DPI: The Data Processing Interface for Modern Networks

Gustavo Alonso[1], Carsten Binnig[2], Ippokratis Pandis[3], Kenneth Salem[4], Jan Skrzypczak[5], Ryan Stutsman[6], Lasse Thostrup[2], Tianzheng Wang[7], Zeke Wang[1], Tobias Ziegler[2]

[1] ETH Zurich, Switzerland [2] TU Darmstadt, Germany [3] Amazon Web Services, USA [4] University of Waterloo, Canada
[5] Zuse Institute Berlin, Germany [6] University of Utah, USA [7] Simon Fraser University, Canada

ABSTRACT

As data processing evolves towards large scale, distributed platforms, the network will necessarily play a substantial role in achieving efficiency and performance. Increasingly, switches, network cards, and protocols are becoming more flexible while programmability at all levels (aka, software defined networks) opens up many possibilities to tailor the network to data processing applications and to push processing down to the network elements.

In this paper, we propose DPI, an interface providing a set of simple yet powerful abstractions flexible enough to exploit features of modern networks (e.g., RDMA or in-network processing) suitable for data processing. Mirroring the concept behind the Message Passing Interface (MPI) used extensively in high-performance computing, DPI is an interface definition rather than an implementation so as to be able to bridge different networking technologies and to evolve with them. In the paper we motivate and discuss key primitives of the interface and present a number of use cases that show the potential of DPI for data-intensive applications, such as analytic engines and distributed database systems.

1 INTRODUCTION

The computer networks available in data centers and clusters are evolving rapidly, increasingly providing sophisticated capabilities such as RDMA (Remote Direct Memory Access), in-network processing, and customizable communication protocols. Once the province of specialized, expensive networks, the new functionality is becoming available in off-the-shelf networks as well. An example of how these advances can help with data intensive applications is RDMA, the ability to directly read or write the memory of remote machines without involving the remote CPU. RDMA makes data transfer more efficient, and it frees up computing capacity, which can lead to substantial performance gains [6–9, 16, 17, 20–23, 27, 28]. Unfortunately, using RDMA is complicated because it lacks higher-level abstractions [10]. Recent work on using RDMA in relational databases has shown that the design involves many low-level, yet significant, decisions around connection management, memory allocation, and the choice of which RDMA operations to use [1, 3].

This fragile dependency on low-level design aspects and lack of portability across networks is not unique to RDMA; it affects other technologies like smart NICs (Network Interface Cards) and programmable switches as well [11]. This is concerning because modern networks are increasingly software-defined, and there is a growing need to tailor them to data processing, e.g., through load balancing and skew detection at the switch level, data partitioning on the NIC, and content based routing. Although recent results [4, 25] have shown that smart NICs and programmable switches can improve the performance of distributed data processing systems, the hand-tuning of low level details remains a problem. Not only is the programming of the devices complex, it also creates resource management problems such as deciding when to offload computation into the network.

In this paper, we propose the Data Processing Interface (DPI) as a way to address these problems. DPI's goal is to make it easier for applications to exploit current and emerging capabilities of modern networks. Accordingly, DPI 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 an interface tailored to data processing, we adopt the approach taken by other high-level interfaces, such as MPI (Message Passing Interface) [13] and PGAS (Partitioned Global Address Space), which have been designed for other application domains and which, consequently, have seen only limited adoption for data processing [2]. Like MPI, DPI defines an interface that provides compatibility and portability across different networking technologies.

In brief, the main idea of DPI is that data movements are represented as *flows*. DPI 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 abstractions. By lifting the level of abstraction, DPI flows factor out much of the low-level complexity of network communication and make it easier for developers to declaratively express how data should be routed to accomplish a given distributed data processing task. Moreover, DPI flows allow developers to declaratively specify *optimization hints*; e.g., to maximize bandwidth-utilization or minimize network latency of transfers. That way, DPI can support a wide variety of applications ranging from bandwidth-sensitive distributed data analytics and machine learning to more latency-sensitive workloads such as distributed transactions or stream processing. Finally, flows can be *active*, meaning application-specified transformations can be applied to the data as it is in transit from source to destination. Thus, flows encapsulate both data movement and data processing. Ultimately, by exposing network flows and transformations explicitly, DPI will enable cost-aware optimization of applications' use of modern programmable networks, e.g., by deciding what functions to offload and when/where to offload them. In the rest of the paper, we present an initial sketch of DPI,

CIDR'19, January 13-16, 2019, Asilomar, California
.

and illustrate application opportunities through a set of potential data-intensive use cases.

2 PITFALLS OF EXISTING INTERFACES

Existing interfaces for programmable networks tend to be either too low-level to exploit data semantics or overly symmetrical and synchronous, limiting their applicability to data processing.

2.1 RDMA Verbs

RDMA provides access to a remote node's main memory without involving the CPU in the data transmission; RDMA is widely deployed at scale via InfiniBand or RDMA over Converged Ethernet (RoCE) [26, 29]. Applications use RDMA via the two types of operations found in the *verbs* interface. *One-sided verbs* (READ-/WRITE/atomics) provide remote memory access semantics and bypass the CPU of the remote node. *Two-sided verbs* (SEND/RECEIVE) provide messaging semantics and actively invoke the remote CPU on message receipt.

The two types of verbs have trade-offs. One-sided verbs have very low latency but often require multiple network round trips to implement complex operations. Two-sided verbs can be used to perform arbitrarily complex operations on multiple remote memory locations in a single round trip using an RPC-style approach [9, 17], but then logic must be executed by the remote CPU. The corresponding message handoff between the network card and the host CPU adds overhead, and the remote CPU must pay the cost of performing the operation. The suitability of each approach depends on whether an application is network or CPU bound [10].

RDMA verbs are fixed in hardware, therefore applications cannot specialize them (e.g., to support APPEND rather than just WRITE) to simplify algorithms or to offload work to network cards. Furthermore, RDMA verbs represent low-level operations requiring applications to choose connection types (reliable vs. unreliable), communication queue sizes, as well as manually handle remote buffer allocation and management. For example, a simple RDMA ping-pong requires over 1,000 lines of code.[1] Worse, implementation details intricately depend on these choices, making it difficult to evolve code as requirements change.

2.2 Other Network Interfaces

Some interfaces support higher-level abstractions for building distributed applications, and exploit RDMA as well. MPI [13] and the PGAS programming model are two prominent examples. MPI is widely used in HPC clusters; it has a network-independent interface similar to what we envision for DPI. MPI is primarily focused on message passing (`MPI_Send`, `MPI_Recv`), but extensions can make use of one-sided primitives (`MPI_Put`, `MPI_Get`) and collectives (`MPI_Reduce`). Mellanox's SHARP can push individual MPI collective operations to InfiniBand switches.[2]

DPI's approach is similar to MPI's, but it differs in several important ways. First, DPI raises the abstraction further, allowing applications to handle and accelerate the transfer of more irregular data, such as records on a table heap. Second, MPI assumes tightly coupled processes working mostly synchronously (e.g., workers processing a matrix partition in time step), whereas DPI works well for many concurrent but dissimilar flows of data between loosely coordinated processes (e.g., migrating shards of a distributed database and performing distributed joins). Third, MPI does not provide good mechanisms to support a dynamic sets of processes communicating with each other (e.g., to implement fault-tolerance and elasticity on the application level), which are two important properties for data processing solutions in cloud environments.

PGAS is sometimes used as an alternative to MPI to provide a shared memory abstraction over a cluster of nodes. PGAS hides the complex details such as connection setup and buffer allocation, but it has no visibility into data. This makes it hard to accelerate operations with in-network processing. Both PGAS and MPI have seen limited adoption for distributed data-intensive applications, in part due to these shortcomings [2].

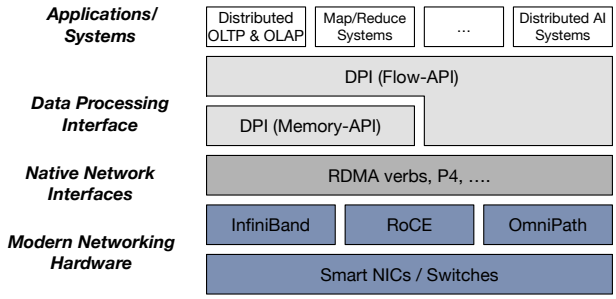


Figure 1: Positioning of the DPI APIs

2.3 Smart NICs and Switches

Pushing computation into the network today requires programming explicitly for each unique type of network devices. Network switches must be programmed with constrained match-action rules [5]; smart NICs have different programming models depending on the type of on-device compute they support (e.g., FPGAs, multi-core CPUs). There are a handful of proposed APIs for this type of offload including sPIN [14], FlexNIC [18], and Portals [12]. Programming languages for SDN such as P4 tend to focus on network operations. For instance, P4 has been used to performance load balancing on caching architectures through the introduction of small caches on the network switches [15]. DPI is orthogonal to such efforts given its focus on an application level API but it can benefit from these platforms as a way to implement a richer API.

3 THE DPI VISION

DPI is intended to simplify the development of distributed data processing systems by abstracting out and implementing a useful set of commonly needed network-centric operations. The aim of DPI is thus to provide abstractions to enable a better exploitation of capabilities in modern networks. Specific goals for DPI include (1) enabling simple and efficient use of RDMA by raising the level of abstraction and (2) to enable distributed data-intensive applications to

[1] See https://github.com/linux-rdma/rdma-core/blob/master/libibverbs/examples/rc_pingpong.c

[2] http://www.mellanox.com/blog/tag/sharp-scalable-hierarchical-aggregation-and-reduction-protocol/

transparently leverage in-network processing capabilities by pushing the DPI abstractions down into networking hardware. Since DPI defines only abstractions and interfaces, it can have multiple implementations for different network technologies (InfiniBand, Omni-Path, RoCE, ...) and implementations that use different types of smart network devices. DPI defines two interfaces, as we discuss next: a high-level *flow-based API* and a lower-level *memory-based API*, as shown in Figure 1.

Flow-based API. The flow-based API allows applications to define data movements of tuples across a network. Applications define different properties of the flow declaratively. Tuples may undergo application-defined manipulations *within* the flow. We describe the flow-based API in more detail in Section 4.

Our vision is that in-network processing should require only a minimal adaptation of the software stack. To this end, the flow-based API forces applications to expose important characteristics of flows, such as routing behavior and tuple manipulations. These characteristics can be further used for optimization and scheduling. In particular, given suitably capable NICs or switches, a DPI implementation can push tuple routing decisions, transformations, filters and aggregations into the network. Although we do not discuss it in this paper in detail, our vision for DPI also includes a cost-based optimizer for decisions such as which, whether, and when to push operations into the network.

Memory-based API. The memory-based API is a lower-level API that is intended to enable simple and effective use of RDMA. As shown in Figure 1, the memory-based API is used to implement the higher-level flow-based API. Moreover, it may also be used directly by applications for which flows do not completely satisfy application requirements (e.g., controlling tuple memory layout). The memory-based API allows applications to transfer raw data over the network and is thus closer to the level of abstraction provided by the native network interfaces such as RDMA verbs. However, unlike RDMA verbs, the memory-based API hides much of the complexity around connection establishment, memory management, etc. The memory-based API also provides more expressive operations like a remote `APPEND`.We describe the memory-based API in more detail in Section 5.

What is important to note is that DPI's main goal is to provide abstractions for efficient network communication that can be used to build data-intensive systems and abstractions on top. For example, flows can be used to implement not only shuffling operations for distributed joins but also other applications such as key/value-stores (e.g., to implement a parameter server) that require a bidirectional RPC-style communication. Some of these applications are discussed in more detail in Section 6.

4 DPI FLOWS

A DPI *flow* defines a coordinated migration of tuples from a set of senders (called *sources*) to a set of receivers (called *targets*). Each source and target represents a communication endpoint on a server in a distributed system.

4.1 Flow API Overview

Flows are intended to simplify the design and development of a broad class of distributed data-intensive applications, examples of which we discuss in Section 6. Specifically, DPI flows can help with the following aspects:

Data Routing: Distributed applications often move data between nodes, e.g., to bring matching tuples together when performing a distributed join. A DPI flow defines such a data movement, allowing the application to control how tuples are routed, filtered, and distributed.

Data Transformation: In addition to moving data, applications may wish to transform it. For example, flows can use a transformation that is defined as an encryption function which is applied before a tuple is injected (i.e., pushed) into a flow.

Synchronization and Coordination: A DPI flow defines whether and when tuple delivery is acknowledged to the source node(s). For example, a DPI flow can be used to distribute an update log from a master server to a set of slaves, while controlling when the master receives acknowledgements from slaves.

Network Optimization: Flows allow applications to provide hints of what the optimization goals of data transfers are. In a first version, applications can declare a flow either to optimize for latency or optimize for bandwidth utilization. This is important to support a wide variety of applications from more analytical workloads that are more bandwidth-sensitive to latency-sensitive workloads such as distributed transactions.

Resource Management: RDMA and other high-performance networking mechanisms place the burden of resource management (i.e., memory and connection handling) at the sources and targets to the application. DPI flows relieve the application of this responsibility.

Once a flow has been initialized, the application can use the API to inject *tuples* into the flow through the sources. DPI routes and transforms the injected tuple according to the flow definition, ultimately distributing the transformed tuple to one or more targets. For consuming tuples, each target defines a (pull-based) iterator, which the application can use to retrieve delivered tuples from the flow. Flows may be finite, eventually terminating, or they may be long-lived and potentially infinite (e.g. for stream processing).

4.2 Flow Characteristics

A DPI flow is defined with the following characteristics:

- $\mathcal{R}$ [mandatory]: The set of targets, specified at flow creation time and fixed for the duration of the flow.
- $\mathcal{S}$ [mandatory]: The set of sources, which may change over the duration of the flow.
- $F_{group}(t) \rightarrow k$ [mandatory]: The *grouping function*, a function that maps a tuple t to a grouping key k.
- $F_{dist}(k) \rightarrow R'$ [mandatory]: The *distribution function*, a function that maps a grouping key k to a set of targets $R' \subseteq R$.
- $F_{map}(t) \rightarrow t'$ [optional]: The *map function*, a transformation function that maps a tuple t into a new tuple t'.
- $F_{reduce}(T) \rightarrow t'$ [optional]: This function is a transformation that is applied to a list of tuples T with the same grouping key. The result is a new tuple t'.

- $F_{ack}(t) \rightarrow a$ [optional]: The *acknowledgement* function maps a tuple t to an acknowledgement key a.
- *Hints* [optional]: This allows applications to provide optimization hints (e.g., if flows should be latency- or bandwidth-optimized).

Together, F_{group} and F_{dist} define how items are routed from sources to targets by DPI. Conceptually, each injected item is first mapped to a DPI *grouping key* using F_{group}. Each grouping key is then mapped to a set of targets by F_{dist}. The injected tuple is delivered to each of these targets, i.e., tuple t is delivered to each target in $F_{dist}(F_{group}(t))$.

A DPI flow can be made to filter (groups of) tuples by defining F_{dist} to be empty for some grouping keys. Similarly, the flow will replicate items if F_{dist} is defined to be one-to-many. Tuples injected at a given source are delivered in injection order to each specific target determined by F_{dist} and F_{group}. DPI makes no guarantees about the delivery ordering of a source's tuples with respect to items that are injected at different sources.

Tuples can be modified by DPI as they flow from sources to targets using map and reduce-style transformation functions (F_{map} and F_{reduce}). These functions can be used to compose transformation pipelines that are executed at the sender before injecting a tuple or at the target after receiving a tuple. Depending on the nature of these functions, they can be pushed into a smart NIC at the sender / receiver node or even further into the network into a switch. Details about these functions are discussed next in §4.3.

Finally, a DPI flow can optionally be configured to provide delivery acknowledgements for tuples that have been injected into the flow. When acknowledgements are used, an acknowledgement key a is generated at each target for each tuple that it receives. These acknowledgement keys are returned to the originating source. Thus, acknowledgements can be viewed as forming a reverse flow of data, from targets to sources.

4.3 Data Transformations

The F_{map} transformation function takes one tuple as input and returns one tuple as output. Thus, mapping functions can be used to encrypt/decrypt tuples, for compression/decompression, for applying projections, or for other application-defined transformations.

In addition to F_{map}, F_{reduce} functions can be applied as a transformation function at the sender/target. As discussed before, F_{group} is used for both grouping and routing. More precisely, when tuple t is injected, DPI uses $F_{group}(t)$ to determine t's grouping key and F_{dist} to determine the destination. A reducer function is applied to all tuples with the same *grouping key* (given by $F_{group}(t)$). The F_{reduce} function can be used to specify aggregations on flows, e.g., to compute aggregates across tuples such as SUM or COUNT. Using F_{reduce} functions, applications like distributed aggregation or a parameter server can be implemented (see §6). Further extensions for window-based aggregations that are required for stream processing is an interesting avenue of future work.

To transform tuples, map and reduce functions can be composed into transformation pipelines. For each pipeline it must be defined where in the flow the pipeline is being applied: either before tuples are injected at the sender or after tuples are received at the target. Moreover, for each function in a pipeline, the application can specify whether the function can be pushed further down/upstream into the network (e.g. a programmable switch) or whether it must be executed at the sender/receiver node. For example, while a F_{map} for encryption/decryption has to be executed at the sender/receiver and should not be pushed into a switch, for an F_{reduce} function that aggregates tuples with the same grouping it might make sense to push it into a switch to avoid congestion in the network as shown in [4].

In a first version, DPI will provide a default set of mappers/reducers only; e.g., using standard aggregate functions such as SUM for the reducer.. A future research challenge is to allow user-defined mappers and reducers in DPI as we discuss in Section 7.

5 DPI MEMORY API

As shown in Figure 1, DPI's memory API provides a lower-level interface that is used to implement flows, and that can also be used directly by applications that require control of the memory layout of the transferred data. In the following, we first give an overview of the memory API and then briefly discuss its use.

5.1 Memory API Overview

Operations of the memory API allow applications to transfer raw data (rather than tuples) between source and targets. The operations are thus more similar to the semantics of native network APIs (e.g., RDMA verbs) which also allow transfer raw data over the network. However, the memory API simplifies connection and memory management, and provides applications with more expressive abstractions and operations for data processing beyond simple reads and writes to implement (potentially) long-running flows. The memory API provides the following key features:

Connection Handling: When using RDMA verbs, connections are hard to establish since the application must take care of many details involving the setup of queues for communication, as well as the orchestration of event handling to ensure queues don't completely fill (or in some cases, drain). The memory API hides this from the application. Instead, connections are established transparently using the cluster specification provided to DPI.

Memory Management: A similar observation holds for the memory management. When using RDMA verbs, applications have to explicitly manage local and remote memory regions that can be used for RDMA by registering them on the NIC and keeping track of which parts of the memory regions are already being used. In contrast, the memory API provides higher-level abstractions that hide this complexity. For example, the main abstraction provided by the memory API are remote buffers into which producers can easily append data, and from which consumers can consume data. Buffers can be accessed by a single or multiple producers as discussed below. Important is that the memory required for accessing buffers is transparently allocated/de-allocated by DPI.

One-to-one Operations: One-to-one operations can be used to transfer data between a pair of nodes. Examples for those operations include `DPI_Read` and `DPI_Write` operations to/from a remote buffer. These operations are similar to RDMA verbs but hide many of the complex details of memory management as mentioned before.

```
string buffer_name = "buffer";
char data1[] = "Hello ";
char data2[] = "World!";
int rcv_node_id = 0; //ID is mapped to a concrete node in cluster spec
DPI_Context context;
DPI_Init(context);
DPI_Create_Buffer(buffer_name, rcv_node_id, context);
DPI_Append(buffer_name, (void*)data1, sizeof(data1), context);
DPI_Flush_Buffer(buffer_name);
DPI_Append(buffer_name, (void*)data2, sizeof(data2), context);
DPI_Flush_Buffer(buffer_name);
```

Listing 1: DPI Append Example

One-to-many Operations: The operations in this category allow one source node to read/write data from/to one or multiple remote nodes. Examples for operations of this category are `DPI_Scatter` and `DPI_Gather` to append/read to/from multiple buffers.

Many-to-one Operations: The many-to-one operations allow multiple source nodes to access the same remote memory at one target node. In this category, the memory API provides calls such as a `DPI_Append` to allow multiple source nodes to append data to a remote buffer in one target node.

One important factor of the memory API is that it is just an abstraction and does not define how the data transfer operations (one-to-one, one-to-many, many-to-one) are actually implemented. To that end, the underlying DPI implementation can decide what the most optimal implementation of an operation of the memory-based API is (e.g. by using either one- or two-sided RDMA verbs or even other primitives if RDMA is not available).

5.2 Using the Memory API

Usage by Applications. In the following, we provide a short example of how the memory API's `DPI_Append` operation can be used in a distributed application. This example shows the behavior of `DPI_APPEND`, and also illustrates the level of abstraction that the memory API provides. Listing 1 shows how a source can use `DPI_Append` to append data to a remote memory buffer created in a target node. DPI's remote memory buffers significantly simplify remote memory management by keeping track of write offsets and managing (re-)allocation of memory (which would need to be done manually if RDMA verbs would be used). As a result, the remote append can be implemented in only a few lines of code as shown in Listing 1, while a similar application which uses RDMA verbs spans more than 1,000 of lines of code. Although it is not illustrated in this example, `DPI_Append` allows multiple sources to append data to the same remote buffer concurrently.

Implementing Flows. As previously noted, the memory API is being used to implement the core functions of DPI flows. For example, in our initial DPI prototype we are using remote memory buffers to move data between sources and targets of a flow. Depending on the distribution function specified by a flow, different data transfer operations (one-to-one, one-to-many, many-to-one) are being used to actually ship the data when tuples are being ingested into a flow.

6 POTENTIAL APPLICATIONS

In this section, we discuss how to use DPI to implement a variety of applications through three examples. However, there are many more not discussed in this section. For instance, stream processing is a clear candidate that would benefit from DPI. Furthermore, many of the applications below can be further enhanced by more advanced in-network processing capabilities such as sampling, conversion, type casting, normalization, etc. We will explore such use cases as part of future work.

6.1 Distributed Hash-Joins

A distributed hash-join can be implemented using DPI flows. Alternatively, the memory-based API could be used if more control is needed over tuple layout. We briefly discuss both options.

Two flows would be required to implement the join using the flow-based API, one flow for each input table. For both flows, the grouping key is the join key and the distribution function uses a hash function to map join keys to target identifiers. Targets use an iterator to consume the incoming tuples. That way pipelining at the targets can be done by consuming tuples as they arrive while adding them into a hash table for the inner table or probing into the hash table for the outer table. When using flows, the distributed DPI-based hash-join can also benefit from in-network processing in different ways. For example, a transformation can be defined that applies on-the-fly compression/decompression to reduce the memory footprint. Moreover, the join can also benefit from a cost-aware DPI flow optimizer in that uses SDN functionality to precisely load-balance traffic over available network routes between sources and targets. This can avoid congestion in the network that, for example, can result from attribute-value skew on the join keys.

As an alternative to the flow-based API, the memory-based API can also be used to implement a classical data shuffling operation. A direct way to do so is to use one remote memory buffer for each target node. Each source scans over its part of the input table, applies the hash-partitioning function to determine the target where the tuple should be routed, and adds the tuple to a local memory buffer for the target. Whenever a local buffer becomes full, `DPI_APPEND` can be used to move the local buffer to the target node. The join could benefit from in-network-processing if `DPI_APPEND` is implemented directly within the network card instead of using multiple RDMA verbs (e.g., an atomic operation and a RDMA WRITE) for each append call.

6.2 Parameter Servers

The parameter server architecture [19] has become a popular way to structure large-scale distributed machine learning systems. The parameter server architecture distinguishes *servers*, which maintain the model being learned, and *workers*, which use training data to compute model updates. At a very high level, servers distribute the model to the workers, the workers compute model changes and send them back to the servers, and the servers update the model by aggregating the workers' changes. This process iterates until the model converges.

DPI flows can be used to implement the parameter server architecture. Large models are normally partitioned across servers. A one-to-many DPI *model flow*, with servers as sources and workers as targets, can allow the servers to jointly broadcast the current model to the workers on each iteration. A second *update flow*, in the reverse direction, can be used to move model updates from the workers to the servers. By using the model parameters as grouping keys in the update flow, the workers can ensure that each parameters' updates are routed to the server responsible for storing it.

By defining an appropriate reducing function on the update flow, aggregation of parameter updates can be pushed off of the servers and into the network, potentially allowing the servers to support larger numbers of workers.

6.3 Log Shipping

DPI flows work well for multi-producer primary/backup log replication. Primaries act as DPI sources, backups act as DPI targets. A primary can multicast log records to multiple backups by configuring F_{dist} and F_{group}. This can even be driven by the content of the log records, for example to partition log records into different physical buffers at the targets based on an application-level key. Transformations can also offload some replication overhead to the network. For example, before and after F_{map} functions can serialize and deserialize log records.

DPI's control over acknowledgements lets primaries inform clients when updates have been safely replicated. For example, a primary can enforce quorum replication by configuring acknowledgement aggregation and waiting for the aggregated acknowledgement. One challenge in DPI is defining a fault-tolerance model that is useful and practical; exposing faults to DPI applications is complicated and fragile, but masking faults can force DPI to include costly recovery mechanisms, which is beyond what DPI should provide. As a result, our thoughts on fault tolerance in DPI are preliminary until we have enough experience with it to strike the right balance.

7 RESEARCH CHALLENGES

DPI opens up many possibilities and enables the efficient use of modern networks by a wide-range of applications. Yet, many research challenges remain open. We now comment on some of these challenges for the database research community and outline initial ideas on how to address them.

Network Monitoring and Optimization: Since DPI is aware of network flows, a DPI implementation can provide a component that collects flow statistics. This opens up new avenues to build a cost-aware optimizer that uses the capabilities of modern programmable networks (i.e., SDNs) to better load-balance all traffic across the available network routes from senders to receivers. Moreover, a cost-aware optimizer could also make decisions about which DPI transformation functions are pushed into the network in order to avoid overloading network devices. We believe that this not only lowers contention in the network and thus prevents typical shortages like network incast, but also leads to a better utilization of networking resources.

Moreover, in data centers, different applications share the same network infrastructure. To that end, another challenge is to develop strategies that can make optimal decisions in the presence of multiple data-intensive applications with different workloads and potentially conflicting requirements. Implementing network monitoring and cost-aware optimizers for different network devices and transformation functions is another research challenge for future work.

Fault-tolerance and Elasticity: Unlike MPI, DPI is assumed to support data processing applications not only in on-premises clusters designed for static deployments but also in cloud environments. In the cloud, data processing applications often need to be able to efficiently provide elasticity and deal with failures. For DPI flows, we thus envision that they can transparently deal with node failures or with situations where nodes join/leave a flow. For example, if a flow could survive the failure of individual sources and targets, then it would not be necessary to handle flow-level exceptions or to establish new flows after failovers.

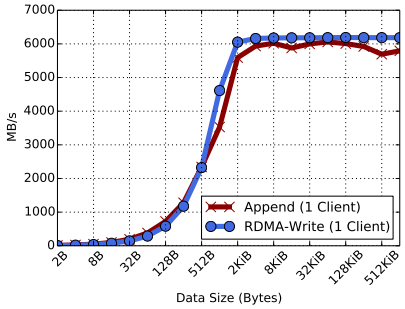
One idea is to leverage the fact that *logical* targets are separated from the *physical* targets of a flow. Flows are defined in terms of a set of logical targets ($\mathcal{R}$) drawn from the range of F_{dist}. Logical targets are bound to specific physical nodes when a flow is established. To handle failures or reconfigurations (e.g., for elasticity), a logical target could be bound to a new physical target without disrupting the rest of the flow or different logical targets that were mapped to the same physical node can be split up. For example, for replicated databases using log shipping, a backup that takes over after a primary failure could unbind itself as a flow target and rejoin as a source. A new backup could be bound as a target to take over the new primary's backup role. It will likely make sense to leave state transfer during reconfigurations in the hands of applications (e.g., reshuffling hash partitioned log records on group size change), to avoid the need for costly built-in fault-tolerance mechanisms.

Moreover, DPI allows data transformation functions to be pushed into network components (e.g., switches). If these transformation functions hold state, then providing fault-tolerance for flows becomes non-trivial. Recovering from failures of network components thus requires either that lineage needs to be kept in DPI to recompute the "lost" state or that state is replicated across multiple network devices.

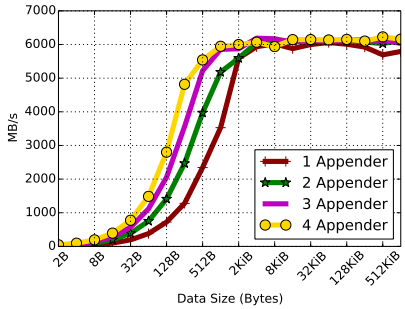
User-defined In-Network Processing: As discussed before, DPI flows allow users to define data transformation functions (e.g., reduce functions such as SUM and COUNT). These functions can either be executed by the CPU of the source/target but, more importantly, can also be pushed *into* network devices (e.g., into a smart NIC or a switch). While we currently only support a small set of pre-defined transformation functions, in the longer term, we envision that users can also implement data transformations as user-defined transformation functions.

To allow user-defined functions to be pushed into network devices, we want to provide a higher-level language that allows applications to implement map- and reduce-based transformations that can automatically be compiled into programs for in-network-processing (e.g., in P4 or other available languages as discussed in Section 2.2). Defining such a higher-level language that suits data-processing needs and can efficiently be compiled into different network components is a research challenge for future work.

Applications and Benchmarks: Finally, we believe that DPI will only see a high adoption if a wide range of data processing applications can be efficiently implemented on top of it. Therefore, we aim to open-source a first implementation of DPI as soon as possible so that different groups in research and industry can leverage DPI to build applications but also benchmark DPI.



(a) DPI vs. Native RDMA



(b) DPI: Multiple Appenders

Figure 2: Throughput of DPI Append

8 EARLY RESULTS

As a proof-of-concept, we have implemented a first version of DPI's memory-based API for InfiniBand and RDMA to illustrate (1) the complexity reduction that can be achieved by using DPI and (2) that the abstractions do not lead to a decreased performance when compared to the use of native interfaces such as RDMA verbs. The code of the initial DPI version is available at https://github.com/DataManagementLab/dpi_library. While the initial prototype does not yet push functions into network devices, it enables applications to make use of RDMA in the way illustrated in Listing 1. As discussed before, this example already shows that a significant reduction of complexity from more than 1,000 of lines of native RDMA code to a few lines of DPI code.

To show that the efficiency of DPI is comparable to native RDMA verbs, despite the added abstraction, we performed a small benchmark in which `DPI_APPEND` was used by multiple DPI nodes to append concurrently to a single remote memory buffer. For the experiment, we used two machines connected by an InfiniBand FDR 4× network. On the first machine we were running the senders (using different threads) and on the second machine we were running one receiver where the remote memory buffer was located.

The results are shown in Figure 2. The graph shows the throughput of `DPI_APPEND` achieved when appending different sized data chunks to a remote buffer (as indicated on the x-axis). As a main observation, we can see that `DPI_APPEND` can leverage the full bandwidth of the InfiniBand FDR 4× network used in this experiment (see Figure 2a). This indicates that using DPI can be as efficient as using plain RDMA verbs while raising the level of abstraction at the same time. Furthermore, using multiple appenders concurrently does not affect the throughput in a negative way (see Figure 2b) showing the efficiency of DPI's concurrency handling scheme.

9 CONCLUSIONS

We have presented our vision for DPI, a way to better exploit modern networks, including RDMA and in-network processing. DPI should foster a new line of research and serve as the basis for a standard interface for distributed data-intensive applications.

ACKNOWLEDGMENTS

We thank Schloss Dagstuhl and the organizers of Seminar 18251 for the opportunity and motivation to pursue the work reported in this paper. T. Ziegler and J. Skrzypczak were partially funded by the German Research Foundation (DFG) under grants BI2011/1 and RE1389/10 (DFG priority program SPP 2037).

REFERENCES

[1] C. Barthels et al. Rack-scale in-memory join processing using RDMA. In *ACM SIGMOD*, pages 1463–1475, 2015.
[2] C. Barthels et al. Distributed join algorithms on thousands of cores. *PVLDB*, 10(5):517–528, 2017.
[3] C. Binnig et al. The end of slow networks: It's time for a redesign. *PVLDB*, 9(7):528–539, 2016.
[4] M. Blöcher et al. Boosting scalable data analytics with modern programmable networks. In *ACM DaMoN@SIGMOD*, pages 1:1–1:3. ACM, 2018.
[5] P. Bosshart et al. P4: programming protocol-independent packet processors. *Computer Communication Review*, 44(3):87–95, 2014.
[6] H. Chen and other. Fast in-memory transaction processing using RDMA and HTM. *ACM Trans. Comput. Syst.*, 35(1):3:1–3:37, 2017.
[7] A. Devulapalli et al. Distributed queue-based locking using advanced network features. In *ICPP*, pages 408–415, 2005.
[8] A. Dragojević et al. FaRM: Fast remote memory. In *NSDI*, pages 401–414, 2014.
[9] A. Dragojević et al. No compromises: distributed transactions with consistency, availability, and performance. In *OSDI*, pages 54–70, 2015.
[10] A. Dragojevic et al. RDMA reads: To use or not to use? *IEEE Data Eng. Bull.*, 40(1):3–14, 2017.
[11] D. Firestone et al. Azure accelerated networking: Smartnics in the public cloud. In *NSDI*, pages 51–66, 2018.
[12] D. S. Greenberg et al. A system software architecture for high end computing. In *ACM/IEEE SC*, page 53, 1997.
[13] W. Gropp et al. *Using Advanced MPI: Modern Features of the Message-Passing Interface*. The MIT Press, 2014.
[14] T. Hoefler et al. sPIN: High-performance streaming processing in the network. In *ACM/IEEE SC*, pages 59:1–59:16. ACM, 2017.
[15] X. Jin et al. Netcache: Balancing key-value stores with fast in-network caching. In *SOSP*, pages 121–136, 2017.
[16] A. Kalia et al. Using rdma efficiently for key-value services. In *Proc. of ACM SIGCOMM*, pages 295–306, 2014.
[17] A. Kalia et al. FaSST: fast, scalable and simple distributed transactions with two-sided (RDMA) datagram RPCs. In *Proc. of OSDI*, pages 185–201, 2016.
[18] A. Kaufmann et al. High performance packet processing with FlexNIC. In *ACM ASPLOS*, pages 67–81. ACM, 2016.
[19] M. Li et al. Scaling distributed machine learning with the parameter server. In *USENIX OSDI*, pages 583–598, 2014.
[20] S. Loesing et al. On the Design and Scalability of Distributed Shared-Data Databases. In *ACM SIGMOD*, pages 663–676, 2015.
[21] C. Mitchell et al. Using One-Sided RDMA Reads to Build a Fast, CPU-Efficient Key-Value Store. In *Proc. of USENIX ATC*, pages 103–114, 2013.
[22] S. Narravula et al. High performance distributed lock management services using network-based remote atomic operations. In *IEEE CCGrid*, pages 583–590, 2007.
[23] J. Ousterhout et al. The case for RAMCloud. *Communications of the ACM*, 54(7):121–130, 2011.
[24] R. Power et al. Piccolo: Building Fast, Distributed Programs with Partitioned Tables. In *OSDI*, pages 293–306, 2010.
[25] A. Sapio et al. DAIET: a system for data aggregation inside the network. In *SoCC*, page 626. ACM, 2017.
[26] J. Vienne et al. Performance analysis and evaluation of InfiniBand FDR and 40GigE RoCE on HPC and cloud comp. systems. In *IEEE HOTI*, 2012.
[27] D. Y. Yoon et al. Distributed lock management with RDMA: decentralization without starvation. In *ACM SIGMOD*, pages 1571–1586, 2018.
[28] E. Zamanian et al. The end of a myth: Distributed transaction can scale. *PVLDB*, 10(6):685–696, 2017.
[29] Y. Zhu et al. Congestion Control for Large-Scale RDMA Deployments. In *SIGCOMM*, pages 523–536, 2015.