DaRPC: Data Center RPC

Patrick Stuedi, Animesh Trivedi, Bernard Metzler, and Jonas Pfefferle
IBM Research
{stu, atr, bmt, jpf}@zurich.ibm.com

Abstract

Remote Procedure Call (RPC) has been the cornerstone of distributed systems since the early 80s. Recently, new classes of large-scale distributed systems running in data centers are posing extra challenges for RPC systems in terms of scaling and latency. We find that existing RPC systems make very poor usage of resources (CPU, memory, network) and are not ready to handle these upcoming workloads.

In this paper we present DaRPC, an RPC framework which uses RDMA to implement a tight integration between RPC message processing and network processing in user space. DaRPC efficiently distributes computation, network resources and RPC resources across cores and memory to achieve a high aggregate throughput (2-3M ops/sec) at a very low per-request latency (10μs with iWARP). In the evaluation we show that DaRPC can boost the RPC performance of existing distributed systems in the cloud by more than an order of magnitude for both throughput and latency.

Categories and Subject Descriptors D.4.4 [*Operating Systems*]: Communications Management – Network Communication; D.4.4 [*Operating Systems*]: Organization and Design – Distributed Systems

General Terms Design, Performance, Measurements

Keywords Remote Procedure Call, RDMA

1. Introduction

Remote Procedure Call (RPC) has been a key building block of distributed systems ever since the early 80s [5]. However, the emergence of new classes of large-scale distributed systems running in data centers has added extra pressure on RPC systems. For instance, systems like HDFS, Zookeeper or OpenFlow contain centralized RPC services (e.g., namenode, scheduler, controller) that are asked to process high volumes of RPC requests per second. Moreover, some recent systems offering low-latency data access, like RAMCloud [16] or Tango [3], demand ultra-low RPC latencies while still requiring the RPC system to scale to high volumes of requests.

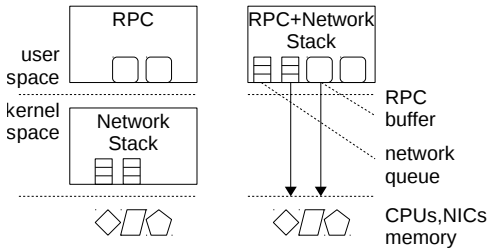

Figure 1. (left) Traditional separation of networking and RPC processing, (right) joint optimization of RPC processing and networking in DaRPC.

Unfortunately, current RPC implementations used in today's cloud-based systems have difficulties meeting these requirements. For instance, RPC services in HDFS and Zookeeper can typically process 100-200K operations per second at latencies between 200 and 500μs. Other systems like the one used in Tango are performing better (600K ops/sec at 60-70μs), but in all these cases the performance of the RPC system is considerably below what the hardware (CPUs, network) can deliver. In fact, we observe that these systems neither manage to saturate the network nor the CPU. Recently, similar inefficiencies have been discussed at the level of the network stack. To overcome these problems it has been suggested to implement the network stack in user space with reduced overhead [9].

In this work, we take this idea one step further and integrate RPC processing with network processing in user space by using Remote Direct Memory Access (RDMA). We present DaRPC, a high-throughput low-latency RPC framework tailored to improve the performance of large-scale distributed systems in data centers. The key idea used in DaRPC is to coordinate the distribution of compute and

SoCC '14, 3-5 Nov. 2014, Seattle, Washington, USA.
ACM 978-1-4503-3252-1.
http://dx.doi.org/10.1145/2670979.2670994

memory resources for both RPC processing and networking within a many-core system. This approach is in contrast to the classical way RPC systems are built, where RPC message processing is implemented in isolation from the network processing in the kernel (see Figure 1). By looking at RPC processing and networking as a joint optimization problem we can avoid context switches, cache misses and achieve a high degree of parallelism as well as ultra-low latency.

Aside from its scalability and latency characteristics, DaRPC also provides an application interface which makes it easy to use for existing cloud based distributed systems for multiple reasons. First, DaRPC is implemented entirely in Java and is thereby pushing its performance advantages directly into the managed runtime of systems like Hadoop, Zookeeper or Spark. Second, DaRPC provides a purely asynchronous programming interface, allowing applications to handle RPC operations with very few context switches and minimal cache pollution. Internally, DaRPC takes advantage of the asynchronous nature of the RDMA network stack to map RPC operations to network operations with the utmost efficiency.

In the paper, we evaluate DaRPC on a 17-node cluster, both as a standalone system and in the context of an existing distributed system like HDFS. We demonstrate that on a single server system, DaRPC can process up to 2.4M RPC req/sec with a latency as low as 10μs on Ethernet-based RDMA adapters. This is an order of magnitude better than the performance of some of the RPC systems used in practice today. We further show that similar throughput gains (with some latency penalties) can be achieved in a configuration with standard Ethernet adapters (without RDMA support) used at the clients. This is appealing considering that data centers are mostly built from commodity equipment.

In summary, this paper's contributions include (1) an analysis of the scalability limits and latency overheads of RPC systems used today (2) the design of DaRPC, which demonstrates how RPC and RDMA network resources can be jointly optimized to maximize throughput and minimize latency of RPC request processing, and (3) the demonstration of how to use DaRPC in a real cloud workload.

2. Motivation

We use HDFS and Zookeeper to highlight the shortcomings of some of the built-in RPC systems available in today's cloud services. The RPC performance of these systems has been looked at before, but these studies are all based on deployments on top of a Gigabit Ethernet network [2, 22]. While Gigabit Ethernet is still the pre-dominant network technology used in data centers today, 10Gbit/s Ethernet is becoming increasingly popular. In a deployment using Gigabit Ethernet, the performance of an RPC service may be limited by the network bandwidth. In this work we are interested in investigating the full potential of an RPC system. Therefore, we ran a series of tests on a 17-node testbed composed of dual-socket 8-core Intel Xeon servers interconnected by a 10 Gbit/s Ethernet network. Figure 2 shows throughput and latency of `getBlockLocation()` operations in HDFS and `exist()` operations in Zookeeper. In terms of throughput both Zookeeper and HDFS namenode scale up to 100-130K ops/sec, until a saturation point is reached. At the same time, the average latency per client increases constantly from initially 200μs up to 500μs.

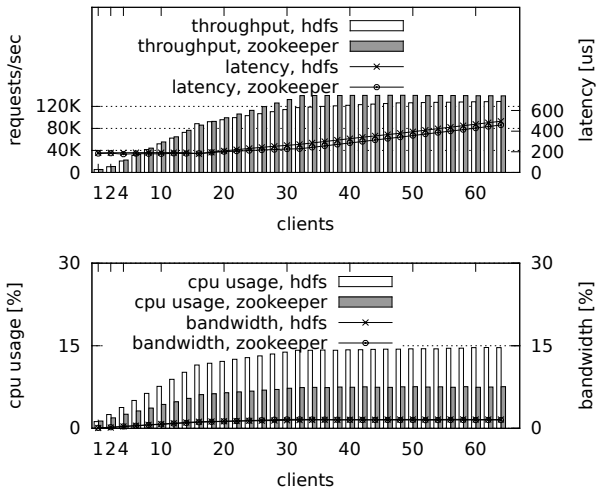

Figure 2. (top) Throughput and latency of RPC operations in HDFS and Zookeeper, (bottom) bandwidth and CPU consumption of RPC operations in HDFS and Zookeeper.

Interestingly, these numbers are in the same ballpark (considering the read-only workload for Zookeeper) as the performance numbers measured on a Gigabit network deployment with less powerful server machines [2, 22]. From that, one can assume the performance not being limited by either the network bandwidth or the CPU consumption at the server. This is also confirmed by our measurements, showing a maximum of 1.6% of the bandwidth being used with the CPU load never exceeding 15% (1 core = 100 %).

Given that both the CPU and the network at the server are not fully used, inefficiencies in I/O processing are a potential remaining bottleneck. Examples of I/O inefficiencies are cache misses, excessive interrupt processing or uneven load balancing among server threads. We found that for both the Zookeeper and the HDFS scenario the cache miss rate was between 25-30% and around 2 context switches were executed per RPC operation. Later in the paper, we show that DaRPC operates with very few context switches and also uses fewer cache misses, both effects leading to higher RPC throughput and lower latencies.

3. Background

At this point, we want to briefly give some background on RDMA networking. DaRPC uses jVerbs [23], a low-latency

RDMA stack for the Java Virtual Machine[1]. jVerbs exposes RDMA network hardware resources directly to the JVM through a standardized interface and provides Java processes with the well-known RDMA features like kernel bypass and zero-copy network I/O when transferring data between two networked JVMs.

jVerbs provides a rich set of operations of which `post_send()/post_recv()` operations are most relevant for RPC messaging. With these operations, the sender sends a message, while the receiver pre-posts an application buffer, indicating where it wants to receive data.

Applications use jVerbs to create hardware resources on the network interface such as send, receive and completion queues. For each physical queue on the NIC, a queue object in the host's I/O memory is allocated which is mapped into the JVM and can be accessed directly by the application using jVerbs. Applications asynchronously post requests for data transfers into the send/receive queues. A data transfer request (also sometimes called work-request) identifies the type of operation (e.g., send or receive) and points to a memory buffer that can be accessed from within the JVM. The RDMA NIC processes requests in send/recv queues (often referred to as "queue pair" (QP)) and places a completion notification into the completion queue once the operation has finished. A completion queue (CQ) can be shared by multiple queue pairs, and it can be queried by the application in either polling mode or in blocking mode. During data transfer, the user memory referenced by data requests is accessed by the RDMA NIC directly via DMA in a zero-copy fashion. For this to work safely, application data buffers are required to be registered with the RDMA subsystem using the `reg_mr` API call available in jVerbs. By doing so, the RDMA subsystem will make sure the memory is pinned and cannot be swapped out. In addition, it is required that all buffers used for jVerbs data operations are of type "direct" buffer. Java provides explicit API calls to allocate "direct" buffers in a dedicated area of the heap that is not managed by the garbage collector. This is essential for jVerbs since we cannot have the garbage collector interfering with RDMA data operations.

A special feature of jVerbs are Stateful Verb Calls (SVCs). SVCs eliminate any overhead that occurs during repetitive data operations. A typical data operation like `post_send()` or `post_recv()` requires the application to prepare an array of work requests which in turn will have to be serialized by jVerbs when placing the work request into the proper queue. With SVCs, the entire state of a jVerbs operation down to the serialized work request can be cached by the application and re-used on repetitive operations. This reduces both the CPU consumption as well as the latency of jVerbs RDMA operations.

[1] jVerbs is available as part of the IBM SDK, Java Technology Edition, Version 7

4. Design of DaRPC

The ultimate goal of this work is to create a RPC system which fulfills the following three requirements:

1. *High throughput:* The system should be able to process large volumes of RPC requests per second. This property is key in the context of large-scale distributed systems operating with centralized components (e.g., locking service, OpenFlow controller, etc.). A single RPC service may have to process millions of RPC requests coming from thousands of servers.
2. *Low latency:* The latencies of individual RPC operations should be close to the raw network latencies, even in times where the RPC server is loaded. This property is essential in cases where a RPC operation is part of a series of serialized operations that need to be completed in a short time frame (e.g., RPC for metadata lookup in a storage system).
3. *Cloud integration:* The system should be easy to use for cloud-based distributed systems.

With DaRPC, we meet the first two requirements by jointly optimizing RPC processing and RDMA networking in user space. And we meet the last requirement by offering a powerful programming interface that provides zero-copy RPC from and to application buffers inside a JVM. We will describe the application programming interface of DaRPC in section 5. Here, we first focus on the key components in the design of DaRPC. We start off by illustrating the basic interplay of RDMA and RPC processing in a single client-server scenario. In a second step we look at how to scale the server-side of DaRPC to efficiently process large volumes of RPC requests. And finally, we describe the principles used at the client-side to sustain low RPC latencies even under concurrent accesses.

4.1 Single Client-Server RPC

Figure 3 illustrates the basic steps involved in a simple RPC operation in DaRPC between a single client and a server. First, the client marshals input parameters and potential protocol data into a pre-registered RPC buffer and triggers a `post_send()` to transmit a message to the server. At the server, the message is DMAed into a RPC receiving buffer by the NIC and a work completion is placed into the server's completion queue. Upon detecting the work completion event, the server unmarshals the message, executes the RPC call, marshals the response value into an outgoing RPC buffer and triggers a `post_send()` on its own in order to transmit the response message. Once the client detects a new work-completion event it knows the response message has arrived and has been DMAed into its user space RPC buffer. The client then de-marshalls the response value and completes the RPC call.

In DaRPC, we take advantage of jVerbs's stateful verb calls (SVCs) to avoid any overhead in serializing work re-

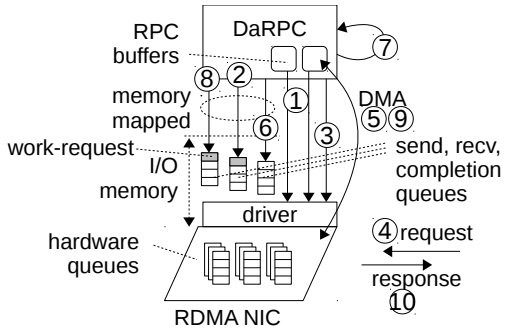

Figure 3. A single-client RPC operation in DaRPC (server-side only): (1) registration of RPC buffers, (2) posting of receive operation, (3) blocking on CQ event, (4+5) incoming RPC request, (6) polling of work completion, (7) execution of RPC service, (8+9+10) transmission of response.

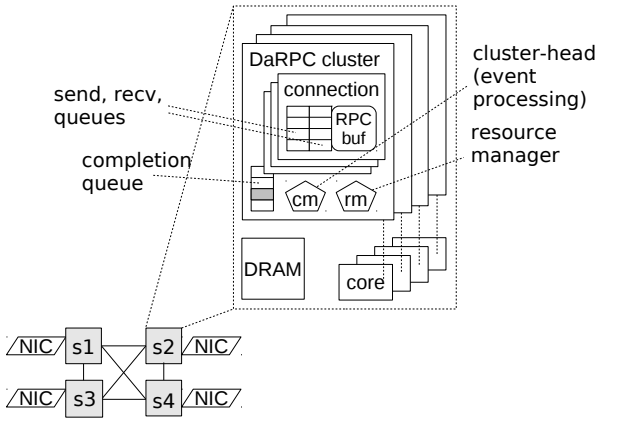

Figure 4. DaRPC architecture illustrated on a 4-socket (s1-s4) NUMA machine.

quests when transmitting RPC requests or responses. Specifically, for each network buffer used for sending and receiving RPC data, DaRPC keeps a pre-serialized SVC object around which encodes the given operation and can be placed into a RDMA send or receive queue immediately.

In such a single client-server RPC message exchange, the RDMA advantages of zero-copy network I/O and kernel-bypass naturally translate into low RPC response times. For instance, DaRPC provides latencies of 10μs for RPC operations in an isolated client-server scenario. This is just 1μs above the raw RDMA send/recv latency we measured on the same deployment.

4.2 Server-Side Processing

With regard to the server side, the key challenge is to parallelize RPC request processing in a many-core system while avoiding I/O inefficiencies such as context switches, locking or cache misses.

In the following, by modifying the server side of the basic RPC mechanism described in the previous section, we present several design options and discuss their advantages and limitations. We illustrate the effect on RPC throughput for each option in Figure 5. The RPC benchmark we use along with this section is implementing a "null" operation with no application level code being executed at the server. All experiments are run on our 17-node cluster described in Section 2.

Simple completion queue scaling: A simple way of extending the single-client setup to multiple clients is to provision a separate completion queue for each connection accepted by the server. Separate server threads can be used to query these completion queues, process RPC messages and transmit responses back to the clients. Similar approaches have been implemented with regular socket based network stacks and are known to scale badly due to heavy context switching between the server threads [17, 27]. In addition, the approach imposes high memory consumption due to replicated resources in server threads.

Our measurements (Figure 5) show that by using simple completion queue scaling, DaRPC can serve 16 clients firing an aggregated volume of 480K RPC requests per second. Although this is more than a factor 3 better than what we have seen previously in HDFS and Zookeeper, adding more clients caused contention at the server and performance degraded quickly.

Completion queue sharing: Contention among large numbers of server threads can be avoided by having individual client connections share a single completion queue at the server. A single server thread can be used to query the completion queue and dispatch events to a bounded pool of processing threads. Such an approach resembles the architecture of scalable web systems where a load balancer dispatches requests to different nodes in the cluster. As can be seen from Figure 5, compared to simple completion scaling, completion queue sharing improves the RPC throughput at the server by another factor of 3.

Completion queue clustering: One problem with completion queue sharing is that a single completion queue can become the bottleneck very quickly. First, the size of a completion queue is limited, therefore it might fill up under a heavy load. Second, a completion queue can only be processed by a single dispatcher thread (only one thread can enter the blocking call), potentially making that thread the bottleneck. And third, the approach adds extra context switches as worker threads need to be scheduled after event dispatching. To mitigate these problems in DaRPC, we use completion queue clustering. In this approach, a pool of completion queues is created with each queue being shared by a subset of the accepted connections at the server. We refer to a group of connections associated with a single completion queue to as a completion queue cluster.

Each cluster contains a cluster-head taking care of the CQ event processing. Essentially, a cluster-head processes

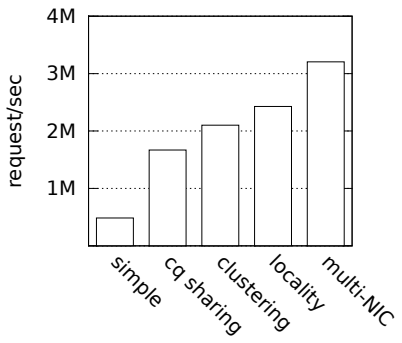

Figure 5. Effect of different design decisions and configurations in DaRPC.

RPC requests, as illustrated in Figure 3, on behalf of a set of connections. New connections get assigned to a dedicated cluster at connection acceptance time. DaRPC constantly monitors the load of each cluster based on the CQ polling history and assign connections accordingly. The architecture of DaRPC using clustering is illustrated in Figure 4.

The clustering approach has two advantages. First, clusters can be assigned to dedicated cores in order to distribute the load. And second, RPC requests can now be processed in-place in the context of the cluster-head rather than dispatching the requests to separate worker threads. This saves a context switch and reduces the CPU load. In our testbed we configured DaRPC to use a pool of 4 completion queue clusters. As can be seen from Figure 5, with such a configuration we managed to process up to 2M RPC req/sec.

Work stealing and load balancing: Maintaining an equal number of connections per cluster may not lead to a uniform load distribution among the cores of a RPC server. In fact, some clusters might experience a higher load than others (due to loaded connections) even though the cluster contains fewer connections. To mitigate this problem, we continuously monitor the load of each cluster and always assign new connections to the least loaded cluster. Here, the history of polling cycles serves as a good indication of the load of a cluster. Typically, a cluster-head – after having processed a RPC request – will re-poll the completion queue and keep processing as long as outstanding requests are found, before eventually entering the blocking mode again. A cluster can be considered loaded if its cluster-head hardly ever falls back to blocking mode. We therefore use the number of consecutive successful polls on the completion queue as a measure of the load of a cluster.

Unfortunately, the load distribution among clusters may change dynamically even in cases where no connections are added or removed, simply because the load of the connections changes over time. In DaRPC we use work-stealing to protect against short term load spikes in clusters. A cluster-head, if its load (based on the metric just discussed) exceeds a certain threshold, will place some of the incoming RPC requests into a global queue which is accessible by all cluster-heads in the system. At the same time, other cluster-heads poll and potentially process the global queue every time before polling their own completion queue. This approach makes sure that spikes in one cluster are consumed by other less loaded clusters.

Memory locality: Modern many-core systems typically provide NUMA-like memory access where the memory access time depends on the memory location relative to the processor core. Even though all cores can access all the memory, access to local memory is generally faster than access to remote memory. Therefore, we want to make sure that all memory resources used within a cluster are residing in the local memory of the core running the cluster-head. Examples of memory resources are queue pairs, completion queues, work completion events or RPC message buffers. In DaRPC, NUMA-like data locality is implemented as follows. First, each cluster is assigned a resource manager which is pinned to the same core as the cluster-head. Second, memory resources requested by cluster-heads and connections will be allocated through the cluster's own resource manager. Typically, operating systems allocate memory local to the requesting thread, in which case all cluster resources are allocated local to the core executing the cluster-head event loop.

Figure 5 shows the throughput performance of DaRPC in a configuration with data locality enabled. As can be seen from Figure 5, such a configuration improves the RPC throughput by another 300K req/sec.

Multiple NICs: Server systems often comprise multiple network interfaces. DaRPC is designed to run on multiple interfaces through the use of instances. A DaRPC instance refers to a pool of completion queue clusters which is bound to a specific network interface. RPC applications request a DaRPC instance to be created on a given pool of CPU cores for a given network interface. Multiple DaRPC instances can be created for multiple network interfaces. A typical approach for a RPC service is to partition the available cores and use each partition to create a separate DaRPC instance. Client requests can be load balanced over the different instances either using DNS or ARP.

We ran tests with two DaRPC instances. Each instance is composed of 4 clusters and all clusters are scheduled on separate cores. The first instance is bound to a Chelsio T5 NIC, the second instance is bound to a Chelsio T4 NIC. This configuration delivers up to 3.2M RPC req/sec. This number almost exactly matches the sum of the RPC throughput performance we have seen in isolated experiments using just one NIC. For instance, with a single DaRPC instance attached to the less powerful T4 NIC we were able to drive 790K RPC req/sec.

Hardware Limits: In our experiments, the Chelsio T5 NIC delivers 2.4M RPC requests/sec. To put this number in

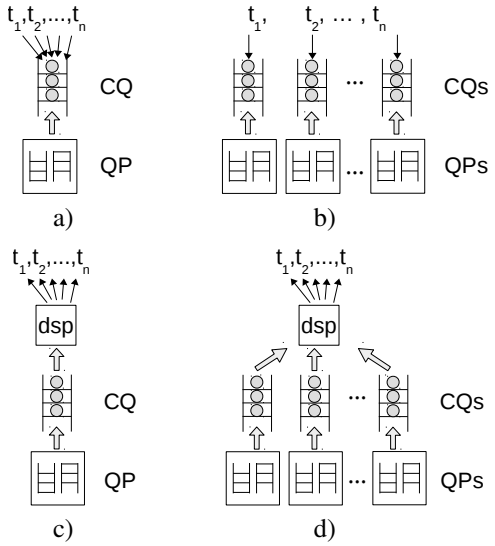

Figure 6. Different modes for client-side CQ processing in configurations with multiple application threads. (a) direct CQ access with a single connection, (b) direct CQ access with multiple connection, (c) CQ dispatching with a single connection, (d) CQ dispatching with multiple connections. CQ: completion queue, QP: queue pair, dsp: dispatcher, (t1-tn): application threads.

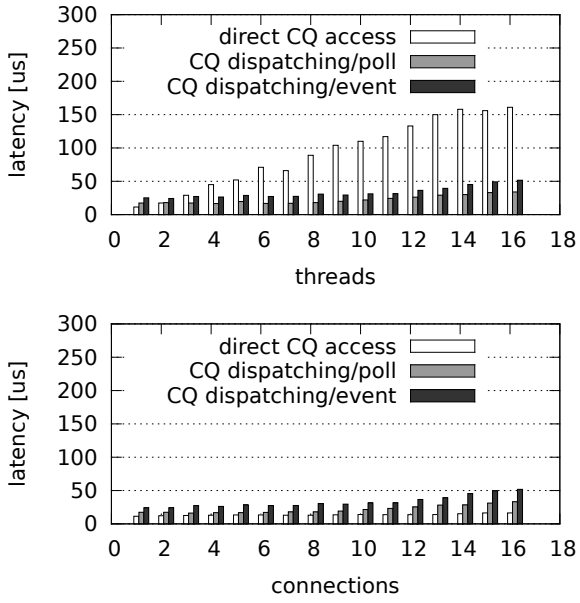

Figure 7. RPC Latency with different client-side CQ processing modes. (top) multiple threads on a single connection (bottom) multiple threads with separate connections.

perspective consider the raw packet rate of 14.8M packets/sec that a 10Gbits/s NIC should support[2]. However, in order to sustain this packet rate modern NICs deliver packets in bursts or batches to amortize the cost of DMA, PCIe transactions, interrupts, and CPU processing etc. MICA [12] uses packet burst size of 32 (RouteBrick [6] also recommends a similar batch size) to sustain packet rates of $8.2-9.6$M packets/sec per NIC. In comparison, T5's performance of 2.4M requests/sec is without any batching. With a modest batch size of 8 (see Section 6.1), DaRPC delivers 8.8M req/sec. Additionally, DaRPC's performance is further governed by the specific implementation of the RDMA NIC. As we have observed earlier, different RNIC generations (e.g., between T4 and T5) have different RDMA processing capacities.

4.3 Client-Side Processing

While at the server-side DaRPC is exclusively processing events from the network, at the client-side actions are triggered both from the network as well as from the application. Again, one key challenge is to be as efficient as possible with regard to context switches, locking, etc. Moreover, special attention is required to handle concurrent accesses from different application threads. We identify two possible design options for the client-side of DaRPC.

Direct CQ access: With this option, each RPC connection will receive its own separate completion queue at the client end of the connection (not to be confused with the server end where the connection is part of a completion queue cluster). A separate completion queue allows clients to completely avoid any context switches by directly polling the CQ while waiting for the RPC reply. Saving context switches can substantially reduce latency – up to 4μs in our deployment. Polling, however, increases the CPU load, and may lead to contention for CPU cycles among competing threads. In the case of multiple threads issuing concurrent RPC requests on the same connection, there might also be contention for accessing the shared completion queue.

CQ dispatching: An alternative design to direct CQ access is to share the completion queue among multiple connections and use a separate thread to query the CQ and dispatch events. The advantage is that there will be no contention for CQ access since the dispatcher owns the completion queue exclusively. This option is similar to the clustering used at the server side, with the one difference that work completion events cannot be processed in the context of the dispatcher but need to be passed to the application. In principle, there is nothing which prevents the dispatcher from enqueuing work completion events in per-connection queues and have applications poll these queues. A more CPU friendly approach, however, will be for the dispatcher to notify applications whenever a new work completion event has been received. Here, one extra context switch is required while dispatching notifications to application threads.

In the following, we compare the latency performance of the two options – direct CQ access and CQ dispatching – for

[2] We will use terms requests and packets interchangeably.

class/interface	API call	input/output parameters
DaRPCInstance	createInstance()	in: DaRPCService, Mode, Affinities[] out: DaRPCInstance
	createEndpoint()	out: DaRPCEndpoint
DaRPCService*	processRPC()	in: DaRPCParam out: DaRPCParam
DaRPCParam*	serializeTo()	in: byte[] out: byte[]
	serializeFrom()	in: byte[] out: byte[]
DaRPCEndpoint	connect()	
	accept()	out: DaRPCEndpoint
	createStream()	out: DaRPCStream
DaRPCStream	rpc()	in: DaRPCFuture out: DaRPCFuture
	poll()	out: DaRPCFuture
	take()	out: DaRPCFuture
DaRPCFuture	isDone()	out: boolean
	join()	out: DaRPCFuture
	getResponse()	out: DaRPCParam

Table 1. The DaRPC application programming interface. Interfaces are marked with *.

two different configurations. In the first configuration, multiple threads are issuing RPC calls concurrently using a single DaRPC connection to the server. In the second configuration, multiple threads are issuing RPC calls but each thread using its own separate connection. Figure 6 illustrates the interplay between threads, RPC connections, queue pairs and completion queues in each of these four cases. In the experiments, we further distinguish CQ dispatching with notification and CQ dispatching with polling. In the former, threads are being woken up by the dispatcher every time a RPC operation has completed. In the latter, the dispatcher places a notification message into a per-connection queue which is polled by the threads for this connection.

Single connection: For the single-connection configuration (top of Figure 7), the option of directly accessing the CQ achieves a RPC latency of 11.5μs if only a single thread is active. Compared to using a CQ dispatcher which results in latencies of 17μs and 25μs respectively, this is 30-60% faster. Clearly, avoiding context switches is key to keeping the overall RPC latency low. However, as the number of threads increases, the situation changes and using a CQ dispatcher becomes more favorable. At the extreme point with 16 threads, direct CQ access results in a per-request latency of 160μs, while CQ dispatching can achieve 34-40μs. These results confirm our thought that the contention for shared CQ access (due to locking) in the direct mode does not scale well with the number of threads. Instead, when sharing a single connection among many threads, eliminating contention for CQ and CPU access by using a dispatcher is more profitable. From the two modes implementing CQ dispatching, polling generally performs better than using notifications, but it is also more CPU intensive (not shown).

Multiple connections: When using a separate connection to the server for each client thread, one can expect to see improved results for the direct CQ option since at least the contention for the shared CQ is eliminated. This is confirmed by Figure 7 (bottom) which shows that the direct CQ option outperforms CQ dispatching in terms of latency throughout the band. For instance, with direct CQ access and a separate connection per thread, we can achieve 16μs RPC latency even if 16 threads are operating concurrently.

These experiments only provide a rough comparison of the two client-side design options. In practice, applications with multiple threads are unlikely to issue RPC requests back-to-back in a tight loop. Instead, one can expect more air between individual requests from single client threads, which in turn would make direct CQ access attractive even in configurations with many threads and a single connection. Generally, we believe that these experiments suggest that both approaches – direct CQ access and CQ dispatching – have their own advantages and disadvantages. We therefore decided in DaRPC to export both options at the API level to let the application choose which option matches best with the application requirements, workload pattern, hardware setup, etc. Typically, a latency sensitive application would use direct CQ access, while applications with many concurrent RPC operations would be better off with CQ dispatching.

4.4 Reliability, Ordering and Flow Control

One important question is regarding reliability in case of failures. DaRPC is designed to operate on top of a reliable transport protocol, such as TCP (in case of iWARP) or the Infiniband reliable transport layer. As such, RPC request and response messages will eventually be delivered, unless the connection breaks. In the latter, it is up to the application to try to re-establish the RPC connection and issue the RPC

request again. It is also up to the application to filter out duplicates in such a scenario.

Another interesting aspect is concerning the order in which RPC requests are processed at the server. Without work-stealing, all RPC requests issued on the same connection will be processed in order. With work-stealing, however, there is no guarantee on the order in which RPC requests are processed, even for requests that are issued on the same connection. Consequently, if a certain order needs to be enforced it has to be implemented at the application level.

Special attention is required in any RDMA-based system when it comes to flow control. Remember that a RPC request issued by a DaRPC client requires a RDMA receive operation to be pre-posted at the server. By no means, a client should be able to overrun the server and issue requests for which no receive is posted at the server. In DaRPC this is solved using per-connection rings of buffers. Prior to accepting a new connection, a DaRPC server posts RDMA receive operations for all the buffers in the ring and shares the size of the ring with the client. Receive buffers at the server are consumed by incoming requests, but can be re-posted immediately once the request is parsed and de-serialized. A client simply has to maintain a request budget which initially is set to the size of the ring, as advertised by the server. Subsequently, every request transmitted decrements the request budget, while incoming responses increment the budget. As long as the client only issues RPC requests if the budget is above zero, no overrunning of the server is possible.

```
/*Defined variables*/
MyRPCService s; /*implements DaRPCService*/
Affinities[] a = ...,
Mode m = CQ_DIRECT;
DaRPCInstance d;
DaRPCEndpoint serverEp, clientEp;
DaRPCStream stream;
DaRPCFuture future;
MyParam req, resp; /*implements DaRPCParam*/

/*new instance for the rpc service*/
d = DaRPCInstance.createInstance(s, a, m);
serverEp = d.createEndpoint();
/*simply wait for new RPC connections*/
while(true)
    clientEp = serverEp.accept();
```

(a)

```
d = DaRPCInstance.createInstance(s, a, m);
clientEp = d.createEndpoint().connect(...);
stream = clientEp.createStream();
future = stream.rpc(req, resp);
/* ... do something ... */
future.join();
if (future.isDone())
    print(future.getResponse());
```

(b)

Figure 8. DaRPC programming (a) server-side, (b) client-side

5. Application Programming Interface

The goal when designing the DaRPC interface was to provide an API that exports all the performance advantages of DaRPC, while at the same time being easy to use and powerful enough to satisfy the demands of large and complex distributed systems in the cloud.

DaRPC meets these requirements by providing an object-oriented asynchronous programming interface. The starting point for each DaRPC application is the `DaRPCInstance` class. A `DaRPCInstance` serves as a factory for RPC connections (`DaRPCEndpoint`). Internally, it implements an array of completion queue clusters. The application defines the size and the NUMA affinities for the clusters by passing an affinity set when creating the instance. A `DaRPCInstance` can be created in either CQ_DIRECT or CQ_DISPATCHED mode. Depending on the chosen mode, the `DaRPCEndpoint` objects created from this instance either have their own RDMA CQ or they share a completion queue as described in section 4.3. One key parameter to be passed to a `DaRPCInstance` is the actual RPC service. The RPC service is required to implement the `DaRPCService` interface, namely the `processRPC()` method which is invoked by the DaRPC runtime for every new RPC request that is received from the network. Both the input and output parameter of this method – referring to the request and the response of an RPC operation – need to be of type `DaRPCParam`, an interface which defines how these parameters are serialized. We use Java "generics" to avoid unnecessary casting from interface types to actual application specific types.

Server applications create a `DaRPCEndpoint`, bind it to a local interface (API call not shown in table) and accept new RPC connections from clients (see Figure 8a). Internally, the DaRPC runtime assigns connections to clusters within the `DaRPCInstance`. From this point on, all RPC processing for established connections is driven by the associated cluster in a context-switch-free and data-local manner as described in Section 4.

A client application will instantiate a `DaRPCInstance` using an implementation of `DaRPCService` that matches the one used at the server. Two implementations of `DaRPCService` match if they are both based on the same request/response types, which guarantees that the messages are wire-compatible. The client – unlike the server – is free to provide an empty implementation of `processRPC()`.

Following a proper instantiation of `DaRPCInstance`, a client application creates a `DaRPCEndpoint` and connects to the RPC server. The main client-side RPC interface is available via the `DaRPCStream` class. The application

uses `rpc()` to issue a new RPC request. This call is non-blocking and returns an object of type `DaRPCFuture`, which serves as a handle for the given RPC operation. At any point in time the application may poll the state of the RPC operation using `DaRPCFuture.isDone()`, or explicitly block for the RPC operation to complete using `DaRPCFuture.join()` (Figure 8). Completed RPC operations will further be enqueued with the `DaRPCStream` object that triggered the RPC in the first place. The `DaRPCStream` interface provides an alternative to `DaRPCFuture` for applications to keep track of RPC operations. For instance, the application may issue multiple RPC operations back to back and later use `DaRPCStream.take()` or `DaRPCStream.poll()` to retrieve the first operation that has completed. For any completed RPC operation `DaRPCFuture.getResponse()` can be used to access the response value.

6. Evaluation

We have implemented DaRPC in Java based on jVerbs [23], a high-performance RDMA stack for the IBM JVM. Earlier in section 4, we have identified three requirements that served as the main driving factors when designing DaRPC: high RPC throughput, low per-RPC latency, and the ability to integrate with cloud-based systems. We have already demonstrated some of the throughput and latency benefits of the different design choices we made in DaRPC. In this section, we want to first extend the evaluation by specifically investigating multi-core configurations and batching in more detail. Later, we look into a real use case and demonstrate how DaRPC can be applied to improve the performance of the HDFS namenode RPC service.

Setup: Our testbed consists of 17 nodes, each of which equipped with dual-socket 8-core Intel Xeon E5-2690 CPUs and a mix of Chelsio T4 and T5 10 Gbit/s adapters with iWARP/RDMA support. iWARP is an implementation of RDMA based on an offloaded TCP/IP/Ethernet stack. In our testbed, one node is used exclusively as the server while client instances are distributed over the remaining nodes. All measurements are done on Linux `3.6.0` using the `perf` framework [1].

6.1 Batching

If latency is not the major concern, applications can batch RPC calls to increase throughput. Figure 9 shows throughput and latency of DaRPC for different batch sizes. In our setup, client RPC requests are held back in a queue. Once a batch of requests is available a single combined RPC request is issued. Similarly, the server responds with batches of response messages transmitted as a single combined message. As expected with such a configuration, both the throughput and the latency increase with the batch size. With a batch size of 32, the RPC throughput reaches 9M ops/sec which corresponds to the bandwidth limit (10 Gbit/s) of the network interface.

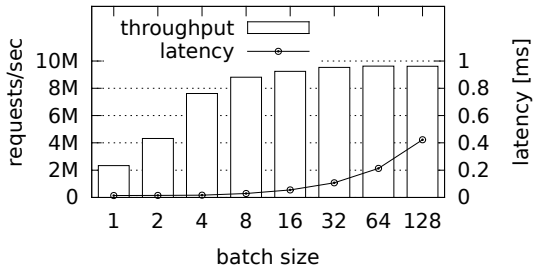

Figure 9. Effect of batching RPC requests.

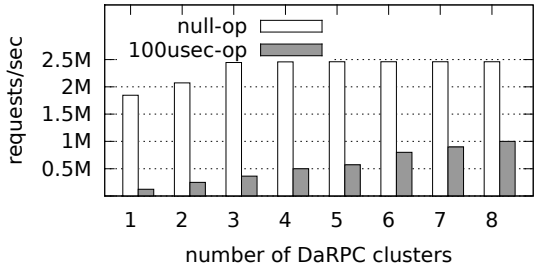

Figure 10. Scaling with the number of clusters.

6.2 Multicore scaling

DaRPC is designed to parallelize RPC processing across an array of clusters at the server side. Applications use the DaRPC API to provide an affinity set which determines the cores the clusters will be scheduled on. Since processing within the cluster is isolated from the processing of other clusters, one can expect the RPC throughput to scale linearly with the number of clusters.

We ran an experiment with 16-client machines where we measure the throughput of a DaRPC RPC service for different numbers of clusters being used at the server. The first series of bars in Figure 10 shows the results when using a "null" operation as the RPC service. As can be observed, the throughput increases from initially 1.8M ops/sec when using a single cluster, to 2.4M when using 3 clusters. 2.4M is also the maximum performance we can achieve with the Chelsio T5 network card, therefore adding more clusters did not increase the throughput further.

Using a "null" operation allows us to measure the raw performance of DaRPC, isolated from the performance of the actual RPC service. In practice, however, RPC services are implementing more complex and compute intensive operations. We emulate such a configuration by using an RPC service that occupies the DaRPC cluster head for 100μs instead of returning right away. As expected, with such a more "complex" RPC service only a fraction of RPC requests can

be processed by the server (Figure 10). Consequently, adding more clusters and thereby engaging more cores helps to increase the RPC throughput.

6.3 Use Case: HDFS

One promise of DaRPC is that it can be used to improve performance and scalability of distributed systems deployed in data centers today. HDFS is a well known distributed file system which serves as the basic storage layer for Hadoop, Spark, Hive, etc. The architecture of HDFS is centered around a nameserver for storing the file metadata such as block locations, permissions, directory structure, etc. Although the current version of HDFS supports a secondary nameserver for fault tolerance reasons, the primary instance of the nameserver typically still is a major performance bottleneck. In section 2, we have shown that the RPC interface of the HDFS nameserver can process between 100-130K ops/sec, at a latency of 200-500μs. This performance is limiting HDFS in two ways. First, the low throughput performance restricts both the size of the HDFS cluster as well as the performance of the read/write operations [22]. Second, the high RPC latencies prohibit the potential HDFS deployment in low-latency storage environments containing non-volatile memories such as Flash.

We argue that by using DaRPC one can implement a nameserver which scales to higher operations per second and provides lower latencies. As a demonstration we implemented DaRPC-HDFS, a stripped-down version of the HDFS nameserver based on DaRPC. The nameserver exports RPC operations to create, rename, and remove files, as well as operations to learn about the block locations of files. We further provide an implementation of the HDFS client interface (`hadoop.fs.FileSystem`) which allows applications to issue metadata operations using the standard HDFS API. Note that even though DaRPC-HDFS is no full-fledged file system and does not offer file read/write operations, its metadata server is fully functional. As such, any state that is created or manipulated by one HDFS operation will be visible to all subsequent operations.

In the following we compare the performance of HDFS metadata operations, once using the unmodified HDFS namenode, and once using DaRPC-HDFS. In the first experiment, clients continuously query the status of a file using the `getFileStatus()` API call. In the second experiment, clients continuously create and delete a file using `create()` and `delete()` calls. Each client operates on its own file. All the metadata operations used are standard HDFS API calls, we use the same benchmark for both unmodified HDFS and DaRPC-HDFS. We further look at two configurations: in the first configuration all nodes (namenode and clients) in the cluster use a Chelsio T4 RDMA network interface (10 Gbit/s), just as in all other experiments shown previously. In the second experiment, the client nodes use a plain 10 Gbit/s Ethernet interface without RDMA capabilities, but instead the RDMA capabilities are emulated in software.

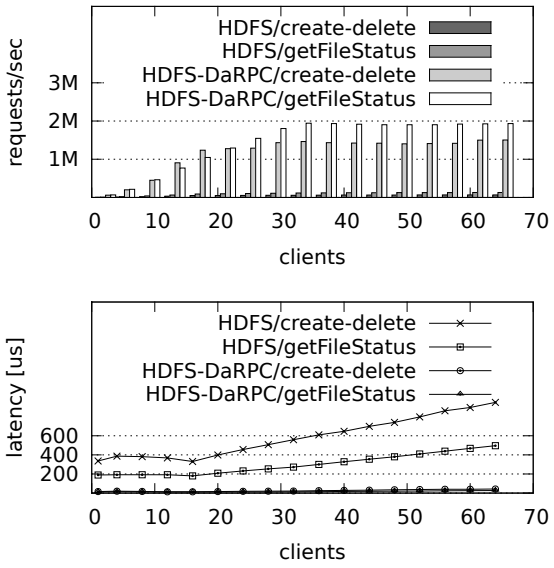

Figure 11. Throughput and latency of HDFS namenode operations in plain HDFS and HDFS-DaRPC measured using RDMA hardware at both namenode and clients.

Hardware-supported: Figure 11 shows the performance in terms of server throughput and client latency for an increasing number of clients. Clearly, HDFS-DaRPC can process more metadata operations per second (up to 2M for `getFileStatus()`) and also provides a much lower latency per operation (as low as 16μs). The throughput numbers are not exactly reaching the 2.4M operations that can be achieved with a raw DaRPC benchmark (Figure 10). However, this is expected since HDFS-DaRPC implements a more complex RPC service and uses larger RPC messages (144 bytes for both RPC request and response, as opposed to 16 bytes in Figure 10). The experiment using `create()/delete()` operations achieves a lower throughput in DaRPC-HDFS (around 1.5M) than the experiment using `getFileStatus()`. This is because both `create()` and `delete()` modify the directory structure and require the RPC service implementation to take a global lock.

One important observation from Figure 11 is that HDFS-DaRPC can sustain low latencies per operation even when operating under load. Table 2 gives some indication of how the different optimizations discussed in Section 4 pay off. Generally, HDFS-DaRPC produces fewer cache misses and fewer context switches per operation than the RPC implementations used in Zookeeper and HDFS. At the same time, HDFS-DaRPC uses only a moderate amount of CPU cycles, which is due to the offloaded TCP/IP stack and the efficient data path in RDMA.

	HDFS	HDFS-DaRPC
Link Utilization	1.65%	25%
CPU	15%	11.5%
Cache Misses	29%	11%
Context Switches	2.8/op	0.2/op

Table 2. Performance characteristics in HDFS and HDFS-DaRPC.

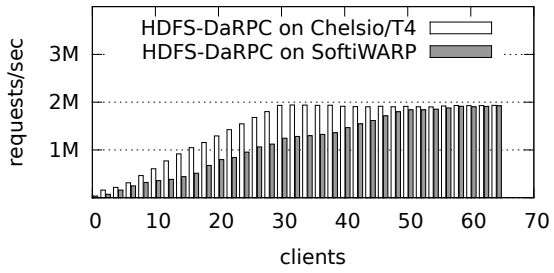

Figure 12. Throughput of HDFS namenode operations in plain HDFS and HDFS-DaRPC measured using RDMA hardware at the namenode and and SoftiWARP at the clients.

Software-based clients: One general concern with RDMA-based solutions is that they require special networking hardware to be deployed. However, with respect to RPC throughput, the critical piece in DaRPC is the server-side. That being said, we can restrict the high-end RDMA hardware to be deployed at the server and use software-based RDMA devices at the clients. SoftiWARP [25] is an in-kernel RDMA device for Linux working with regular Ethernet NICs while being wire-compatible with iWARP. In Figure 12 we show that a configuration with RDMA hardware deployed at the server and SoftiWARP deployed at the clients is indeed sufficient for HDFS-DaRPC to reach high throughput numbers. At the same time, latencies of metadata operations increase due to the missing hardware acceleration at the clients. Overall, we believe that employing DaRPC in such a hybrid setup is very attractive for distributed systems running in commodity data centers. By adding RDMA hardware support to performance critical server machines (e.g., HDFS namenode, Zookeeper server, OpenFlow Controller, etc.), DaRPC can substantially improve throughput, scaling and latency of RPC operations, without the necessity of any special hardware being deployed at the clients.

7. Related Work

RPCs have been the backbone of many distributed systems. With the emergence of high-performance networks in the 1990s many projects looked at low-latency application to application data transfer operations in the context of RPCs [4, 10, 20, 24]. These systems identify RPC performance overheads stemming from data copies, context switches, inefficient host interfaces, marshaling and unmarshaling parameters, protocol and packet processing etc. Modern high-performance RDMA interfaces and userspace networking stacks are built upon their findings. DaRPC goes a step further and integrates multicores, CPU/NIC-locality, RPC capacity scalability under load, and language run-time considerations (e.g. managed runtime such as JVM) etc. into the RPC performance.

RPCs are known to be small and packet-processing heavy. On modern 10 or 40 Gbits/s networks they can easily generate millions of requests per second. Recently, there has been a lot of work aiming at efficiently handling small packets on multi-core servers [8, 9, 19, 21]. Further, the availability of fast userspace packet processing frameworks has also led to userspace implementations of the complete network and application stack. In a similar spirit to DaRPC, mTPC [9] manages packet processing, TCP connection management and application interface in userspace for best parallelism. However, the primary focus of mTCP is high small-packet throughput that is achieved by eschewing the kernel as well as by efficient packet- and event-level batching on the I/O path. DaRPC's architecture of handling individual RPC requests by a single core (with a minimum context switch overhead), is inspired from RouteBricks [6].

In modern large-scale distributed systems, RPC performance has mostly been treated as an optimization problem in the process of building large distributed systems. For instance, the RPC framework of Tango leverages special operating system features to achieve 570K RPC ops/sec with a latency of 35μs per operation in the single-client scenario [3]. Building RPC stacks as part of a large-scale distributed system (e.g. Key-Value store) enables across-the-layer performance optimizations. One such optimization is extensive batching done at the different layers to amortize the cost of small packet handling for RPC. The Percolator system runs a centralized timestamp oracle implemented as a RPC service at a throughput of 2M req/sec using batching [18]. Similarly, the vector interface proposed by Vasudevan et al. for their key-value store can achieve 1.6 M req/sec on a single server [26]. Masstree is another key-value server that provides 6M queries/sec with batching [14]. Though batching increases the RPC throughput it does so at the cost of latency. As an example, the latency numbers at the peak performance of mTCP are orders of magnitude slower (in msecs) than those of DaRPC. We have shown that DaRPC can scale linearly with the batch size and process close to 10M RPC/sec at sub-millisecond latency.

MICA [12] is a recently proposed Key-Value store that handles packet I/O and key-value request processing as a joint problem. Both MICA and DaRPC need efficient network I/O processing, however, there are some fundamental differences in the way MICA and DaRPC handle requests. MICA leverages the fact that in the key-value store values

	Locality	Reliability	Batching	Application Interface	Zero copy	Performance Scaling	Low Latency	Cloud Ready
SHRIMP RPC [4]	shared	yes	none	SunRPC	yes	N/A	yes	no
mTCP [9]	per-core	yes, TCP	function calls	mTCP sockets	no	yes	no	no
MegaPipe [8]	per-core	yes, TCP	syscalls	lwsockets	no	yes	no	no
MICA [12]	per-core/NIC	no, lossy	packets	Key-Value	yes	yes	yes	no
VectorOS [26]	shared	yes, TCP	I/O requests	Key-Value	no	yes	no	no
DaRPC	per-core	yes, TCP	none	Async. RPC	yes	yes	yes	yes

Table 3. Comparison of networking efforts to achieve high-performance with small transfer sizes.

are generally less than a packet size. Hence, MICA cannot handle request sizes greater than a packet size. Moreover, it uses UDP for I/O and in the case of a packet loss, client needs to detect and re-transmit the lost request (or packet). Though it is acceptable for idempotent key-value requests, this behavior is undesired for RPCs. RPCs usually involve state changes on the data or service server and require a consistent invoked-once semantic.

Furthermore, none of the systems discussed above are implemented in a managed runtime, which makes their deployment in a cloud/virtualized environment difficult. Table 3 captures critical properties of these systems and compares them against DaRPC.

There has recently been an increasing interest in using RDMA networks in cloud-based distributed systems [7, 11, 15]. These works do not address RPC performance in particular, but they are demonstrating the potential of a close integration between networking and higher-level systems.

A work which specifically tackles the RPC performance in Hadoop is [13]. By using a more sophisticated buffer allocation scheme and zero-copy on the transmission path, their work is able to improve the RPC performance in Hadoop by up to 82%.

8. Conclusion

In this work we presented DaRPC, a framework for high-throughput low-latency RPC. DaRPC uses RDMA to implement a tight integration between RPC processing and network processing in user space. By jointly distributing RPC and RDMA resources across cores and memory, DaRPC manages to increase parallelism and scale to large numbers of RPC requests. In the paper we demonstrated that DaRPC can process up to 2.4M RPC requests/s with a single server at latencies of 3-10μs per RPC. DaRPC is written entirely in Java and therefore can be used comfortably by many of the distributed systems deployed in the cloud today. As shown in this work, DaRPC can improve scaling and performance of large scale distributed systems running data centers such as HDFS.

Acknowledgement

We thank the anonymous reviewers and our shepherd, Indranil Gupta, for their helpful comments and suggestions. We also would like to thank Felix Marti and Joerg-Eric Sagmeister for their support with the networking hardware.

References

[1] Perf: Linux Profiling with Performance Counters. http://perf.wiki.kernel.org/.

[2] Zookeeper Performance. http://wiki.apache.org/hadoop/ZooKeeper/.

[3] M. Balakrishnan, D. Malkhi, T. Wobber, M. Wu, V. Prabhakaran, M. Wei, J. D. Davis, S. Rao, T. Zou, and A. Zuck. Tango: Distributed Data Structures over a Shared Log. In *Proceedings of SOSP'13*, pages 325–340. ISBN 978-1-4503-2388-8. URL http://doi.acm.org/10.1145/2517349.2522732.

[4] A. Bilas and E. W. Felten. Fast RPC on the SHRIMP Virtual Memory Mapped Network Interface. *J. Parallel Distrib. Comput.*, 40(1):138–146, Jan. 1997. ISSN 0743-7315. URL http://dx.doi.org/10.1006/jpdc.1996.1272.

[5] A. D. Birrell and B. J. Nelson. Implementing Remote Procedure Calls. *ACM Trans. Comput. Syst.*, 2(1):39–59, Feb. 1984. ISSN 0734-2071. URL http://doi.acm.org/10.1145/2080.357392.

[6] M. Dobrescu, N. Egi, K. Argyraki, B.-G. Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy. RouteBricks: Exploiting Parallelism to Scale Software Routers. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 15–28, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-752-3. URL http://doi.acm.org/10.1145/1629575.1629578.

[7] A. Dragojević, D. Narayanan, O. Hodson, and M. Castro. FaRM: Fast Remote Memory. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI'14, pages 401–414, Berkeley, CA, USA, 2014. USENIX Association. ISBN 978-1-931971-09-6. URL http://dl.acm.org/citation.cfm?id=2616448.2616486.

[8] S. Han, S. Marshall, B.-G. Chun, and S. Ratnasamy. MegaPipe: A New Programming Interface for Scalable Network I/O. In *Proceedings of OSDI'12*, pages 135–148. ISBN 978-1-931971-96-6. URL http://dl.acm.org/citation.cfm?id=2387880.2387894.

[9] E. Y. Jeong, S. Woo, M. Jamshed, H. Jeong, S. Ihm, D. Han, and K. Park. mTCP: A Highly Scalable User-level TCP Stack for Multicore Systems. In *Proceedings of*

the 11th USENIX Conference on Networked Systems Design and Implementation, NSDI'14, pages 489–502, Berkeley, CA, USA, 2014. USENIX Association. ISBN 978-1-931971-09-6. URL http://dl.acm.org/citation.cfm?id=2616448.2616493.

[10] D. B. Johnson and W. Zwaenepoel. The Peregrine High-performance RPC System. *Softw. Pract. Exper.*, 23(2):201–221, Feb. 1993. ISSN 0038-0644. URL http://dx.doi.org/10.1002/spe.4380230205.

[11] A. Kalia, M. Kaminsky, and D. G. Andersen. Using RDMA Efficiently for Key-value Services. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, pages 295–306, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2836-4. URL http://doi.acm.org/10.1145/2619239.2626299.

[12] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky. MICA: A Holistic Approach to Fast In-memory Key-value Storage. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI'14, pages 429–444, Berkeley, CA, USA, 2014. USENIX Association. ISBN 978-1-931971-09-6. URL http://dl.acm.org/citation.cfm?id=2616448.2616488.

[13] X. Lu, N. S. Islam, M. Wasi-Ur-Rahman, J. Jose, H. Subramoni, H. Wang, and D. K. Panda. High-Performance Design of Hadoop RPC with RDMA over InfiniBand. In *Proceedings of the 2013 42Nd International Conference on Parallel Processing*, ICPP '13, pages 641–650, Washington, DC, USA, 2013. IEEE Computer Society. ISBN 978-0-7695-5117-3. URL http://dx.doi.org/10.1109/ICPP.2013.78.

[14] Y. Mao, E. Kohler, and R. T. Morris. Cache Craftiness for Fast Multicore Key-value Storage. In *Proceedings of Eurosys'12*, pages 183–196. ISBN 978-1-4503-1223-3. URL http://doi.acm.org/10.1145/2168836.2168855.

[15] C. Mitchell, Y. Geng, and J. Li. Using One-sided RDMA Reads to Build a Fast, CPU-efficient Key-value Store. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*, USENIX ATC'13, pages 103–114, Berkeley, CA, USA, 2013. USENIX Association. URL http://dl.acm.org/citation.cfm?id=2535461.2535475.

[16] D. Ongaro, S. M. Rumble, R. Stutsman, J. Ousterhout, and M. Rosenblum. Fast Crash Recovery in RAMCloud. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 29–41, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0977-6. URL http://doi.acm.org/10.1145/2043556.2043560.

[17] D. Pariag, T. Brecht, A. Harji, P. Buhr, A. Shukla, and D. R. Cheriton. Comparing the Performance of Web Server Architectures. In *Proceedings of Eurosys'07*, pages 231–243. ISBN 978-1-59593-636-3. URL http://doi.acm.org/10.1145/1272996.1273021.

[18] D. Peng and F. Dabek. Large-scale Incremental Processing Using Distributed Transactions and Notifications. In *Proceedings of OSDI'10*, pages 1–15. URL http://dl.acm.org/citation.cfm?id=1924943.1924961.

[19] A. Pesterev, J. Strauss, N. Zeldovich, and R. T. Morris. Improving Network Connection Locality on Multicore Systems. In *Proceedings of Eurosys'12*, pages 337–350. ISBN 978-1-4503-1223-3. URL http://doi.acm.org/10.1145/2168836.2168870.

[20] H. V. Shah, C. Pu, and R. S. Madukkarumukumana. High Performance Sockets and RPC over Virtual Interface (VI) Architecture. In *Proceedings of the Third International Workshop on Network-Based Parallel Computing: Communication, Architecture, and Applications*, CANPC '99, pages 91–107, London, UK, UK, 1999. Springer-Verlag. ISBN 3-540-65915-3. URL http://dl.acm.org/citation.cfm?id=646093.680566.

[21] L. Shalev, J. Satran, E. Borovik, and M. Ben-Yehuda. IsoStack: Highly Efficient Network Processing on Dedicated Cores. In *Proceedings of USENIX ATC'10*, pages 5–5. URL http://dl.acm.org/citation.cfm?id=1855840.1855845.

[22] K. V. Shvachko. HDFS Scalability: The Limits to Growth. 2010.

[23] P. Stuedi, B. Metzler, and A. Trivedi. jVerbs: Ultra-low Latency for Data Center Applications. In *Proceedings of SOCC'13*, pages 10:1–10:14. ISBN 978-1-4503-2428-1. URL http://doi.acm.org/10.1145/2523616.2523631.

[24] C. A. Thekkath and H. M. Levy. Limits to Low-latency Communication on High-speed Networks. *ACM Trans. Comput. Syst.*, 11(2):179–203, May 1993. ISSN 0734-2071. URL http://doi.acm.org/10.1145/151244.151247.

[25] A. Trivedi, B. Metzler, and P. Stuedi. A Case for RDMA in Clouds: Turning Supercomputer Networking into Commodity. In *Proceedings of the 2nd APSys'11*, pages 17:1–17:5. ISBN 978-1-4503-1179-3. URL http://doi.acm.org/10.1145/2103799.2103820.

[26] V. Vasudevan, M. Kaminsky, and D. G. Andersen. Using Vector Interfaces to Deliver Millions of IOPS from a Networked Key-value Storage Server. In *Proceedings of SOCC'12*, pages 8:1–8:13. ISBN 978-1-4503-1761-0. URL http://doi.acm.org/10.1145/2391229.2391237.

[27] M. Welsh, D. Culler, and E. Brewer. SEDA: An Architecture for Well-conditioned, Scalable Internet Services. In *Proceedings of SOSP'01*, pages 230–243. ISBN 1-58113-389-8. URL http://doi.acm.org/10.1145/502034.502057.