets need to decide which segments are ready to be read while sources must decide whether a remote segment was consumed already and thus can be reused by the source. For the synchronization each segment defines a footer which holds metadata about the segment as shown in Figure 5. The flags in the footer indicate whether a segment is *writable* or *consumable*. In a *writeable* state, a segment is free to be written by the source and *consumable* state indicates that the target can consume the tuples stored in the segment.

Unfortunately, an RDMA write of one segment is not guaranteed to be persisted atomically into remote memory since the segment might be split into multiple DMAs by the remote NIC. In order to avoid a checksum per segment, we make use of the fact that DMAs of the remote NIC are guaranteed to be written in an increasing memory order [9, 24]. Therefore, DFI places the metadata which indicates the state whether the segment is consumable or not *after* the payload of the segment. This ensures that when the target has detected a change of segment state, the payload of the segment is completely written.

From a target perspective, the footer is read when the target thread calls the consume function, which returns a pointer to the payload if the state is consumable and sets the state to writable on subsequent consume calls. Thereby allowing the application to process the returned tuples directly without memory copy. To write new segments to the target-side buffer the source first needs to verify if the current target segment is free and ready to reuse. Therefore, the source reads the footer of the remote target segment with RDMA reads to check if the state is writeable.

To efficiently read the remote segment states we pipeline the write of the current segment with the read of the footer of the next segment. In other words, while we transfer a source-side segment to the target-segment n we immediately read the footer of the target segment $n + 1$. Therefore, upon the next push call on the source, if the remote segment is detected to be writeable, the RDMA write can be executed directly without waiting. The RDMA write of the source includes the update of the footer for the target (i.e., setting its state back to consumable). If the remote segment is detected to be not writeable, the source periodically polls the segment footer that should be written next with a small random backoff, to avoid overloading the network with read requests.

Optimizations: For the buffer design, we use various optimizations for efficient RDMA. For instance, our design increases the chance to exploit DDIO which allows DMA data to be directly written to CPU caches. Additionally, we use common RDMA optimizations like inlining small messages and selective signaling. Moreover, in both buffers (source- and target-side), we additionally enable two performance-relevant settings in the InfiniBand stack: (1) We disable expensive spin locks in the RDMA library when using private buffers. (2) We use huge-pages for RDMA to avoid the high costs of TLB misses in the RDMA NIC.

5.3 Latency Optimization

The previously presented flow implementation is centered around maximizing bandwidth utilization. However, for some classes of distributed data processing systems (e.g., for OLTP or consensus) achieving low latency for data transfer is crucial. Hence DFI provides an optimization option for latency. This requires several changes in the buffer design and the overall execution flow.

A naïve way to support low latency would be to simply reduce the segment size of buffers to the size of an individual tuple and rely

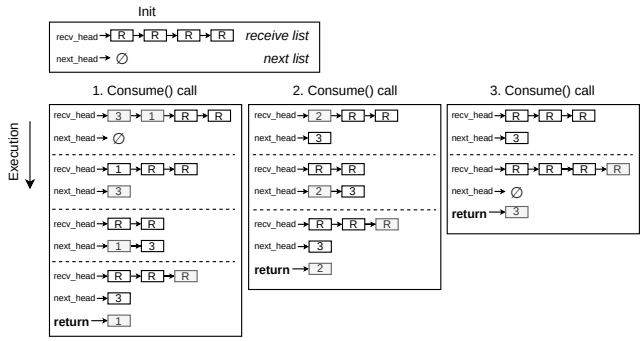


Figure 6: Ordering example for replicate flows: Receive requests (R) are initially posted and stored in the *receive list*. Subsequent consume calls detect incoming segments and insert segments into the *next list* in order. Messages are returned in order from *next list*.

on the same synchronization protocol between sources and targets. However, this implies that sources would need to check the footer of the next segment before every RDMA write to make sure that the segment is writable. In contrast to the bandwidth optimized version in which the cost of the additional read is amortized by the batch of tuples an additional read for each tuple transferred incurs high overhead. Hence, we use a different design to support low-latency data transfer with low overhead.

While the footer is still used by the target to decide if a segment is consumable, we use a credit system to decide how many segments a source can write without any synchronization. Therefore, a credit counter on the target side is used which is initialized to the number of segments in the ring, to reflect the number of tuples a source can write without overwriting tuples that haven't been consumed by the target yet. The credit counter is incremented by the target each time a tuple is consumed. The source thread holds a copy of the remote credit counter which is decremented on every RDMA write of a segment. Moreover, the remote credit is read once the local credit counter reaches a certain threshold.

5.4 Other Flow Types

The replicate and combiner flows employ similar buffer structures as the shuffle flow but introduce important optimizations.

Replicate Flow (no ordering): The replicate flow sends a tuple to a group of targets. In order to avoid that the outgoing network link of the source becomes a bottleneck, we exploit RDMA multicast which replicates tuples in the switch. However, RDMA multicast relies on two-sided communication over unreliable transport, which leads to changes in the buffer design and the control flow. Instead of detecting incoming messages on the target side by polling directly in main-memory, for two-sided RDMA primitives the target has to poll a completion queue. To avoid high overhead of coordinating for each RDMA send, we implemented a credit score for sources (similar as the one we used for latency-sensitive flows) to know for *all* targets of a replicate flow, how many messages can be sent without coordination. For this, we initially pre-populate the receive queues on the target sides with as many receive requests as given by the credit score and as soon as a segment has been written into the target-side buffer a new receive request will be added to the receive queue once the *consume* call on the target-side buffer has returned the payload.

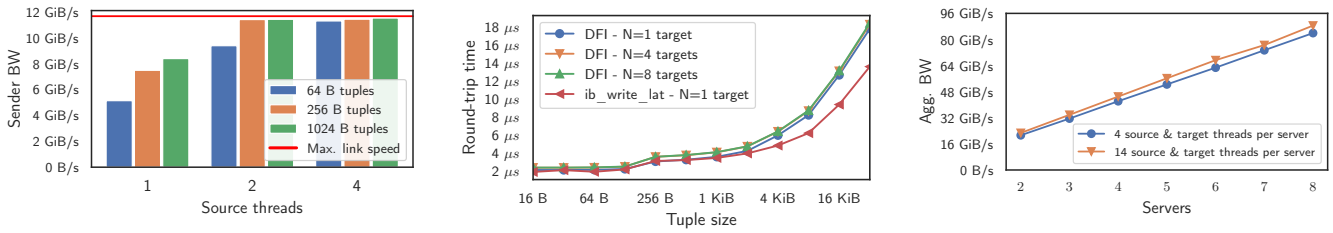
For coordinating credits at the source side we employ a back-flow from targets to source. This back-flow is used by targets to inform sources how many messages have been received which allows the sender to increase its credit score accordingly. Moreover, we add a sequence number to each segment such that targets can report missing segments to sources together with the credit back-flow. Lost segments are requested if a configurable timeout is reached. One important issue is that segments might arrive late (which were reported as lost by the target). In DFI this is handled by the target which filters out duplicate segments.

Replicate Flows (globally-ordered): DFI can also provide global ordering guarantees on replicate flows. Global ordering is a common primitive for distributed systems which, however, often needs to be implemented on the application layer. For instance, distributed consensus requires a global ordering as described in Section 4.3.2. DFI guarantees global ordering by implementing a so-called tuple sequencer, in which sources append sequence numbers to segments using an RDMA fetch-and-add on a global counter. With the advent of programmable switches, a tuple sequencer can instead be implemented in the network as shown in [22] to avoid the additional round-trip for the RDMA fetch-and-add. However, as we see in our experiments in Section 6.3, already the naïve solution with a global counter can provide benefits for consensus over solutions which rely on flows without ordering guarantees.

While a tuple sequencer adds global sequence numbers to segments, they can still arrive out-of-order at the different targets. On the target-side we thus have to ensure that segments arrive in the same order by reordering the incoming potential out-of-order segments. Figure 6 exemplifies how reordering is implemented on the target side. For reordering, two linked lists are used: a *receive list* for storing incoming segments in the arrival order and a *next list* for ensuring ordering.

In the example in Figure 6, on the first consume call, segments with sequence number 3 and 1 have been received. The head of the *receive list* (i.e., 3) is "moved" to the *next list* which involves no data copy but only pointer updates and the segment with number 1 is returned. For the second consume call, in the example, the segment with number 2 has arrived and is directly returned while the segment 3 is kept in the *next list* which is then returned for the last consume call. Losses in this protocol are detected (as before) through gaps in sequence numbers (in case a configurable timeout is reached). Optionally, we only notify the application on a consume call of gaps and its left up to the application to handle re-transmission (which is a feature we use for our consensus implementation).

Combiner Flows: A last flow type supported in DFI is the combiner flow. The flow directly follows the design of a shuffle flow (using a N:1 topology) but adds functionality to aggregate tuples in the target buffer using an aggregate function/ group-by specification as explained in Section 4. An interesting optimization is to use in-network-processing capabilities such as the SHARP protocol that enables in-network-aggregation in a switch to avoid incast congestion on the in-going link to the target of a combiner flow. However, implementing this is an interesting avenue of future work.



(a) Sender bandwidth (1:8) **(b) Median latency (1:N)** **(c) Aggr. sender bandwidth (N:N)**

Figure 7: Shuffle flow performance. DFI achieves max. bandwidth and low latency for various scenarios.

6 Experimental Evaluation

We evaluated DFI on three different levels. First we look at the efficiency of DFI in terms of how well the high-level interface utilizes the network compared to low-level RDMA verbs. Next, we provide a detailed comparison of DFI and MPI and argue that MPI is the wrong abstraction for data processing systems. Lastly, we evaluate DFI for two typical use cases in data processing systems and compare the implementations to existing state-of-the-art solutions.

In all experiments we use the notation (N:M) to indicate the number of servers involved in a flow topology. The number of threads per server is reported separately per experiment.

Evaluation Environment: All experiments were conducted on an 8 node cluster where 6 of the nodes are equipped with two Intel(R) Xeon(R) Gold 5120 CPUs (14 cores) and 512 GB main-memory, and 2 nodes equipped with two Intel(R) Xeon(R) Gold 5220 CPUs (18 cores). Hyper-threading is disabled for all nodes. Each node is equipped with two Mellanox ConnectX-5 MT27800 NICs (InfiniBand EDR 4x NICs, 100 Gbps), connected to one SB7890 InfiniBand switch. The operating system is Ubuntu 18.04.1 LTS, with Linux 4.15.0-47 kernel on all nodes. DFI is implemented with C++17 and compiled with gcc-7.3.0.

6.1 Experiment 1: Efficiency of DFI

The first experiment shows the efficiency of DFI compared to low-level RDMA verbs.

6.1.1 Shuffle Flows: In the following, shuffle flows are evaluated with bandwidth and latency optimization and lastly, a scale-out experiment is presented.

Bandwidth-Optimized: Our first experiment evaluates performance for the shuffle flow from 1 server to 8 servers with varying tuple sizes. Further, we vary the number of sources (threads) pushing tuples into the flow. The batch size for the bandwidth optimized version in our experiments is 8 KiB. We choose a batch size of 8 KiB as this offers a good tradeoff between network bandwidth and time until the batch is filled.

Figure 7a reports results for the bandwidth-optimized flow. As we see, in most settings we achieve the full network bandwidth. Only, the single-threaded scenario shows some overhead since batches must first be filled on the source side with individual tuples before they can be transferred to the target. This overhead can, however, be amortized by using more threads per server as shown in Figure 7a. Due to the efficient multi-threading support of DFI, we see that from two source threads on, the bandwidth is limited by the speed of the outgoing link (100 Gbps / 11,64 GiB/s - red line) for tuple sizes larger than 128 B. Moreover, when using 4 threads the maximal bandwidth is achieved independent of tuple sizes.

Latency-Optimized: We additionally evaluated the shuffle flow that implements latency optimizations. For measuring latency, two shuffle flows are used to implement a request and response pattern to measure the round-trip time between two nodes. To show that DFI's buffer design only adds minimal latency overhead, we compare the latency of DFI to *ib_write_lat*[2] which is a standard tool for performance testing that uses low-level verbs to implement a round-trip between a sender and a receiver node. For DFI, we additionally used a varying number of receiving servers (1, 4, and 8) to observe the effect on latency when shuffling to various destinations.

As we see in Figure 7b, the median latency of DFI for one full round-trip only adds minimal overhead when compared to *ib_write_lat* which is due to buffer. Moreover, keep in mind that DFI provides a high-level abstraction and thus not only reduces application complexity but also provides several optimizations to applications. This includes an efficient overlapping of compute and communication as well as many other optimizations such as efficient replication and ordering guarantees. As we show in Section 6.3, this enables DFI to provide superior performance in different use cases when compared to existing approaches that are using other interfaces (low-level RDMA verbs or MPI).

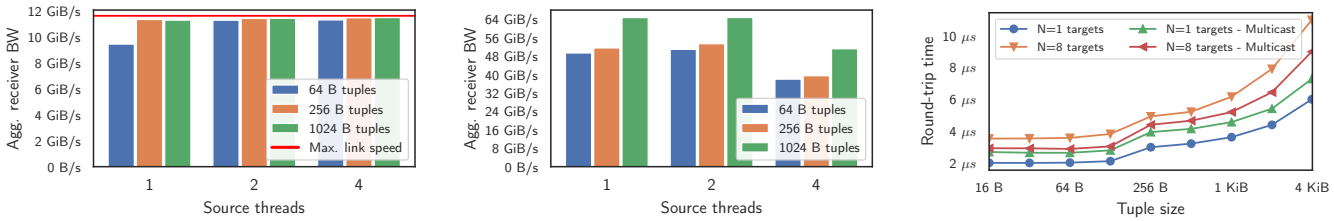
Moreover, the advantage of DFI compared to plain RDMA is the encapsulated memory management, which allows applications to use RDMA transparently without hand-tuned memory management while still achieving optimal performance. The experiment shows that this abstraction hardly incurs any overhead compared to *ib_write_lat*. For multiple targets the latency of DFI is only slightly higher due to the internal routing in the shuffle flow. Multiple targets are not supported by *ib_write_lat* though (i.e., *ib_write_lat* uses only one target in this experiment).

Scale-out: Since data processing systems often need to scale out to many nodes, we conducted a scale-out experiment for the shuffle flow, increasing the number of source and target servers. Moreover, we use 14 sources and targets on all nodes which in total gives 12544 unique source/target connections for the maximal number of nodes used. As shown in Figure 7c, DFI scales linearly with the number of nodes (as indicated by the x-axis), effectively increasing the aggregated bandwidth with the link-speed of each added node.

6.1.2 Replicate Flows: Next we benchmarked the replicate flow in terms of achievable bandwidth, with and without multicast and finally the latency behavior.

Bandwidth and Multicast Optimization: We tested the replicate flow bandwidth for two optimization settings, naïve one-sided replication and multicast. The evaluation was conducted by replicating data from 1 node to 8 nodes. In Figure 8a, the reported bandwidth

[2] https://github.com/linux-rdma/perftest.



(a) Aggregated receiver bandwidth (naïve one-sided, 1:8)

(b) Aggregated receiver bandwidth (multicast, 1:8)

(c) Median latency (all targets, 1:N)

Figure 8: Replicate flow performance. DFI achieves max. bandwidth and low latency for various scenarios.

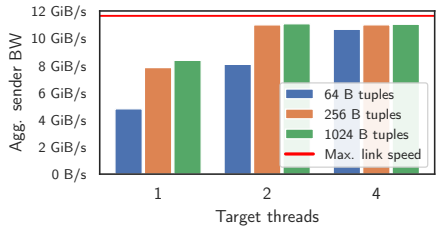


Figure 9: Combiner flow with sum aggregation (8:1). Aggregated sender bandwidth for various scenarios.

already achieves the practical network limit of the sender with 1 thread and tuples bigger than 64 B. In comparison to the shuffle flow bandwidth in Figure 7a, the replicate performance reaches max. bandwidth earlier due to network messages being replicated with one-sided writes that are issued in parallel by the NIC, reducing the per-tuple overhead.

As we see in Figure 8a, the naïve replication is limited by the network speed of the sender. However, the replicate flow also provides a multicast optimization that replicates messages in the switch. With the multicast optimization (Figure 8b) the bandwidth goes beyond the 100 Gbps (11,64 GiB/s) limit of the outgoing sender link and reaches up to 64 GiB/s. Using more source-threads on the sender node, however, does not yield better performance as the NIC inhibits bad multi-thread scalability within same multicast group.

Latency Optimization: For evaluating the latency for the replicate flow, we conducted an experiment where a source replicates a request to 8 targets and measures the time for it to get replies from all. The reported latency for naïve and multicast replication is shown in Figure 8c. The naïve (one-sided) replication achieves the lowest latency with only 1 target, but increases with more targets. For multicast, though, the increase in latency from 1 to 8 targets is much smaller and outperforms the naïve implementation.

6.1.3 Combiner Flows: The last flow evaluated in this experiment is the combiner flow. Figure 9 reports the aggregated sender bandwidth for a combiner flow with a sum aggregation. As seen for 2 and 4 threads, the bandwidth becomes limited by the in-going link to the target node. As an avenue of future work, the advent of in-network processing such as SHARP [13] could be used to further speed up the aggregation beyond the limits of the in-going link.

6.1.4 Memory Consumption: In order to provide insight to the degree of memory consumed through DFI, we observed the memory allocated in the scale-out experiment of the shuffle flow in Figure 7c. This setup is the most memory consuming one since each pair of source/target threads uses a private send/receive buffer.

For the smallest setup with only 4 source and 4 target threads per node in a cluster with two nodes in total, DFI consumes just 16 MiB per node[3]. Moreover, when increasing the setup to 8 nodes where each node has again 4 source and 4 target threads, the memory consumption grows only to 64 MiB. For the largest setup in Figure 7c (14 source & target threads and 8 servers in total) DFI consumes 785.5 MiB in total on each node.

However, as discussed in our experiments before, 4 source/target threads per node are sufficient at the moment to saturate the high bandwidth provided by the InfiniBand network in our setup. Moreover, the size of buffers in DFI are configurable. Hence, even smaller memory footprints can be achieved. As an example reducing the number of segments to 50% (i.e., 16 per buffer) the performance on 8 nodes just decreases by 2.7%, and further reducing the size to 25% (i.e., 8 per buffer) decreases performance by 8%.

Key Insights (Exp. 1): Our results show, that DFI flows can provide a high-level abstraction with no or only negligible overhead compared to low-level RDMA verbs.

6.2 Experiment 2: DFI vs. MPI

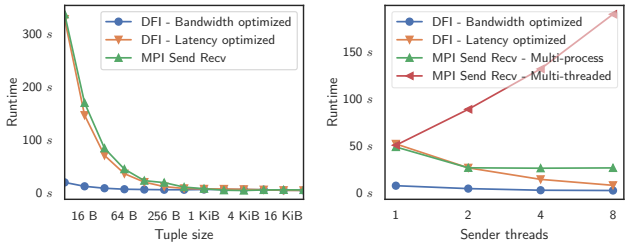
In the following experiment we evaluate the performance of MPI and DFI in various settings. First we will show point-to-point performance for single-threaded and multi-threaded setups. Next we look at the collective functions provided by MPI and compare their usage for a typical shuffle scenario. For MPI we use the latest version (4.0.3rc4) shipped with HPC-X (2.6.0) for our InfiniBand hardware. MPI is therefore highly optimized to make use of the RDMA primitives offered by the network.

6.2.1 Point-to-Point Primitives: As described in Section 2, MPI is process-centric, meaning it achieves parallelism by executing parts of the program in multiple processes on the same server. We therefore first compare MPI and DFI in a single-threaded setup before we then study the multi-threaded extensions provided by the MPI version of our InfiniBand deployment.

Single-threaded: Figure 10a reports the runtime for transferring a fixed table size (16 GiB). The *MPI_Send* and *MPI_Recv* primitives are used for sending the various tuple sizes, thereby using both MPI and DFI on a tuple-basis. Since MPI does not support any bandwidth/batching optimizations, the runtime is high for lower tuple-sizes since the network is inefficiently used. The bandwidth optimization for DFI makes efficient use of the network and therefore achieves a small runtime already for small tuple sizes.

Multi-threaded: Data processing systems typically use multi-threading (and not multi-process) to achieve parallelism while being able to

[3]In DFI, each buffer uses 32 segments each having a 8 KiB size in its default configuration.



(a) Single-threaded (b) Multi-threaded - 64 B tuples

Figure 10: MPI vs. DFI - point-to-point runtime

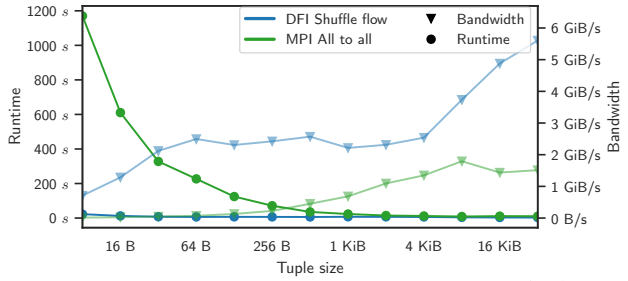


Figure 11: MPI vs DFI - collective shuffling (8:8)

share data between threads within the same virtual memory space. As such we evaluate the multi-threaded performance for DFI and MPI (using MPI_THREAD_MULTIPLE where multiple threads may call MPI primitives at once with no restrictions).

Figure 10b reports the runtime with an increasing number of threads. While DFI scales with the number of threads, the performance for multi-threaded MPI gets worse (red line). We analyzed this behavior and found that for data-heavy transfers (which MPI was not designed for), even a few threads lead to high internal contention on latches of MPI which causes the significant drop in performance.

The alternative in MPI for achieving parallelism per node is to use multiple processes instead of multiple threads per server. As we show in Figure 10b (green line) this leads to a better scalability than the multi-threaded MPI. However, multi-process solutions come at the cost that common data structures need to be accessed via (more expensive) shared memory.

6.2.2 Collective Primitives: MPI also offers primitives that encapsulates communication between multiple nodes. Since MPI provides a rich library of collectives and we cannot provide an analysis in this paper which covers all collectives, we focus on the *MPI_Alltoall* collective since it resembles in a closest manner the semantics of an N:M shuffle flow of DFI.

Shuffle (Pipelined): We first look at the shuffle performance when using MPI in a streaming-based manner (i.e., we shuffle data in mini-batches with a size of 8 tuples - on average one tuple per target). In this experiment, only one thread per node is used (for MPI and DFI) which scans a table and shuffles the tuples based on their keys. Multi-threading in MPI does not provide any benefit as we have seen before. For shuffling, we are using *MPI_Alltoall* (position-based) which uses a send buffer of the size of all nodes. As Figure 11 shows, the runtime of MPI for smaller tuple sizes is very high since the network is not utilized efficiently. However, as the tuple size increases, the bandwidth approximates that of DFI.

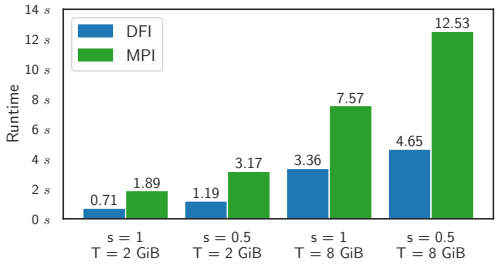


Figure 12: MPI vs DFI - collective shuffling (8:8) - One node straggling - s: straggling (s ∗ CPU freq.) - T: table size

Shuffle (Batched): To increase the network efficiency for MPI, we locally pre-shuffle the table on the shuffle key and invoke a *MPI_Alltoall* function for the complete batch. While this improves the bandwidth, collective functions are then susceptible to straggling behavior.

We evaluated the performance impact of MPI and DFI with one straggling node. To simulate a straggler, we decrease the CPU frequency of one of the nodes. The result is shown in Figure 12, where both the table sizes and straggling are varied. The increase of runtime for MPI with a straggling node (i.e., $s = 0.5$) comes from the fact that collective functions are blocking until all data is ready to be sent, and therefore limits the pipelining possibilities.

This is different for DFI. While DFI is also affected by straggling, it can constantly send data while the MPI implementation only starts the transfer once all data is available. Hence, DFI better overlaps the communication with the computation and therefore the straggling effect is less severe.

Key Insights (Exp. 2): In this experiment, we compared DFI to MPI. The experiments confirmed our speculations: (1) MPI neither provides efficient multi-threading, (2) nor does MPI allow to efficiently overlap compute and computation and hence support efficient pipelining.

6.3 Experiment 3: Use Cases

In the last experiments we evaluate DFI by implementing the two use cases we discussed in Section 4.3.

6.3.1 Distributed Joins: Distributed joins are crucial operators in OLAP due to large amounts of data having to be transferred across the network, and therefore a good candidate to evaluate bandwidth-optimized flows of DFI.

Radix Join: We implemented a distributed radix hash join on DFI and compared its performance to a state-of-the-art implementation for RDMA using MPI [2]. Both implementations employ the same optimizations (e.g., write-combine buffer in partitioning phase and tuple compression). However, the MPI join of [2] uses multi-process parallelism while our join uses multi-threading instead. Figure 13 shows the average runtime of the two joins for all 8 nodes.

The DFI radix join achieves the best runtime mainly due to two design choices of DFI. At first, the DFI radix join does not need to first compute a global histogram of the partition buckets. The MPI radix join in [2] makes use of one-sided *MPI_Put* primitives. In order to achieve coordination free writes, it thus has to compute exclusive writing offsets for each partition using one additional pass. Different from this, DFI encapsulates the memory management through our buffer design which makes the additional pass superfluous.

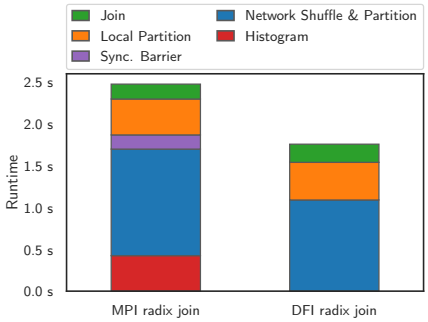


Figure 13: Distributed radix join - 8 nodes, 64 threads (DFI)/64 processes (MPI) in total. 2.56 B ⋈ 2.56 B tuples.

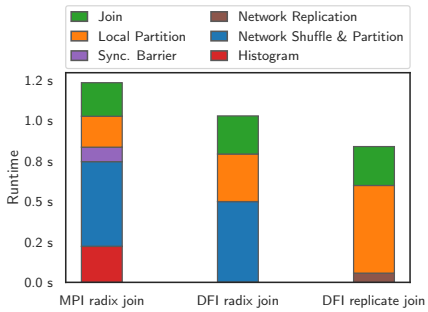


Figure 14: Distributed joins - 8 nodes, 64 threads (DFI)/64 processes (MPI) in total. 2.56 M ⋈ 2.56 B tuples.

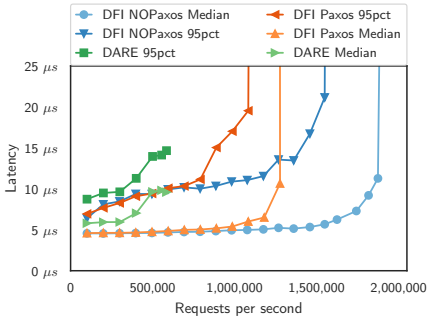


Figure 15: Performance comparison of DARE [28] with DFI-based implementations of Multi-Paxos and NOPaxos

The other reason for the runtime gap is due to the synchronization barrier needed in the MPI radix join after the network partition phase. Here, the join algorithm needs to make sure that all data has arrived before starting to process the local partitioning. While the data in this experiment is uniformly distributed, some runtime variance between multiple parallel workers still exists and is more pronounced in high-speed networks. This synchronization is not needed with DFI, since incoming tuples can already be processed when they arrive in a streaming-wise fashion.

Join Adaptability: Flows in DFI offer a high-level abstraction which encapsulates the data transfer of applications. As a result, it is trivial to adapt algorithms to use a different communication pattern. To demonstrate this, we adapted our radix hash join implementation to a fragment-and-replicate join variant which uses one replicate flow that replicates the inner table on all nodes. Figure 14 shows the runtimes of the three different join implementations with a smaller inner table (1000× smaller than the outer table). The replication of the small inner table is comparably cheap compared to shuffling the big outer table over the network. Overall, for this setup this helps to further reduce the overall runtime by another 20%,

6.3.2 State Machine Replication: In this experiment, we implemented a simple key-value store that replicates data using a consensus protocol. For the experiment, we used two different consensus protocols, classical Multi-Paxos [20] and NOPaxos [22]. We modeled the normal, failure-free operation of Multi-Paxos as depicted in Figure 3. For NOPaxos, we implemented its *normal operation* protocol, which relies on the OUM primitive that can be provided by DFI's replicate flow, as well as its *gap agreement* protocol to detect lost messages. We compare both implementations with DARE [28], a state-of-the-art replicated key-value store that is based on a hand-crafted consensus protocol and heavily relies on one-sided RDMA.

We deployed all approaches with five replicas (a leader and four followers). Load was generated by six clients distributed across three separate nodes. Clients submitted 64 byte sized requests using YCSB's read-dominated workload [7] (95% reads and 5% writes). The results are shown in Figure 15.

The two DFI-based implementations consistently outperform DARE in our settings in both achieved throughput and latency. This is caused mainly by DARE's sequential design. First, each DARE client cannot submit a new request until it has received the result from its previous request, which limits its achievable throughput.

Second, DARE's write protocol serializes requests. While this limitation is mitigated by separately batching reads and writes, a mix of both request types frequently interrupts batches [32]. This is confirmed by DARE's own evaluation [28].

Our Multi-Paxos and NOPaxos implementation exhibit near-identical response latencies as long as they are not saturated. This appears counter-intuitive at first, as Multi-Paxos requires four message delays to respond to a client, whereas two messages delays suffices for NOPaxos as long as no messages are lost. However, fetching a global sequence number from the tuple sequencer of the ordered replicate flow incurs an additional two message delays.

For a load higher than 700k requests/s, we see benefits of our NOPaxos over our Multi-Paxos implementation. Under this load, the leader in Multi-Paxos becomes saturated as it has to repeatedly collect responses from a majority of replicas. In contrast, in NOPaxos the clients themselves collect these responses. This alleviates the burden placed on the leader in Multi-Paxos, which leads to stable response latencies in DFI's NOPaxos up to even higher request rates of almost 1.5M (95th percentile).

Key Insights (Exp. 3): In summary, DFI does not only achieve a better performance for distributed joins and consensus than state-of-the-art, but also offers an ease-of-use high-level abstraction to implement efficient solutions with a low code complexity.

7 Conclusions

In this paper, we presented DFI, a new data-centric interface for fast networks. With our implementation for InfiniBand we have shown that DFI adds only minor overhead compared to low-level abstractions such as RDMA verbs. Moreover, by implementing two use cases, we demonstrated that DFI can efficiently support data-centric applications with different requirements (high-bandwidth vs. low-latency) at high performance.

In future, we plan to integrate further useful extensions into DFI flows such as fault-tolerance as well as elasticity of flows to add/remove nodes at runtime. Furthermore, by open-sourcing our implementation, we hope to stimulate not only follow-up research but also allow that commercial vendors will provide a DFI implementation also for other high-speed network stacks.

Acknowledgments

This work was partially funded by the German Research Foundation (DFG) under the grants BI2011/1 & BI2011/2 (DFG priority program 2037), the DFG Collaborative Research Center 1053 (MAKI) as well as gifts from Mellanox and Huawei.

References

[1] G. Alonso, C. Binnig, I. Pandis, K. Salem, J. Skrzypczak, R. Stutsman, L. Thostrup, T. Wang, Z. Wang, and T. Ziegler. DPI: the data processing interface for modern networks. In *CIDR 2019, 9th Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 13-16, 2019, Online Proceedings*, 2019.

[2] C. Barthels et al. Distributed join algorithms on thousands of cores. *Proc. VLDB Endow.*, 10(5):517–528, 2017.

[3] C. Barthels, S. Loesing, G. Alonso, and D. Kossmann. Rack-scale in-memory join processing using rdma. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, page 1463–1475, New York, NY, USA, 2015. Association for Computing Machinery.

[4] C. Binnig, A. Crotty, A. Galakatos, T. Kraska, and E. Zamanian. The end of slow networks: It's time for a redesign. *Proc. VLDB Endow.*, 9(7):528–539, 2016.

[5] Q. Cai, W. Guo, H. Zhang, D. Agrawal, G. Chen, B. C. Ooi, K. Tan, Y. M. Teo, and S. Wang. Efficient distributed memory management with RDMA and caching. *Proc. VLDB Endow.*, 11(11):1604–1617, 2018.

[6] J. Cipar, Q. Ho, J. K. Kim, S. Lee, G. R. Ganger, G. Gibson, K. Keeton, and E. P. Xing. Solving the straggler problem with bounded staleness. In P. Maniatis, editor, *14th Workshop on Hot Topics in Operating Systems, HotOS XIV, Santa Ana Pueblo, New Mexico, USA, May 13-15, 2013*. USENIX Association, 2013.

[7] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing, SoCC 2010, Indianapolis, Indiana, USA, June 10-11, 2010*, pages 143–154, 2010.

[8] A. Dragojevic et al. RDMA reads: To use or not to use? *IEEE Data Eng. Bull.*, 40(1):3–14, 2017.

[9] A. Dragojevic, D. Narayanan, M. Castro, and O. Hodson. Farm: Fast remote memory. In R. Mahajan and I. Stoica, editors, *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2014, Seattle, WA, USA, April 2-4, 2014*, pages 401–414. USENIX Association, 2014.

[10] A. Dragojevic, D. Narayanan, E. B. Nightingale, M. Renzelmann, A. Shamis, A. Badam, and M. Castro. No compromises: distributed transactions with consistency, availability, and performance. In *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015*, pages 54–70, 2015.

[11] P. Fent, A. van Renen, A. Kipf, V. Leis, T. Neumann, and A. Kemper. Low-latency communication for fast DBMS using RDMA and shared memory. In *36th IEEE International Conference on Data Engineering, ICDE 2020, Dallas, TX, USA, April 20-24, 2020*, pages 1477–1488, 2020.

[12] P. W. Frey and G. Alonso. Minimizing the hidden cost of RDMA. In *29th IEEE International Conference on Distributed Computing Systems (ICDCS 2009), 22-26 June 2009, Montreal, Québec, Canada*, pages 553–560. IEEE Computer Society, 2009.

[13] R. L. Graham, D. Bureddy, P. Lui, H. Rosenstock, G. Shainer, G. Bloch, D. Goldenberg, M. Dubman, S. Kotchubievsky, V. Koushnir, L. Levi, A. Margolin, T. Ronen, A. Shpiner, O. Wertheim, and E. Zahavi. Scalable hierarchical aggregation protocol (sharp): A hardware architecture for efficient data reduction. In *First International Workshop on Communication Optimizations in HPC, COMHPC@SC 2016, Salt Lake City, UT, USA, November 18, 2016*, pages 1–10. IEEE, 2016.

[14] W. Gropp et al. *Using Advanced MPI: Modern Features of the Message-Passing Interface*. The MIT Press, 2014.

[15] C. Jia, J. Liu, X. Jin, H. Lin, H. An, W. Han, Z. Wu, and M. Chi. Improving the performance of distributed tensorflow with RDMA. *Int. J. Parallel Program.*, 46(4):674–685, 2018.

[16] A. Kalia, M. Kaminsky, and D. Andersen. Datacenter rpcs can be general and fast. In *16th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2019, Boston, MA, February 26-28, 2019*, pages 1–16, 2019.

[17] A. Kalia, M. Kaminsky, and D. G. Andersen. Using RDMA efficiently for key-value services. In F. E. Bustamante, Y. C. Hu, A. Krishnamurthy, and S. Ratnasamy, editors, *ACM SIGCOMM 2014 Conference, SIGCOMM'14, Chicago, IL, USA, August 17-22, 2014*, pages 295–306. ACM, 2014.

[18] A. Kalia, M. Kaminsky, and D. G. Andersen. Fasst: Fast, scalable and simple distributed transactions with two-sided (RDMA) datagram rpcs. In K. Keeton and T. Roscoe, editors, *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*, pages 185–201. USENIX Association, 2016.

[19] S. J. Kang, S. Y. Lee, and K. M. Lee. Performance comparison of openmp, mpi, and mapreduce in practical problems. *Adv. MultiMedia*, 2015, Jan. 2015.

[20] L. Lamport et al. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.

[21] B. Li, Z. Ruan, W. Xiao, Y. Lu, Y. Xiong, A. Putnam, E. Chen, and L. Zhang. Kv-direct: high-performance in-memory key-value store with programmable NIC. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP)*, pages 137–152, 2017.

[22] J. Li, E. Michael, N. K. Sharma, A. Szekeres, and D. R. K. Ports. Just say NO to paxos overhead: Replacing consensus with network ordering. In K. Keeton and T. Roscoe, editors, *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*, pages 467–483. USENIX Association, 2016.

[23] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B. Su. Scaling distributed machine learning with the parameter server. In J. Flinn and H. Levy, editors, *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014*, pages 583–598. USENIX Association, 2014.

[24] J. Liu, J. Wu, and D. K. Panda. High performance rdma-based MPI implementation over infiniband. *Int. J. Parallel Program.*, 32(3):167–198, 2004.

[25] X. Lu, D. Shankar, S. Gugnani, and D. K. Panda. High-performance design of apache spark with RDMA and its benefits on various workloads. In J. Joshi, G. Karypis, L. Liu, X. Hu, R. Ak, Y. Xia, W. Xu, A. Sato, S. Rachuri, L. H. Ungar, P. S. Yu, R. Govindaraju, and T. Suzumura, editors, *2016 IEEE International Conference on Big Data, BigData 2016, Washington DC, USA, December 5-8, 2016*, pages 253–262. IEEE Computer Society, 2016.

[26] C. Mitchell, Y. Geng, and J. Li. Using one-sided RDMA reads to build a fast, cpu-efficient key-value store. In *USENIX Annual Technical Conference*, pages 103–114, 2013.

[27] D. Ongaro and J. K. Ousterhout. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference, USENIX ATC '14, Philadelphia, PA, USA, June 19-20, 2014*, pages 305–319, 2014.

[28] M. Poke and T. Hoefler. DARE: high-performance state machine replication on RDMA networks. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing, HPDC 2015, Portland, OR, USA, June 15-19, 2015*, pages 107–118, 2015.

[29] W. Rödiger, S. Idicula, A. Kemper, and T. Neumann. Flow-join: Adaptive skew handling for distributed joins over high-speed networks. In *32nd IEEE International Conference on Data Engineering, ICDE 2016, Helsinki, Finland, May 16-20, 2016*, pages 1194–1205. IEEE Computer Society, 2016.

[30] W. Rödiger, T. Mühlbauer, A. Kemper, and T. Neumann. High-speed query processing over high-speed networks. *Proc. VLDB Endow.*, 9(4):228–239, 2015.

[31] C. B. Walton, A. G. Dale, and R. M. Jenevein. A taxonomy and performance model of data skew effects in parallel joins. In G. M. Lohman, A. Sernadas, and R. Camps, editors, *17th International Conference on Very Large Data Bases, September 3-6, 1991, Barcelona, Catalonia, Spain, Proceedings*, pages 537–548. Morgan Kaufmann, 1991.

[32] C. Wang, J. Jiang, X. Chen, N. Yi, and H. Cui. APUS: fast and scalable paxos on RDMA. In *Proceedings of the 2017 Symposium on Cloud Computing, SoCC 2017, Santa Clara, CA, USA, September 24-27, 2017*, pages 94–107, 2017.

[33] X. Wei, Z. Dong, R. Chen, and H. Chen. Deconstructing rdma-enabled distributed transactions: Hybrid is better! In *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*, pages 233–251, 2018.

[34] X. Wei, J. Shi, Y. Chen, R. Chen, and H. Chen. Fast in-memory transaction processing using RDMA and HTM. In *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015*, pages 87–104, 2015.

[35] J. Worringen. Pipelining and overlapping for MPI collective operations. In *28th Annual IEEE Conference on Local Computer Networks (LCN 2003), The Conference on Leading Edge and Practical Computer Networking, 20-24 October 2003, Bonn/Königswinter, Germany, Proceedings*, pages 548–557, 2003.

[36] J. Xue, Y. Miao, C. Chen, M. Wu, L. Zhang, and L. Zhou. Fast distributed deep learning over RDMA. In G. Candea, R. van Renesse, and C. Fetzer, editors, *Proceedings of the Fourteenth EuroSys Conference 2019, Dresden, Germany, March 25-28, 2019*, pages 44:1–44:14. ACM, 2019.

[37] D. Y. Yoon, M. Chowdhury, and B. Mozafari. Distributed lock management with rdma: Decentralization without starvation. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, page 1571–1586, New York, NY, USA, 2018. Association for Computing Machinery.

[38] E. Zamanian, C. Binnig, T. Kraska, and T. Harris. The end of a myth: Distributed transaction can scale. *Proc. VLDB Endow.*, 10(6):685–696, 2017.

[39] E. Zamanian, J. Shun, C. Binnig, and T. Kraska. Chiller: Contention-centric transaction execution and data partitioning for modern networks. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, page 511–526, New York, NY, USA, 2020. Association for Computing Machinery.

[40] T. Ziegler, V. Leis, and C. Binnig. Rdma communciation patterns. *Datenbank-Spektrum*, 20(3):199–210, Nov 2020.

[41] T. Ziegler, S. Tumkur Vani, C. Binnig, R. Fonseca, and T. Kraska. Designing distributed tree-based index structures for fast rdma-capable networks. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, page 741–758, New York, NY, USA, 2019. Association for Computing Machinery.