

Low-Latency Communication for Fast DBMS Using RDMA and Shared Memory

Philipp Fent, Alexander van Renen, Andreas Kipf, Viktor Leis*, Thomas Neumann, Alfons Kemper
Technische Universität München
{fent,renen,kipf,neumann,kemper}@in.tum.de

Friedrich-Schiller-Universität Jena*
viktoria.leis@uni-jena.de*

Abstract—While hardware and software improvements greatly accelerated modern database systems’ internal operations, the decades-old stream-based Socket API for external communication is still unchanged. We show experimentally, that for modern high-performance systems networking has become a performance bottleneck. Therefore, we argue that the communication stack needs to be redesigned to fully exploit modern hardware—as has already happened to most other database system components.

We propose L5, a high-performance communication layer for database systems. L5 rethinks the flow of data in and out of the database system and is based on direct memory access techniques for intra-datacenter (RDMA) and intra-machine communication (Shared Memory). With L5, we provide a building block to accelerate ODBC-like interfaces with a unified and message-based communication framework. Our results show that using interconnects like RDMA (InfiniBand), RoCE (Ethernet), and Shared Memory (IPC), L5 can largely eliminate the network bottleneck for database systems.

I. INTRODUCTION

Modern main-memory database systems can process hundreds of thousands of TPC-C transactions per second [51], and for key/value-style workloads, millions of transactions per second are possible [53], [51]. Such benchmark results are, however, virtually always measured by generating the workload within the database system itself—ignoring the question of *how to get the load into the system* in the first place.

For decades, the standard approach for communication between different processes has been (and still is) the operating system’s Socket API. Sockets are well-understood, widely-available, fairly portable, and fast enough for traditional database systems. For example, using OLTP-Bench [14] we measured that PostgreSQL achieves around 220 TPC-C transactions per second using one thread. At these low transaction rates, standard Sockets are not the bottleneck—even though OLTP-Bench does not use stored procedures but rather sends each SQL statement separately over the network.

For modern in-memory database systems the situation is very different: We found that the backend of Silo [51] can process 58 thousand transactions per second using a single thread (more than 200× faster than PostgreSQL). However, unlike OLTP-Bench, this number does not include communication (the very thread processing transactions also generates the workload). As Figure 1 shows, once we send each SQL statement through the operating system’s network stack, the performance drops to 1,497 using TCP (39× slower) or 2,710 using Domain Sockets (21× slower). These numbers show that for high-performance

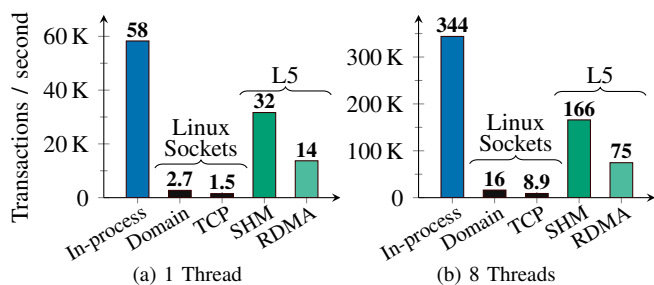


Fig. 1. Communication Bottleneck. TPC-C throughput using Silo.

database systems, networking and inter-process communication have become *the* performance bottleneck.

It is important to realize that slow communication is *not* due to fundamental limitations of the networking hardware. To achieve Silo’s backend transaction processing performance, the communication would need to support roughly one million round trips per second. On a hardware level, both Ethernet and InfiniBand have this capability with single-digit microsecond latencies—corresponding to hundreds (rather than tens) of thousands of round trips per second. Furthermore, using Shared Memory, modern processors can exchange more than one million messages per second.

Given these hardware characteristics, one may wonder why most systems still rely on Sockets. After all, several cloud providers already provide RDMA-capable instances [10], [2] and Shared Memory is available on any system. We believe that the main reason, as is often the case when a technically superior solution fails to become widely adopted, is ease of use. To fully exploit today’s networking hardware, one has to use hard-to-use APIs like InfiniBand’s Remote Direct Memory Access (RDMA) or its Ethernet pendant RDMA over Converged Ethernet (RoCE). For fast inter-process communication on the same machine, one has to implement concurrent message passing in Shared Memory. Finally, one also has to orchestrate and coordinate client and server processes to set up and use these low-level techniques. Needless to say, this is much harder and less portable than simply using Sockets.

To address these problems, we propose the *Low-Level, Low-Latency messaging Library (L5)*. L5 replaces traditional Sockets and can transparently be configured to use RDMA (InfiniBand), RoCE (Ethernet), or Shared Memory (IPC) as a communication channel. For both remote communication over InfiniBand, as well as between isolated processes on the same machine, L5 improves throughput and latency by over an order of magnitude.

This paper’s contributions are as follows:

- 1) We introduce L5 as a unified communication interface to address the problems with adaptive switching of traditional protocols with direct memory access (Section II).
- 2) We provide an efficient implementation of local communication using Shared Memory (Section III).
- 3) We show that a high-performance implementation using RDMA can provide similar performance within a data center (Section IV).
- 4) We demonstrate in microbenchmarks and in an end-to-end evaluation based on YCSB [13] that we can reach one million transactions per second with just a single client (Section V).

II. L5: LOW-LEVEL, LOW-LATENCY MESSAGING LIBRARY

The network bottleneck makes it necessary to redesign the communication stack of data management systems. As a core building block, we present **L5 (Low-Level, Low-Latency Library)**. L5 provides (1) high transaction throughput for small payloads, (2) optimal performance with few clients, and (3) a unified interface for adaptive selection of the best available communication technology. Furthermore, L5 provides very low latency without relying on batching at the application level.

L5 achieves this by bootstrapping high-performance connections via regular Sockets, but then switches to a faster communication channel. For intra-machine communication, e.g., on a container host, L5 upgrades the connection to use Shared Memory. For intra-datacenter communication, we instead upgrade the connection to RDMA, all while providing the same interface, no matter the underlying implementation.

In the following, we provide an extensive analysis and evaluation of database interconnects and document an optimized direct memory access protocol for low latency database system communication. L5 is publicly available: <https://github.com/pfent/L5RDMA>

A. Messaging Layer

L5 provides a message-based communication layer, designed after the protocols which we found to be the default for all database systems. This accelerates the synchronous use-case: Applications send statements to a database server, wait for a reply, and then (based on the reply) continue their execution.

Traditionally, database systems would avoid this problem by giving applications the possibility to move most of the interactive logic inside the database system using stored procedures. However, empirical evidence shows that many applications are not willing to move their business logic to the database. Andrew Pavlo [38], for example, presented results of a database administrator survey on real-world database system usage. More than half of the DBAs reported that they do not use stored procedures or only very rarely. By neglecting communication performance, current database systems cannot cater the needs of one half of their users.

This is especially limits applications with data dependencies between statements. While techniques such as batching can help for the simplest data dependencies like issuing queries

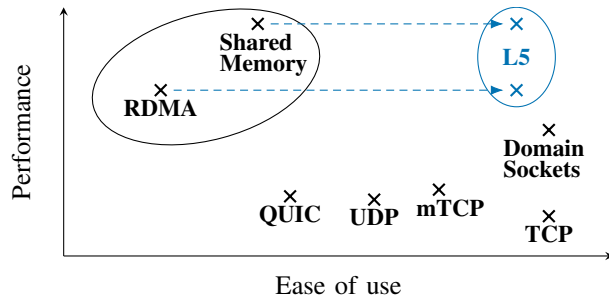


Fig. 2. Communication Technologies. An overview of network and IPC technologies. L5 brings ease of use to high-performance interconnects.

in a loop to load a set of values, transactional workloads like the TPC-C payment already need multiple dependent round-trips for each transaction. Inherently data-dependent workloads such as graph traversals even need a round-trip per node, extremely amplifying communication overhead. This leads to the somewhat paradoxical situation that, while clients are starved due to communication overhead, DBMS load is still low. With L5, data-dependent applications can instead make use of the otherwise idle DBMS resources.

Additionally, there are also applications that have real-time requirements. One example are financial transactions, similar to the brokerage described in TPC-E [11]. Depending on the outcome of analyzing transactions, client decision systems execute or abort Trade-Order transactions. For clients, the system response time is crucial, since faster issuing of buys or sells at market value might give better prices.

While higher parallel throughput (and generally more bandwidth) can always be achieved by using more network hardware, reducing response times actually needs careful optimization. For the size of the messages, L5 targets around 100 Byte, which is based on the fact that the commonly used TPC-C benchmark [27] has a weighted average of around 49 Byte payloads per transaction¹.

B. System Integration

Existing implementations and installations make it necessary to design a system that can be integrated into database systems. One way to allow for a higher transaction ingestion rate would be to simply eliminate expensive context switches between kernel and user space by using a user-space network library such as mTCP [21].

However, we found that mTCP offers minimal performance benefit for single client scenarios and is inferior to RDMA for a larger number of clients (cf. Section IV). In fact, it takes over 100 clients to saturate an mTCP interface with 1.2 million msgs/s. Instead, we want to be able to saturate a system with only a handful of clients, which is not possible with current TCP-based interfaces.

If we instead use communication based on direct memory access, we can also partially relax the strong guarantees of TCP

¹TPC-C uncompressed payload literals profile: 45 % new-order with an average of 82 Byte, 43 % payment with 24 Byte, 4 % delivery with 8 Byte, 4 % order-status with an average of 19.2 Byte, and 4 % stock-level with 12 Byte.

and the Socket API. When sending data over a Socket, it is impossible to retroactively interact with the message contents. On the contrary, in a shared-access message buffer, data can still be read back and even be changed while the remote side reads it. While this has the new potential for “time-of-check to time-of-use” bugs, simply copying the data out of the message buffer (similar to how Sockets work under the hood) alleviates this problem, while still providing excellent performance. On the other hand, using direct memory access does not need to relax the more serious guarantees of strong isolation between client and server, and between different clients connected to the same server, since it only needs to have the message buffer as a shared object between one client and the server.

We categorize the complexity of the different connection technologies compared to the potential benefit in Figure 2: While TCP has the lowest barrier of entry and is used almost everywhere, mTCP is only a moderate improvement for each connection. UDP and protocols on top of it, like QUIC [20], also have the potential to improve performance, but currently lack hardware acceleration. Local communication options like Domain Sockets and Shared Memory are more efficient, but can obviously only be used between processes on the same machine. When using RDMA for remote communication, the potential of modern database management systems can be leveraged almost as if they run on the same machine, and maximum remote throughput can be reached with few clients.

In summary, L5 can significantly increase the ease of use for complex direct memory access protocols: Bootstrapping over regular sockets allows zero configuration and setup overhead, and L5’s unified interface eliminates the need to write thousands of lines of codes for RDMA and Shared Memory. In combination, this allows effortless integration of high-performance interconnects into the ODBC driver or the database connectivity library of existing systems.

C. Why Sockets are Slow

The implementation of operating system interfaces has seen a lot of development and its implementation is generally very optimized. We argue that the reason behind Sockets lying in the lower half of the performance spectrum is their fundamental requirement of interoperability between architectures and operating systems. A database system, on the other hand, has more freedom to optimize the common case. Nevertheless, they still use the operating system’s TCP Sockets, which is the only protocol that allows reliable communication with a wide interoperability, and is the de-facto standard for a wide variety of different use-cases.

This interoperability is also the main source for TCP’s complexity, which needs to support a wide variety of networks. Contrarily, data management systems are often located in datacenters where connection endpoints are as close to each other as physically possible, either on the same network, or even the same machine, only separated by thin virtualization layers. TCP needs to deal with many edge cases, which simply never appear in such environments and uses byte streams, which do not fit the use-case of database systems, where both

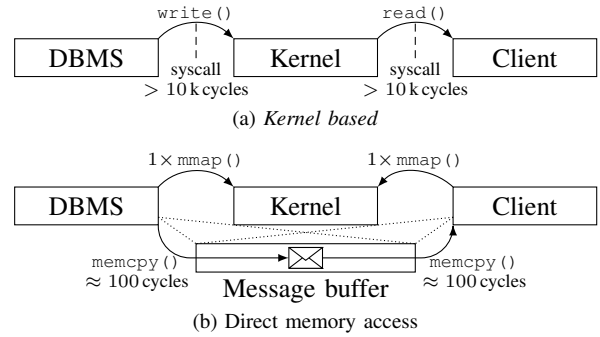


Fig. 3. Communication Concepts. Kernel based communication requires expensive system calls, while direct memory access allows cheap communication.

queries and transactional statements follow a request-reply pattern with clear message semantics. This mismatch manifests in most database access protocols in use today, where all implementations we know of use messages on top of TCP’s byte stream semantics.

Local alternatives like Domain Sockets can solve many of TCP’s problems and significantly improve performance. But even after shedding most overhead, they still bottleneck modern in-memory database systems. In addition to TCP’s complexity, system call overhead causes each message to consume more than 10 000 cycles [47]. Recent mitigations for side channel attacks like Meltdown [31] additionally amplify this effect. We can instead use direct memory access, to which we will refer in Unix terms as Shared Memory (SHM) and use a common memory area to exchange messages. Figure 3 visualizes this way of bypassing the kernel, where reading and writing data only takes about 100 cycles.

L5 provides a way to use direct memory access for communication with a similar interface to those existing techniques. E.g., it can use the same connection configurations as sockets, since they bootstrap SHM, while it simultaneously avoids the problem of noticing new or disconnected connections over SHM alone. On top of the SHM or RDMA message buffer L5 implements effective polling for new messages without any system calls and allows equally efficient sending and receiving of data. Both, RDMA and Shared Memory have more subtle challenges, which we will discuss in the following two chapters.

III. LOCAL MESSAGING

In many latency-critical applications, the database client (e.g., a web server) and the database server are located on the same machine. In this setting, it might still be desirable to have a separation of client and server into separate processes. Lightweight container solutions like Docker make this setup increasingly popular, since they make it easy to safely host different applications on the same machine. Containers should have very good local messaging performance without the need for heavyweight network protocols, since applications can also communicate through Shared Memory. This application is different from single process database systems like SQLite [3], where messages do not cross process boundaries. However, unlike other high-performance applications, e.g., browsers or

```

char* client_setup_shm(int connection_to_server)
int fd = memfd_create("debug_name",
MFD_CLOEXEC | MFD_ALLOW_SEALING)
ftruncate(fd, SIZE)
send_fd(connection_to_server, fd)
return mmap(NULL, SIZE, prot, flags, fd, 0)
char* server_setup_shm(int connection_to_client)
int fd = recv_fd(connection_to_client)
fcntl(fd, F_ADD_SEALS, F_SEAL_SHRINK)
ftruncate(fd, SIZE)
// SHRINK_SEAL ensures: filesize >= SIZE
return mmap(NULL, SIZE, prot, flags, fd, 0)

```

Fig. 4. Shared Memory Setup. Code for safely setting up Shared Memory mappings between two processes with `memfd_create()`.

display servers, database systems rarely make use of Shared Memory for communication.

In this section, we show how database systems can use L5’s Shared Memory messaging layer, which greatly outperforms other techniques. For local communication, L5 offers a one-to-one channel, which is instantiated for every client.

A. Shared Memory

Shared Memory only offers direct sharing of system memory resources via low-level access. This lack of safe interfaces requires careful implementation, but has unprecedented performance. In this section, we show that we can safely set up Shared Memory, which still ensures proper process isolation.

1) *Ring Buffer Setup*: For the initial connection establishment, L5 uses a Domain Socket. They provide many connection management features, and are a convenient out-of-bounds control communication channel. After connection establishment, we can use the standard set of system calls (`shm_open`, `ftruncate`, `mmap`) to create, map, and exchange a Shared Memory segment. L5 exchanges the Shared Memory file descriptor via the Domain Socket ancillary data channel (`cmsg`), which bootstraps the high-performance connection.

The well-known setup still has some unfortunate pitfalls, which need to be addressed to maintain a database’s safety requirements. First, we require that clients allocate the initial memory segment. Otherwise, a client could control the server’s memory allocation and bypass its own resource limitations (`ulimit`, `cgroup`). Second, the default Shared Memory mappings are visible for third-party processes. To ensure that only client and server can read the memory, we require non-standard extensions for unnamed anonymous mappings (`O_TMPFILE` on Linux or `SHM_ANON` on FreeBSD).

Third, the most intricate problem is that clients could also arbitrarily manipulate the underlying file. A malevolent client might shrink the file, causing the server to read beyond file boundaries. This causes a `SIGBUS` signal for the server, which is very hard to handle correctly. Identifying the causing file and client would require significant runtime introspection, which itself causes more problems than Shared Memory solves.

Since version 3.17, Linux has the most mature way to deal with all of those problems: the `memfd_create` system call. With it, we can create an anonymous memory mapping by default and additionally can “seal” the underlying file. By applying a seal, we permanently fix the sealed file’s properties.

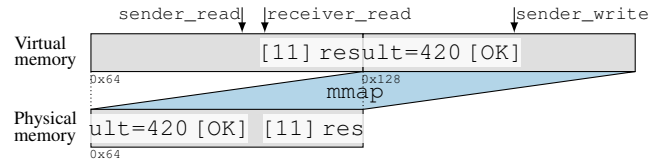


Fig. 5. Ring Buffer. For communication on one machine, we use a ring buffer in Shared Memory with a virtual memory wrap-around mapping.

With `F_SEAL_SHRINK` we disallow any shrinking of the file, eliminating the need to handle `SIGBUS` errors. Figure 4 shows a simplified version of L5’s setup, which eliminates unintended client interference with the database server.

For the actual communication, we place two ring buffers into this Shared Memory segment: One for sending messages from the client to the server and one for the reverse. Figure 5 shows the ring buffer’s memory layout with a virtual memory wrap-around mapping. With a second call to `mmap`, we configure the memory mapping in a way that the buffer’s consecutive virtual memory addresses map to the same physical address. This common technique simplifies the implementation, as writes to the buffer automatically wrap around.

2) *Ring Buffer Management*: The ring buffer has two main purposes: To store in-flight messages and to catch under- and overflow situations. For the messages, we use a simple message format of the message size followed by the specified amount of bytes. The buffer can then be polled for the next message by reading the next size. Since random access in RAM is cheap as long as there is no cache contention, we first write the actual message and afterwards set the size. Once the size is set, x86’s total store order guarantees that the message has already been written completely.

We additionally maintain three pointers to track free space:
receiver_read This pointer is stored in Shared Memory, but is only written by the receiver. It points to the first byte of the next message to be received. The receiver polls this memory, until it reads a non-zero value. In the example in Figure 5, the receiver reads 11, indicating a message of that size. It can then read the actual message, do its necessary processing, zero out the memory (required to allow polling the size), and then advance `receiver_read`.

sender_read This pointer is only stored at the *sender* (not in Shared Memory). It ensures enough remaining empty buffer space, preventing the sender from overwriting not yet read messages. This pointer is a copy of the `receiver_read`, and caches it lazily to reduce latency by minimizing cache contention. It is synced occasionally (necessarily when the buffer appears to be full, but ideally slightly before without data dependencies) with the `receiver_read` pointer. Therefore, it is not always up to date and can lag behind the real progress of the receiver, as shown in the example.

sender_write This pointer is also only stored at the *sender*. It points to the address that the next message should be written to. When sending a message, we first check, if the buffer has enough remaining capacity by querying the `sender_read` pointer. Then it can first write the actual message, and then set the preceding size and finally advance `sender_write`.

3) *Adaptive Polling*: On both sides, direct memory polling ensures minimum latency and thereby increases throughput. However, when there is not much traffic on the connection, it consumes an entire CPU core without doing any useful work. To avoid wasting resources, we deploy an adaptive polling scheme, which detects an idle connection and backs off to less resource-intensive methods: After sending a message, L5 assumes a reply within a short duration and uses polling. After a configurable number of tries, L5 stops busy polling and uses `yield` commands to allow other threads to run on the core. When even more time passes and no new message were received, the thread transitions to waiting.

We can use a binary semaphore to safely and efficiently fall back to blocking, but we require cross-process synchronization. On POSIX systems, this is possible using a `PTHREAD_PROCESS_SHARED` mutex and a binary condition variable. On the receiver side, the transition is made by first locking the mutex then setting an atomic flag (`sleeping`). This flag indicates a receiver waiting on the condition variable. Since a message could come in between checking the buffer and setting the `sleeping` flag, the receiver needs to check the ring buffer once again. This process guarantees a transition to waiting on the condition variable, without missing a message. On the sender side, the `sleeping` flag is checked after sending a message, which does not increase response time.

B. Shared Memory Bandwidth

While the previous sections focus on achieving high synchronous throughput for small messages, Shared Memory also provides high bandwidth. Since there are several tuning knobs, we also optimize bandwidth to achieve high throughput, not only for small messages, but also for the occasional big data transmission, which can equally profit from using Shared Memory.

Baseline: Between processes, we are not limited by the available network bandwidth, but only by local memory speed. Our Intel Xeon E5-2660 v2 has a theoretically available memory bandwidth of 60 GByte/s (more details in Section V-A), but can only be saturated using multiple threads. As a baseline, the single threaded `STREAM` Benchmark [35] on our system achieves 6.9 GByte/s for the copy operation. In our case of inter process communication and given that there is some synchronization overhead, our goal therefore is to get as close as possible to that number.

Parameters: To determine the optimal parameters for maximum bandwidth, we transmit 10 GB over a Shared Memory connection and measure the average bandwidth of this transmission. The heat map plot in Figure 6 can be used to determine the optimal configuration to transmit data over Shared Memory. The y-axis varies the size of the underlying transmission buffer, which stores the “in-flight” data. On the x-axis, we vary the size of the individually transmitted chunks. This chunked transmission is necessary, because we transmit more data than the underlying buffer can store. Therefore, we copy a chunk of n Byte into the Shared Memory segment, then increment `sender_write` by n and repeat. The upper

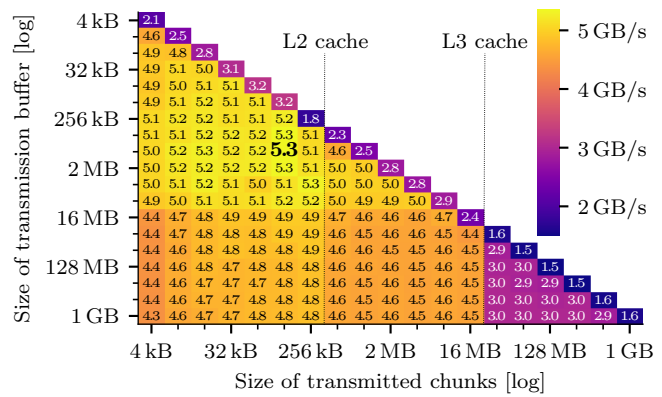


Fig. 6. Efficient Shared Memory Usage. Heat map indicating the achievable bandwidths using different buffer and chunk sizes to transmit large amounts of data.

right-hand side is empty, because writing chunks exceeding the underlying buffer’s size is impossible.

We achieved the best bandwidth of 5.35 GByte/s with 128 kB chunks transmitted via an 1 MB buffer (marked in bold). However, results near this hot spot only vary by a few percents. One very distinct feature of the heat map is the diagonal line, where the buffer size is equal to the chunk size. This has the effect that the reader can only start reading when the writer has finished the current chunk and subsequent chunks can only be written when the reader has finished reading this chunk. This effectively turns the buffer into a locking mechanism with mutual exclusion, greatly reducing the bandwidth.

Results: The figure also distinctly shows the cache sizes of the processor (cf. Section V-A), with a slight performance drop for chunk sizes exceeding the 256 kB Level 2 cache of our system and a bigger performance drop when exceeding the 25 MB Level 3 cache. In conclusion, we use chunk sizes fitting completely into the L2 cache and never exceeding the L3 cache. Transmission buffer sizes are harder to recommend, since this strongly depends on the workload. Without inherent data requirements, one should use a buffer size of approximately $5\times$ to $10\times$ the used chunk size.

IV. REMOTE MESSAGING

In this section we discuss L5’s implementation of a high-performance message buffer in shared remote memory. We found that RDMA has non-trivial performance characteristics that need to be taken into account. For the implementation decisions, we first evaluate different RDMA communication building blocks in microbenchmarks and then use these to construct our messaging implementation. Furthermore, we implement an efficient way to serve multiple remote clients accessing a database server in a request-reply pattern.

A. User-Space TCP Is Not Enough

To validate that RDMA is the right technology to use, we first tried replacing the server’s TCP stack with user-space networking like `mTCP` and saw, that this does not significantly improve response time. To get a performance baseline, we measure the throughput of synchronous 64 Byte messages over

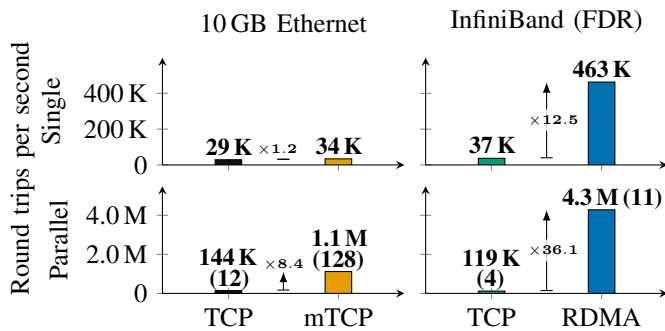


Fig. 7. Remote Throughput. Synchronous transmission of 64 Byte messages between one server thread and a single client resp. an optimal number of clients (in parenthesis).

TCP, mTCP, and RDMA. Figure 7 shows the number of message round trips per second for this workload (hardware details in Section V-A with a DPDK compatible NIC): All TCP based configurations transmit less than 40 Kmsgs/s. RDMA can transmit significantly more messages per second, which makes its performance comparable to the throughput of modern data management systems.

The issue is similar when moving to a multi client scenario. Figure 7 also shows results of an experiment with an optimal number of clients (in parentheses). We used one server, running a single threaded RDMA endpoint and one client with multiple threads to determine the peak message throughput of the server. We also noticed, that TCP over InfiniBand has less overall throughput, despite it being the faster fabric. The results show that already a few RDMA clients can move the bottleneck to a single threaded server.

B. RDMA Design Decisions

RDMA and RoCE offload most the network stack processing from the processor onto the NIC to reduce CPU load. Recent work [21] has shown that the fraction of CPU time spent processing the network stack can be up to 80%. RDMA can eliminate the overhead with hardware support for reliable transmission of data over RC connections. Additionally, RDMA and RoCE bypass the operating system kernel and allow the applications to talk directly with the NICs, thus avoiding costly context switches.

C. Optimizing RDMA for Small Messages

We highlight two design decisions in L5’s use of the IB verbs interface:

Request Polling: We compare different ways of using RDMA primitives to transfer fix-sized messages between two machines in Section IV-C. In this experiment, the client machine sends a message to the server. Once received, the two machines switch roles and the process is repeated. The three “Write” approaches shown use a RDMA *write* work requests to place data directly into the server’s memory. They differ in the way the server is notified about the message’s arrival: In the “Polling” case, we write data with a single write request and a busy loop constantly polls front and back of the incoming memory location to detect when transmission is finished. This approach relies on

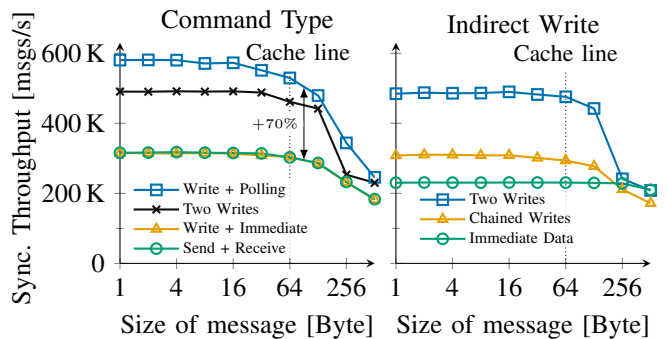


Fig. 8. Efficient RDMA Usage. We investigate RDMA primitives for small messages: The best IB verbs and the most efficient way to write to two locations.

a front-to-back write order within one write request. While this behavior has been documented for all hardware RDMA implementations [15], [33], [24], software implementations following RFC 5040 [40] might have a less strict write order. As an alternative, we measure a “Two Write” implementation that issues one write request for the bulk of the data and a second single-byte request for RFC 5040 compliant message detection. In the “Immediate” case, we attach a so called immediate data value to the *write* work request. The immediate value transfers 4 Byte of data outside the actual message and is propagated to the receiver’s completion queue. In this case, the server polls the completion queue instead.

The last approach “Send + Receive” uses *send/receive* work requests to exchange messages. Just like in the immediate case, we consistently poll the completion to reduce the latency as much as possible. The experiments clearly show that this is necessary to avoid the additional lookup in the completion queue to achieve high message throughput rates. With larger messages this overhead becomes less relevant, because transferring the actual message becomes expensive.

We base L5’s implementation on polling a single *write* work request, which most efficiently uses the hardware capabilities for small message sizes. For our target message size of around 100 Byte sized messages, we get around 70% faster synchronous throughput compared to using *receive* requests.

Message Delivery: To support a special mode for multiple clients, L5’s messaging implementation requires two RDMA writes per message: One to set an indicator flag that a new message arrived and one containing the actual message (details in Section IV-D). In Section IV-C we compare different techniques for doing two consecutive RDMA write operations. First, we use two *write* work requests and send these individually to the NIC. The first one writes the actual data and the second one sets the indicator flag. Due to the ordering guarantees of RDMA, the message is completely written before the flag is set. Next, we use the chaining feature of RDMA work requests, which allows creating a list of work request that can be sent to the NIC with a single function call. Lastly, we make use of the immediate data feature again, by putting the indicator flag into the immediate data value.

Our results show that chained work requests cause a surprisingly large overhead, even though they execute fewer

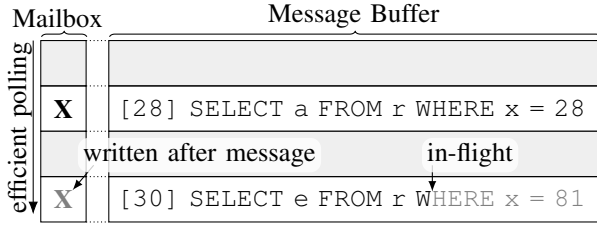


Fig. 9. Server Message Buffer. The mailbox can be efficiently polled, since the indicator flags are packed in memory. The RDMA write order guarantees that no in-flight messages are detected.

instructions on the CPU. The same holds for the immediate data value, which forces the server to poll the completion queue instead of directly polling the indicator flags. In result, it is advisable to use the first technique, which simply uses two distinct RDMA write request.

D. Implementation

Based on the previous findings, we implement a solution matching our goals laid out in Section II. The RDMA connection is initially bootstrapped with out-of-band communication channel over TCP. This also allows upgrading existing connections after authentication and RDMA capability detection phases. The out-of-band channel is then used to communicate locations of the mailbox flag and message buffer. The control channel can also be used to adjust the initially fixed buffer sizes for each client. When a client requests a larger buffer, the server reallocates this client’s message buffer and transmits the new location. This should be a rare case in transactional workloads and therefore not influence the steady state performance.

For the remote case, L5 supports an asymmetric connection behavior, i.e., one database server that is serving small sized requests from many clients. This is useful for a common pattern we observed, where a database server can have many open connections, but only a few are active in bursts (e.g. when an ORM reads an object hierarchy). In related work, Chen et al. [12] identified CPU cache efficiency as a contention point for inbound messages. L5 therefore implements a cache efficient polling mechanism for the server-side:

Client → Server: Figure 9 shows the memory layout on the server side with two distinct memory regions. Each row in the message buffer on the right represents the receive-buffer for one client. Each corresponding “mailbox” flag on the left indicates whether this row’s client has written a new message into the “message buffer”. As described in Section IV-C, we use two RDMA *write* work requests: The first one writes the message and the second one sets the mailbox flag. Due to the ordering guarantees of reliable RDMA connections, the message data is guaranteed to be completely written before the flag is set and thus, the server can never see incomplete messages. In the example in Figure 9, the second client has completed sending a message and thus already set the mailbox flag. The fourth client still has a message “in-flight”, without the mailbox flag set.

The separation of messages from indicators for available messages in the “mailbox” allows efficient polling for incoming messages. This dense indicator buffer is possible, because RDMA allows writes of single bytes. Directly polling the message buffer would cause increased latency because of additional cache misses. L5’s continuous mailbox array has optimal cache locality, which allows polling 64 client connections with a single cache line. Additionally, the server can use SIMD instructions to efficiently poll the mailbox. Whenever it encounters a set flag, it handles the message, clears the flag, and sends a reply. Once the client has received a reply, it knows that it is safe to send the next message.

Server → Client: In the other direction, we assume to receive only answer messages from a single source, due to the asymmetric relationship between client and server. Therefore, L5 can use an optimized layout that requires only a single RDMA write request per message:



The first field [10] is the message’s size and is always transmitted as a 4 Byte integer value. The client waits for a message, by constantly polling this memory address. Once it reads a value different from zero, it detects the start of a new message. The server appends an additional byte [OK] after the actual message. Once this [OK] byte is set, the RDMA RC write order guarantees that the message has been completely transmitted. A second, validating read of the message size detects torn writes. This structure resembles the buffer in Shared Memory, as it supports arbitrary sized result sets. Typically, requests are small or even fixed size, but transaction results might be larger than expected and consist of multiple messages. This way, the buffer can seamlessly handle typical workloads.

Apart from the efficiently using RDMA primitives, our implementation benefits from three additional optimizations: (1) A virtual memory wrap-around mapping similar to the local ring buffer reduces the total amount of writes. It allows to always use a single, continuous, and unconditional write, which reduces worst-case latency. (2) Common RDMA optimization techniques, such as using inline messages for small payloads and selective signaling of verb completion reduces overhead. (3) Eager, asynchronous *reads* of the remote read position allow single RTT writes in the common case.

V. EVALUATION

So far, we justified the design of L5 primary with microbenchmarks, and Figure 1 showcases the overall performance impact of low-latency communication on an in-memory database system running the TPC-C benchmark. In the following, we first discuss the experimental setup, then evaluate L5 with a lightweight workload that is sensible for the network bottleneck, and compare our implementation to popular DBMSs. Finally, we compare the RDMA implementation of L5 to two state-of-the-art communication frameworks.

A. Hardware Details

We conducted our experiments on two dual socket machines equipped with Intel Xeon E5-2660 v2 processors running at

2.2 GHz. The machines have 256 GByte of main memory and are organized as NUMA systems with 128 GByte per socket. Both machines are equipped with a Mellanox ConnectX-3 VPI NIC, which supports FDR InfiniBand with 56 GBit/s, and are connected via a Mellanox SX6005 switch. To avoid NUMA effects, which are not the focus of our work, we run our experiments exclusively on the socket that is directly connected to the network card.

B. Yahoo! Cloud Service Benchmark

As an end-to-end workload, we use the Yahoo! Cloud Service Benchmark (YCSB) [13]. YCSB is a simple key-value store workload, which uses one table with a 4 Byte key and 10 string fields with 100 Byte each. It defines CRUD-style operations, but since we are focused on the network we only use the read workload YCSB-C. Each transaction consists of the following steps: First, the client generates and sends a randomized, valid lookup key using a Zipf distribution [19] with $z = 1$. Once received, the server queries its key-value store and returns one of the string fields to the client.

The in-memory DBMS Silo achieves around 1 million YCSB-C lookups per second on a single thread without communication overhead. For the network-centric evaluation, we send prepared-statement messages via L5 to Silo. The benchmark of Figure 1 already demonstrated that changing the underlying communication layer using L5 can significantly improve the network bottleneck.

C. Software Setup

We compare our own implementation to state-of-the-art commercially available DBMSs. DBMS X uses the ODBC API [17] and supports three different connection options on Windows: TCP, Shared Memory, and Named Pipes. We consider it the most advanced implementation of a Shared Memory, client-server database connection. L5 is designed for Linux, where DBMS X’s only available connection option is TCP. Therefore, we conduct local DBMS measurements on Windows in addition to Linux. Networked experiments were measured between two Linux machines.

We also include MySQL [5], since Raasveldt and Mühleisen [39] measured very promising serialization times. In our measurements, we used its Connector/C (libmysqlclient). By using each database’s native client library, we achieve maximum performance, since the native libraries use the communication protocol with the least overhead. All tested databases also provide ODBC connectors, which would be significantly easier to test, but are usually implemented as a wrapper of the native libraries used in our experiments. PostgreSQL [48] is another interesting competitor, since many other systems implement and support its protocol. To measure it, we used the native client library libpq. Additionally, we compare SQLite as an in-process database without the communication between processes.

All database systems use prepared statements with placeholders to reduce message size and avoid SQL parsing overhead. In case of Silo, we transmit a structure specifying the prepared statement ID and the placeholder value to routines written in

TABLE I
LOCAL YCSB WORKLOAD C THROUGHPUT. COMPARISON OF THE LOCAL SYNCHRONOUS THROUGHPUT OF DIFFERENT DATABASES. TESTED CONNECTIONS: TCP, SHARED MEMORY (SHM), NAMED PIPES (NP), DOMAIN SOCKETS (DS), AND LOOPBACK RDMA.

[sync. tx/s]	TCP	SHM	NP	DS	RDMA
Silo + L5	50.5 K	685 K	—	72.1 K	364 K
DBMS X*	7.56 K	11.5 K	11.5 K	—	—
MySQL*	10.0 K	45.9 K	27.6 K	—	—
DBMS X†	6.88 K	—	—	—	—
MySQL†	25.0 K	—	—	42.9 K	—
PostgreSQL†	11.3 K	—	—	18.4 K	—
SQLite†	—	378 K	—	—	—

C++. For the other systems, we use their native SQL capabilities to execute the prepared statement.

D. Local Measurements

To evaluate L5’s Shared Memory implementation (Section III), we use a single machine and compare against locally available connection options. For this setup it is also possible to use library database management systems such as SQLite, which does not have a dedicated database connection but instead uses regular function calls to access data. What makes those systems undesirable is that there are a number of ways, e.g. memory corruption bugs, in which the host process can corrupt the database². Our approach instead uses a dedicated one-to-one connection in order to prevent bugs in the application to break the database system’s ACID guarantees.

Throughput—YCSB Workload C: Table I shows a comparison of different systems’ synchronous transactional throughput. In this experiment, we compare the locally available connection types. Shared memory—where available—gives the best performance. Other alternatives like Named Pipes or Domain Sockets are consistently faster than the link-local TCP baseline. Those results show that the connection technology can greatly limit the throughput of the application. Even the best results of traditional database connections are still orders of magnitude slower than we would expect. This shows that it is worthwhile to rewrite the network stack and have a dedicated implementation for local communication. In the measurements of Silo with L5, we additionally include the result for RDMA in local loop-back mode. This implementation is very similar to Shared Memory, but suffers from the round trip over the PCIe bus to the NIC and can only reach around half of the SHM performance.

Silo in combination with L5 is consistently faster than that of other databases, but we can also observe significant performance differences between DBMSs, with MySQL having a relatively good network stack. Still, SQLite’s in-process, no-communication transactions outperform MySQL by an order of magnitude. We achieve the overall best performance with L5’s Shared Memory implementation, performing at 15× compared to MySQL and even outperforming SQLite’s in process implementation.

²<https://www.sqlite.org/howtocorrupt.html>

*on Windows Server 2016

†on Ubuntu 18.04.1

TABLE II
LOCAL YCSB TABLESCAN BANDWIDTH. COMPARISON OF LOCAL BANDWIDTH OF DIFFERENT DATABASES OVER TCP, SHARED MEMORY (SHM), NAMED PIPES (NP), AND DOMAIN SOCKETS (DS).

[MByte/s]	TCP	SHM	NP	DS
Silo + L5	257	274	—	261
DBMS X*	105	511	518	—
MySQL*	27	28	29	—
DBMS X [†]	186	—	—	—
MySQL [†]	508	—	—	439
PostgreSQL [†]	148	—	—	256
SQLite ^{†3}	—	711	—	—

Bandwidth—YCSB Tablescan: Table II shows bandwidth measurements for local communication channels. All systems are far from reaching the theoretical bandwidth limits, having over an order of magnitude headroom to the measured baseline in Section III-B. For Silo + L5, we can observe that the underlying connection technology only has a minor influence on the tablescan bandwidth.

However, the mediocre tablescan bandwidth of Silo is not caused by the network stack, but seems to be an inherent limitation of its OLTP focused design. DBMS X’s bandwidth can be significantly improved using Shared Memory, but it is still an order of magnitude slower than what is theoretically possible. In comparison, MySQL has consistently poor performance on Windows, but performs 10× better on Linux. Given its quite good transaction throughput using Shared Memory on Windows, we suspect that this is a performance regression in the current release.

Discussion: In most systems, we can observe sizable YCSB-C performance variations from just changing the underlying connection. But even DBMS X and MySQL, which both support SHM under Windows, are still over an order of magnitude off of the expected performance. Since DBMS X is able to reach over 2 million operations in an internal T-SQL loop, we suspect that this is caused by an incomplete operating-system bypass. Instead of detecting incoming messages directly through SHM, MySQL uses the Windows’ named event API for notifications, which apparently has significant overhead.

The measurements of local tablescan bandwidth also show significant differences between systems. Maybe as a surprise, no tested DBMS can come close to saturating the available bandwidth. While this might be caused by inefficient serialization formats, as Raasveldt and Mühleisen [39] suggest, we believe that many more aspects of system design play a role. E.g., Silo’s storage and transaction implementation is optimized for small and local accesses, but turns out to be a bad choice for larger range-scans. Nevertheless, for DBMS X the used data transport has high impact of reachable bandwidth and SQLite’s in-process bandwidth shows that the network interfaces are still lacking.

E. Remote Measurements

In Section IV, we conducted performance measurements between two servers over TCP and RDMA. Since we focus on client server communication, we concentrate on a single (one-

TABLE III
REMOTE YCSB WORKLOAD C THROUGHPUT. COMPARISON OF SUPPORTED REMOTE COMMUNICATION TECHNOLOGIES.

[sync. tx/s]	1 G Eth	56 G IB	RDMA
Silo + L5	15 K	27 K	302 K
DBMS X	3.1 K	3.7 K	—
MySQL	7.1 K	8.0 K	—
PostgreSQL	6.3 K	7.5 K	—

TABLE IV
REMOTE TABLESCAN BANDWIDTH. COMPARISON OF SUPPORTED REMOTE COMMUNICATION MODES.

[MByte/s]	1 G Eth	56 G IB	RDMA
Silo + L5	99	227	266
DBMS X	111	76	—
MySQL	111	327	—
PostgreSQL	97	140	—

to-one) connection. The concept of scaling to many connections is somewhat orthogonal (cf. Section V-F) and benefits from efficient individual connections.

Throughput—YCSB Workload C: Table III shows a comparison of remote synchronous transaction throughput over different network connections. L5’s results in this experiment are similar to our microbenchmarks introduced earlier: To utilize the performance of modern database systems (or any network application with high message rates), it is necessary to migrate to RDMA-based communication.

While upgrading the network hardware can already scale the performance without any software modifications, most systems show only minor improvements. Using RDMA-aware messaging gives, similarly to SHM, over an order of magnitude performance improvement.

Bandwidth—YCSB Table Scan: A full fetch of the YCSB table amounts to about 1 GB payload data over the network, which the ODBC 3.8 interconnect used by DBMS X transmits using paged data block cursors. For our implementation using L5, we use a similar approach and fetch blocks of 128 kB (same as in Section III-B).

As Table IV shows, most databases somewhat profit from the available bandwidth of the faster InfiniBand network. Slow networks limit the overall throughput, i.e., the slowest configuration with TCP over Gigabit Ethernet (1 G Eth). Surprisingly, DBMS X is even slower with TCP over InfiniBand, which might be caused by the computational overhead of the translation layer (we previously observed it being sensitive in Table II). No implementation even closely reaches the theoretical maximum of 7 GByte/s.

Discussion: The bandwidth measurements also make the protocol overhead visible. When bandwidth is limited by the Gigabit Ethernet fabric, we can observe a direct impact of serialization format’s size overhead on throughput. E.g. PostgreSQL’s is known to have high overhead.

When switching to InfiniBand, the database systems instead run into processing limitations. Most systems already reach the same bandwidth as link-local TCP, while we could reach much higher bandwidth with L5’s RDMA implementation. In a sepa-

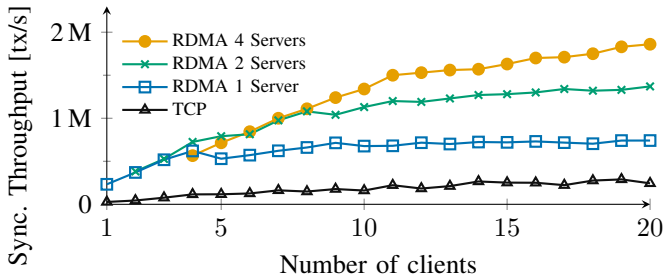


Fig. 10. Silo + L5 Scale Out. Comparison of RDMA and TCP performance with a growing number of client threads (YCSB-C, read-only, 10M tuples).

rate experiment, we determined this to be about 1.22 GByte/s. We want to point out that this is reasonably close to the maximum single stream communication an application can only use at $1 \times \text{FDR}$ signaling rate (1.75 GByte/s). Our InfiniBand installation uses a total of $4 \times$ link aggregation, which results in the nominal data rate of $4 \times 1.75 \text{ GByte/s} = 7 \text{ GByte/s}$. However, the nominal data rate can only be saturated with multiple parallel streams, but RDMA over FDR InfiniBand can be used to mitigate this situation.

F. Scale Out

Figure 10 shows the scale-out behavior of client connections via different technologies. We run YCSB-C on Silo with L5 and increase the number of clients on the horizontal axis.

With this experiment, we can observe that both TCP and RDMA can scale to some degree, but RDMA has a huge head start. The TCP based implementation scales moderately from 1 to 20 clients, reaching about 300k transactions per second. RDMA already surpasses 300k transactions per second with 2 clients, before saturating a single server with 4 clients. An increasing number of server threads allows scaling to even more clients, peaking at around 1.9M transactions per second with 4 server threads and 20 clients.

The 4 server threads scale linearly up to 10 client connections, at which point we reach some limits of our system where the clients start to run on hyperthreads. When scaling to even more clients, some related work raises concerns due to dedicated packet queues of RC connections. To evaluate this, we run a similar experiment with 200 open connections. There, we measure a 5% overhead for a $10 \times$ increase in open connections, which is significantly less than the overhead of using *receive* requests, which we measured in Section IV-C.

G. Communication Frameworks

In the previous sections, we looked at the communication of commercially available databases, which is slower than L5 by over an order of magnitude. Related work also offers general purpose communication frameworks targeted at high-performance network needs. They either use DPDK for user-space networking [46], [23] or directly support RDMA [1], [49], [28], [23]. In the following, we compare L5 against two promising implementations: Seastar [46] using DPDK and eRPC [23] with RDMA support.

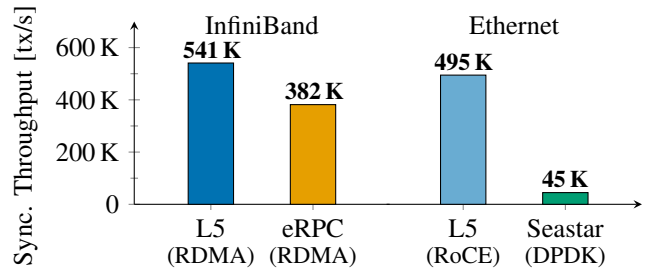


Fig. 11. Communication Frameworks. Comparison of YCSB-C throughput over communication frameworks. L5 and eRPC use RDMA, while Seastar only supports DPDK.

These communication frameworks differ in applications and target use-case from L5. For example, they commonly report their performance numbers with batched messages, i.e., sending 8 or more messages in one transmission to distribute the communication overhead over multiple messages. We do not consider batching a part of the communication interface, but rather a responsibility of higher-level frameworks, such as the ODBC driver. In addition, they also do not provide a shared memory interface, thus have only limited performance for link-local database clients, i.e., multiple containers colocated on one container host would need to incur full network overhead.

The hardware configuration for this experiment is largely unchanged. One exception is the network fabric, where we can not consistently use InfiniBand due to limited DPDK support. Instead, we change our setup to use the same NICs, but without the intermediary InfiniBand switch, which restricts the configurable network fabric. With this change, we can run the experiment either over Ethernet (DPDK and RoCE) or InfiniBand (RDMA). Additionally, since the callback based programming interfaces of eRPC and Seastar are not easily integratable into Silo we run this experiment with a simple custom hashtable instead. Those changes cause this experiment to be not comparable to the previous experiments with a significantly higher throughput.

Seastar: The communication framework in ScyllaDB [45] is designed for extreme scalability. Seastar’s architecture is built upon asynchronous programming with lightweight threads and a custom network stack on top of DPDK, which enables it to scale to multiple thousands of connections. They showcase their performance with dual 40G Ethernet NICs, where they serve 7M HTTP requests per second to 2048 clients, each with multiple concurrent connections.

eRPC: Instead of treating network messages as a stream of bytes, eRPC implements messages as *remote procedure calls*. This design follows similar reasoning to ours and should be a good fit for transaction throughput. Unlike L5, eRPC uses UD *send/receive* operations, which they argue to be more scalable.

In our evaluation of eRPC, we used the same hard- and software configuration as for L5, with RDMA and RoCE using the standard Mellanox drivers for our NICs. In eRPC’s own evaluation, they use a modified driver with no overflow and invalid opcode checks, removed unused features, and disabled locks ensuring thread safety.

Results: To compare the communication frameworks, we run the YCSB-C workload as before, with one concurrent in-flight transaction and no batching. Since neither of the two frameworks supports Shared Memory, we limit our evaluation to the remote case. Figure 11 shows that RDMA implementations significantly outperform Seastar. This is rather unsurprising, since Seastar uses a custom TCP/IP implementation and performs as expected from our microbenchmarks in Section IV. L5 outperforms eRPC by around 42%, which is caused by their decision to use *send/receive* instead of direct memory polling. In this experiment, we can also see that the choice of underlying network fabric is actually less important than the use of direct memory communication and a performant implementation thereof. Using L5 over Ethernet (RoCE) only reduces performance by around 10% and still outperforms both other communication frameworks.

Both Seastar and eRPC trade individual client’s performance for better scale out behavior. Instead, we argue that it should be possible to saturate a system already with few clients, especially when running them on the same machine. This is exactly the setup where L5 shines: *Optimal performance for each client.*

VI. RELATED WORK

High-speed networking hardware (RDMA over InfiniBand) has already been widely adopted in research and industry to improve the performance of data management systems. Due to the obvious benefits of faster network fabric, a large body of work in the database community adopts RDMA into the systems. Many papers focus on high-performance distributed data structures as a basis for storage engines. In addition, we see many advancements in distributed query and transaction processing powered by the use of RDMA. However, one important aspect that has been largely overlooked is the improvement of the communication layer. In short:

How do we get requests into a database engine?

In the following, we compare our work with existing research on client-database communication and give an overview of other areas in database systems where RDMA has been applied.

Network protocols: Raasveldt and Mühleisen speed up the communication of database systems with client applications [39]. They argue that de-/serialization of result sets dominates execution time for OLAP workloads and propose a more efficient columnar serialization method. In contrast, our work studies the effects of replacing the underlying network protocol of database systems to improve performance. Both are important, but orthogonal efforts towards the same goal and might have an optimal effect when combined.

Another effort for optimizing the client-database communication is MICA [30], a high-performance key-value store with an optimized network stack. Their network layer uses direct NIC access (kernel bypassing; similar to mTCP). Like our implementation, MICA avoids the overhead of TCP, but implements a custom communication protocol optimized for small key-value items. The RDMA part of our paper extends their idea by adopting it for high-performance interconnects and provides a deep analysis of the involved techniques.

Distributed data structures: There is a large body of work building data structures on RDMA [52], [22], [24], [55]. We present three representative ones in the following:

FaSST [25] is a key-value store that efficiently uses RDMA for small messages. It implements remote procedure calls (RPC) using RDMA *send/receive* requests instead of directly reading or writing memory. In combination with message batching techniques to reduce NIC to CPU communication, their system perfectly scales for parallel workloads. FaSST uses RDMA in unreliable datagram mode, which they reason to be reliable due to extremely rare packet losses. In contrast to their work, we show that competitive latencies are possible without batching, while additionally providing reliable communication channels.

FaRM [15] uses a message passing approach that uses RDMA *writes* (similar to ours) for update transactions. Their system also uses a ring buffer and a message detection approach polling the memory location of the next message. For read requests, the client traverses server side data structures using RDMA. In contrast to our work, their approach requires multiple round trips to traverse remote data structures and is more difficult to extend to full-fledged transactions.

Pilaf [36] is a cuckoo hash table with self-verifying data structures that can detect read-write races without client-server coordination. Clients directly read from the server’s memory via RDMA read operations. While their self-verifying feature is promising, remote traversal of the hash table causes a significant latency overhead.

Query and transaction performance: Classic database workloads can also profit from using RDMA and InfiniBand [50], [32], [4], [16]. For join processing, Barthels et al. [7], [6] investigate and optimize the scale out behavior of radix joins for very large InfiniBand clusters. Other research has focused on join performance by taking advantage of data distribution [43] and skew [41]. Alonso et al. [4] propose a high-level API for data flows. They simplify throughput-oriented, OLAP-style applications rather than latency, which is important for OLTP.


Beyond joins, RDMA is beneficial for extending local main memory capacity and thus avoids spilling on slow local disks [29], [37]. Building on the huge bandwidth of RDMA, MonetDB has been distributed by a ring topology and continuously rotating data [18]. Rödiger et al. [42] use a distributed exchange operator that can extend existing systems and scale. While it is notoriously difficult to scale distributed transaction processing [54], using RDMA made large advancements in largely partitioned workloads [26]. Abstractions like Network-Attached-Memory (NAM) [44], [8], [9] or partitioning [34] have been proposed. They separate a distributed database into compute and storage nodes, interconnected via RDMA. Using this architecture, recent research suggests RDMA latency has evolved so far that distributed transactions *can* scale [54].

VII. SUMMARY

We have shown that it is necessary to redesign and improve existing network stacks to fully utilize the performance of modern in-memory database systems. Our experiments suggest that current network implementations are not performant

enough for high transaction rates. L5 addresses the problem of low latency remote and local communication by leveraging RDMA over InfiniBand and Shared Memory.

Using L5 makes the underlying network protocol transparent for the database system. With this approach, we can adaptively choose the best network technology while allowing to integrate new ones without affecting the application itself. In result, L5 provides a single, performant interface for multiple different technologies.

Acknowledgments: This project received funding by the European Research Council (ERC) under the European Union’s Horizon 2020 research and innovation programme (grant agreement No 725286). 

REFERENCES

- [1] Accelio. Accelio the opensource I/O, message, and RPC acceleration library. <http://www.accelio.org/>, 2019.
- [2] Alibaba Container Service. Using RDMA on container service for Kubernetes. <https://www.alibabacloud.com/blog/594462>, 2019.
- [3] G. Allen and M. Owens. *The Definitive Guide to SQLite*. Apress, Berkeley, CA, USA, 2nd edition, 2010.
- [4] G. Alonso, C. Binnig, I. Pandis, K. Salem, J. Skrzypczak, R. Stutsman, L. Thostrup, T. Wang, Z. Wang, and T. Z. 2. DPI: The data processing interface for modern networks. In *CIDR*, 2019.
- [5] D. Axmark and M. Widenius. *MySQL Reference Manual*. O’Reilly, 2002.
- [6] C. Barthels, S. Loesing, G. Alonso, and D. Kossmann. Rack-scale in-memory join processing using RDMA. In *SIGMOD*, 2015.
- [7] C. Barthels, I. Müller, T. Schneider, G. Alonso, and T. Hoefler. Distributed join algorithms on thousands of cores. In *PVLDB*, 2017.
- [8] C. Binnig. Scalable data management on modern networks. In *Datenbank-Spektrum*, volume 18, 2018.
- [9] C. Binnig, A. Crotty, A. Galakatos, T. Kraska, and E. Zamanian. The end of slow networks: It’s time for a redesign. *PVLDB*, 9, 2016.
- [10] E. Burness. Introducing the new HB and HC Azure VM sizes for HPC. <https://azure.microsoft.com/en-us/blog/introducing-the-new-hb-and-hc-azure-vm-sizes-for-hpc/>, 2019.
- [11] S. Chen, A. Ailamaki, M. Athanassoulis, P. B. Gibbons, R. Johnson, I. Pandis, and R. Stoica. TPC-E vs. TPC-C: Characterizing the new TPC-E benchmark via an I/O comparison study. *SIGMOD*, 2011.
- [12] Y. Chen, Y. Lu, and J. Shu. Scalable RDMA RPC on reliable connection with efficient resource sharing. In *EuroSys*, 2019.
- [13] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *SoCC*, 2010.
- [14] D. E. Difallah, A. Pavlo, C. Curino, and P. Cudré-Mauroux. OLTP-Bench: An extensible testbed for benchmarking relational databases. *PVLDB*, 7, 2013.
- [15] A. Dragojevic, D. Narayanan, M. Castro, and O. Hodson. FaRM: Fast remote memory. In *NSDI*, 2014.
- [16] A. Dragojevic, D. Narayanan, E. B. Nightingale, M. Renzelmann, A. Shamis, A. Badam, and M. Castro. No compromises: distributed transactions with consistency, availability, and performance. *SOSP*, 2015.
- [17] K. Geiger. *Inside ODBC*. Microsoft Press, 1995.
- [18] R. Goncalves and M. L. Kersten. The data cyclotron query processing scheme. In *TODS*, number 4, 2011.
- [19] J. Gray, P. Sundaresan, S. Englert, K. Baclawski, and P. J. Weinberger. Quickly generating billion-record synthetic databases. In *SIGMOD*, 1994.
- [20] J. Iyengar and M. Thomson. QUIC: A UDP-based multiplexed and secure transport. Internet-Draft draft-ietf-quic-transport-16, Internet Engineering Task Force, Oct. 2018. Work in Progress.
- [21] E. Jeong, S. Woo, M. A. Jamshed, H. Jeong, S. Ihm, D. Han, and K. Park. mTCP: a highly scalable user-level TCP stack for multicore systems. In *NSDI*, 2014.
- [22] J. Jose, H. Subramoni, M. Luo, M. Zhang, J. Huang, M. Wasi-ur-Rahman, N. S. Islam, X. Ouyang, H. Wang, S. Sur, and D. K. Panda. Memcached design on high performance RDMA capable interconnects. *ICPP*, 2011.
- [23] A. Kalia, M. Kaminsky, and D. Andersen. Datacenter RPCs can be general and fast. In *USENIX NSDI*, Boston, MA, 2019. USENIX Association.
- [24] A. Kalia, M. Kaminsky, and D. G. Andersen. Using RDMA efficiently for key-value services. In *SIGCOMM*, 2014.
- [25] A. Kalia, M. Kaminsky, and D. G. Andersen. Faszt: Fast, scalable and simple distributed transactions with two-sided (RDMA) datagram RPCs. In *OSDI*, 2016.
- [26] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. B. Zdonik, E. P. C. Jones, S. Madden, M. Stonebraker, Y. Zhang, J. Hugg, and D. J. Abadi. H-store: a high-performance, distributed main memory transaction processing system. In *PVLDB*, 2008.
- [27] S. T. Leutenegger and D. M. Dias. A modeling study of the TPC-C benchmark. In *SIGMOD*, 1993.
- [28] C. Lever. Network file system (NFS) upper-layer binding to RPC-over-RDMA version 1. *RFC*, 8267, 2017.
- [29] F. Li, S. Das, M. Syamala, and V. R. Narasayya. Accelerating relational databases by leveraging remote memory and RDMA. In *SIGMOD*, 2016.
- [30] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky. MICA: A holistic approach to fast in-memory key-value storage. In *NSDI*, 2014.
- [31] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. orn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg. Meltdown: reading kernel memory from user space. *USENIX Security*, 2018.
- [32] F. Liu, L. Yin, and S. Blanas. Design and evaluation of an RDMA-aware data shuffling operator for parallel database systems. In *EuroSys*, 2017.
- [33] J. Liu, J. Wu, S. P. Kini, P. Wyckoff, and D. K. Panda. High performance RDMA-based MPI implementation over infiniband. *ICS*, 2003.
- [34] S. Loesing, M. Pilman, T. Etter, and D. Kossmann. On the design and scalability of distributed shared-data databases. In *SIGMOD*, 2015.
- [35] J. D. McCalpin et al. Memory bandwidth and machine balance in current high performance computers. In *TCCA*, 1995.
- [36] C. Mitchell, Y. Geng, and J. Li. Using one-sided RDMA reads to build a fast, CPU-efficient key-value store. In *USENIX ATC*, 2013.
- [37] H. Mühleisen, R. Goncalves, and M. L. Kersten. Peak performance: Remote memory revisited. In *DaMoN*, 2013.
- [38] A. Pavlo. What are we doing with our lives?: Nobody cares about our concurrency control research. In *SIGMOD*, 2017.
- [39] M. Raasveldt and H. Mühleisen. Don’t hold my data hostage - A case for client protocol redesign. In *PVLDB*, volume 10, 2017.
- [40] R. Recio, B. Metzler, P. R. Cullley, J. Hilland, and D. Garcia. A remote direct memory access protocol specification. *RFC*, 5040, 2007.
- [41] W. Rödiger, S. Idicula, A. Kemper, and T. Neumann. Flow-Join: Adaptive skew handling for distributed joins over high-speed networks. *ICDE*, 2016.
- [42] W. Rödiger, T. Mühlbauer, A. Kemper, and T. Neumann. High-speed query processing over high-speed networks. In *PVLDB*, volume 9, 2015.
- [43] W. Rödiger, T. Mühlbauer, P. Unterbrunner, A. Reiser, A. Kemper, and T. Neumann. Locality-sensitive operators for parallel main-memory database clusters. In *ICDE*, 2014.
- [44] A. Salama, C. Binnig, T. Kraska, A. Scherp, and T. Ziegler. Rethinking distributed query execution on high-speed networks. *IEEE Data Eng. Bull.*, 2017.
- [45] ScyllaDB Inc. ScyllaDB is the real-time big data database. <https://www.scylladb.com/>, 2019.
- [46] ScyllaDB Inc. Seastar high performance server-side application framework. <http://seastar.io/>, 2019.
- [47] L. Soares and M. Stumm. FlexSC: flexible system call scheduling with exception-less system calls. In *OSDI*, 2010.
- [48] M. Stonebraker and G. Kemnitz. The postgres next generation database management system. In *CACM*, volume 34. ACM, 1991.
- [49] M. Su, M. Zhang, K. Chen, Z. Guo, and Y. Wu. RFP: when RPC is faster than server-bypass with RDMA. In *EuroSys*, 2017.
- [50] B. Tierney, E. Kissel, D. M. Swany, and E. Pouyoul. Efficient data transfer protocols for big data. In *eScience*, 2012.
- [51] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden. Speedy transactions in multicore in-memory databases. In *SOSP*, 2013.
- [52] Y. Wang, L. Zhang, J. Tan, M. Li, Y. Gao, X. Guerin, X. Meng, and S. Meng. HydraDB: a resilient RDMA-driven key-value middleware for in-memory cluster computing. In *SC*, 2015.
- [53] X. Yu, A. Pavlo, D. Sánchez, and S. Devadas. Tictoc: Time traveling optimistic concurrency control. In *SIGMOD*, 2016.
- [54] E. Zamanian, C. Binnig, T. Kraska, and T. Harris. The end of a myth: Distributed transactions can scale. In *CoRR*, 2016.
- [55] T. Ziegler, S. Tumkur Vani, C. Binnig, R. Fonseca, and T. Kraska. Designing distributed tree-based index structures for fast RDMA-capable networks. *SIGMOD*, 2019.