A Scalable Distributed Lock Manager using One-Sided RDMA Atomic Operations

by
Sämy Zehnder (zehnders@)

Supervised by
Prof. Dr. Alonso, Jonas Pfefferle

June 2015–December 2015
Abstract

One of the major factors that limits scalability in distributed databases is concurrency control such as locking. Traditionally, lock managers have been designed for local access while distributed locking was implemented on top with the assumption of slow networks. Recently, there has been an increasing interest for Remote Direct Memory Access (RDMA) in the database community, due to its low latency and high throughput with minimal CPU load.

In this work we design a lock manager around RDMA atomic operations, shifting the work from the server to the clients. Locks are not preallocated but stored in a hash table which can be directly accessed over RDMA. With this approach we achieve a lock operation latency as low as 6µs and a throughput of up to 1.4M acquire/release cycles per second. Experiments from running TPC-C lock traces show that our approach can achieve up to 2.7M tpmC.
Acknowledgments

First of all, I am deeply grateful for the support I received from various people during my thesis. Thank you Prof. Dr. Gustavo Alonso, Jonas Pfefferle and Patrick Stuedi for your assistance and guidance. Your valuable inputs led me through this project and were essential for its success.

Special thanks also go to you, Luc Humair and Sabine Eichenberger. You were willing to sacrifice many hours of your spare time to add comments and suggestions to earlier versions of this document.
## Contents

1 Introduction ........................................... 4

2 Background ............................................ 5

3 Design and Implementation ......................... 10
   3.1 Overview ........................................... 10
   3.2 Distributed Mutex ................................. 10
      3.2.1 CAS ........................................... 11
      3.2.2 FADD ......................................... 12
      3.2.3 Comparison ................................... 14
   3.3 Distributed Hash Table ............................ 15
      3.3.1 Components ................................... 15
      3.3.2 Acquiring a lock ............................... 19
      3.3.3 Releasing a lock ............................... 24
   3.4 Free-list ......................................... 27

4 Evaluation ............................................. 29
   4.1 Performance of RDMA operations .................. 29
   4.2 Acquiring and releasing locks ....................... 31
      4.2.1 Single client node ............................. 33
      4.2.2 Multiple client nodes ......................... 35
      4.2.3 Non-uniform lock access rates .................. 37
   4.3 Wakeup time ....................................... 39
   4.4 TPC-C ............................................. 40

5 Related Work ........................................... 46
   5.1 Lock managers ....................................... 46
   5.2 Hash tables ......................................... 47

6 Conclusion ............................................. 48

7 Future Work ............................................ 48
   7.1 Lock upgrades ....................................... 48
   7.2 Improving local accesses ......................... 49
   7.3 Fault tolerance ...................................... 50

Glossary ................................................. 52

References ............................................... 53
1 Introduction

One of the major factors that limits scalability in distributed systems is concurrency control. For example in databases, data integrity is enforced by using ACID transactions. One of the reasons why they do not scale well are locks (cf. [14, 30]). Traditionally, lock managers have been designed and optimized for local access [14, 15, 27] and distributed locking was implemented on top with the assumption of slow networks.

Therefore, distributed databases like Amazon Dynamo [5] and Cassandra [17] are moving away from traditional ACID transactions. However, applications that do not easily fit into the new concurrency models cannot use these databases or have to provide atomicity and isolation guarantees by themselves. This shifts the complexity from the database to the application but does not solve the underlying problem as application programmers are faced with the same challenges.

However recently, there has been an increasing interest for Remote Direct Memory Access (RDMA) in the database community [1, 28, 36]. It provides low-latency and high-throughput networking with minimal CPU load. RDMA also provides remote atomic operations like fetch-and-add and compare-and-swap. Designing a lock manager around these operations has the potential to provide lower latency and better scalability than existing systems, making full concurrency control affordable again.

We aim at designing a Distributed Lock Manager (DLM), leveraging the benefits of one-sided RDMA operations. It should meet scalability demands by moving most of the work from the server to the clients. The main challenge will be to keep a low memory footprint: Having a decent number of locks makes it impossible to maintain them all in DRAM. Hence, ruling out solutions as [2, 22], which rely on previously knowing where locks are stored in memory. Instead, we design a system that dynamically creates and removes lock entries as needed, while keeping the latency low and throughput high.

Designing a versatile lock manager requires the support of different lock modes: Gray et al. [10] names Shared and eXclusive, as well as the lesser known IS, IX and SIX. However, any number of modes should be supported to provide a generic solution for applications where that model does not fit. Two clients requesting the same lock in non-conflicting modes are to be granted simultaneously. Moreover, lock requests have to be served in order of occurrence and need to be eventually granted.
2 Background

2.1 Locking

Concurrent setups create scenarios where multiple competitors might simultaneously want access to a single resource. Ensuring consistent and linearizable histories (cf. [12]) requires synchronization among the competitors. This is usually achieved by a mechanism called locking. During lock acquisition it is checked if the request is compatible with already existing ones. If so, it is granted, and otherwise blocked. Upon release of a lock, waiting competitors are woken up.

Locks come in various flavors: Some only ever allow one competitor entering a critical section to ensure mutual exclusion. This is heavily used in distributed file systems such as NFS (using NLM) or GPFS [31]. Either a whole file or byte ranges are locked to avoid competitors from overwriting each other’s changes. Different lock types loosen that exclusive manner: Read-Write locks allow several competitors to read in parallel, while writers are still kept shielded from others. They achieve more concurrent accesses in situations where writes are rare. Even more sophisticated locks are used in databases. They employ various lock modes with fine-tuned compatibility matrices. By using them at several granularities,\footnote{E.g. permissive-mode locks at table level are combined with stricter row level locks.} very high concurrency of non-conflicting operations is gained while increasing the locking overhead.

2.2 Two-phase locking

In concurrency control, two-phase locking (2PL) describes a locking protocol for parallel running operations. It splits them into two consecutive phases: Locks are only allowed to be acquired in the \textit{growing phase}, while releases are only supported in the \textit{shrinking phase}. The latter is initiated as soon as the first lock is released. Afterwards, switching back to the growing phase is prohibited.

By following this scheme, serializable histories are guaranteed. A history is said to be \textit{serializable}, if operations running in parallel can be moved in time s.t. one is executed after the other without changing their outcome.

2.3 Atomic primitives

Atomic primitives form the fundamental building blocks of locks. They perform a small set of operations to a memory location, while it is guaranteed that no other atomic primitive is able to run in parallel. Thereby, atomicity is usually only guaranteed in respect to that value: Concurrent instructions to different locations are allowed. Atomic
primitives generally require hardware support, and are e.g. part of the instruction set of the CPU. They are sometimes also referred to as read-modify-write operations.

A famous atomic primitive is Fetch-And-Add (FADD). It increases a value by a given amount and returns the old value.

```
uint64_t FADD(uint64_t *value_ptr, uint64_t add) {
    atomic {
        uint64_t value = *value_ptr;
        *value_ptr = value + add;
        return value;
    }
}
```

Listing 1: The FADD instruction.

Another example of an atomic primitive is Compare-And-Swap (CAS). It compares the contents of a memory location with a given compare value. Only if they match, the given swap value is written into the field. In any case, the old value is returned.

```
uint64_t CAS(uint64_t *value_ptr, uint64_t compare, uint64_t swap) {
    atomic {
        uint64_t value = *value_ptr;
        if (value == compare) {
            *value_ptr = swap;
        }
        return value;
    }
}
```

Listing 2: The CAS instruction.

### 2.4 Remote Direct Memory Access

Remote Direct Memory Access (RDMA) enables access from one computer to another’s memory without involving any of the two operating systems. That allows zero-copy transfers, reducing both CPU overhead and latency. RDMA is provided by specialized interconnects such as InfiniBand (a switched fabric network) or RoCE (a network protocol which provides RDMA over Ethernet). They both use specialized RDMA NICs (RNICs) which implement part of the network stack for efficiency reasons in hardware. Alternatively, there are also solutions like iWARP delivering RDMA services built on-top of TCP/IP.

RDMA connections come in two different flavors: They can be reliable where successful transmission is ensured by ACK/NAK packages. Unreliable connections avoid these extra packets, resulting in potential message loss. As a third option, one can use connection-less datagram transmission which avoids maintenance of per-connection state.
in the RNIC.\footnote{Solutions such as iWARP, which are based on TCP/IP, provide reliable connections only.}

To setup a data transfer, mutable memory is pinned and registered with the RNIC. By using Direct Memory Access (DMA) to it, the RNIC completely bypasses the CPU. Several one-sided operations are provided by RDMA to manipulate that memory. \textbf{READS} and \textbf{WRITES} are used to access and change remote DRAM. Additionally, atomic changes are supported by using the \textbf{CAS} and \textbf{FADD} operations.\footnote{These operations are only atomic with respect to the RNIC and not the memory bus in general. I.e., concurrent atomic accesses by the CPU are not synchronized. See Section 7.2 for a discussion.}

RDMA also specifies two-sided \textbf{SEND/RECV} operations. A recipient has to register a receive operation by providing a buffer in memory. Only then a sender is allowed to transmit a message which is pushed into that buffer. The recipient is either informed by interrupt or due to spinning on a flag in memory, which is changed by the RNIC on completion of the operation.

The RDMA specification only mentions basic atomic operations on 64-bit words. However, some Mellanox RNICs support an extended verbs API supporting masked, multi-field atomic operations \cite[Appendix B]{26}. Multi-field operations allow using more than 64 bits. Masked \textbf{CAS} operations enable to compare and swap only part of the value. Masked \textbf{FADD} operations support memory ranges being split into several counters. Thereby, updates are still atomic no matter which counter being modified. We exploit that functionality – if available – in our implementation to improve performance.

### 2.5 Hash tables

One often wants to store values associated with an identifier. During insert and update that key is provided along the data. Later, the identifier is used to search for the value in the store, as well as for removals. Hash tables form a family of such stores. They usually provide preallocated space where either values themselves or pointers referring to the values are stored. Hash tables typically split their storage space into several slots. Further, a hash function is used to map keys to them. The resulting slot is then used as a hint for where the value might be stored. However, the key domain usually is way bigger than the number of available slots. Thus, multiple keys might hash into the same slot. The main difference of hash table types lies in the way they handle such clashes.

The cuckoo hash table \cite{25} is one of this kind. Initially, the whole storage space is split into halves, and slots are defined in both of them. Moreover, for each half an individual hash function is used. This provides every key with two possible slots to reside in. A new entry is placed in either one of them. However, situations may arise where both candidate slots are already occupied by values of different keys. We call this a \textit{cuckoo situation}. To make room for the to be inserted value, the alternative locations of the currently occupying keys are inspected. If one of them happens to be free, the
corresponding entry is moved leaving space for the new one. This is recursively done in
the event of the alternative locations also being full. Eventually, either a free slot or a
circle is found. In the former case, cascading moves ensure an initial slot becoming free.
In the latter, no moves are possible and the table is too full. Either resizing or rehashing
is necessary. Alternatively, the insertion can be marked as failed.

Cuckoo hashing ensures constant insertion, lookup and deletion time. A popular variant
is bucketized cuckoo hashing [29] which provides \( n \) sub-slots per hash location. Cuckoo
relocation is only necessary if all of them are occupied in both locations. Bucketized
cuckoo hashing achieves up to 95-99\% occupancy\(^4\) while the fan-out during relocations
is \( n \)-fold. That reduces the expected number of rounds for a relocation exponentially. In
each round a read to the alternative slots needs to be done. When employing bucketized
cuckoo hashing over RDMA, this requires a remote read in every round which takes one
round-trip on the network. But, the fewer number of relocations comes at the price of
more contention on the hash locations since the hash domain shrinks by a factor of \( n \)
for the same table size.

There is also an option of choosing more than two hash functions: \( d \)-cuckoo hashing [9].
One splits the storage into \( d \) parts and uses \( d \) different hash functions to provide more
potential locations where an entry can be stored. This further increases the occupancy
level of the table before a resize is required. For ‘normal’ cuckoo hash tables the load
factor increases from 80\% to 91\% by choosing \( d = 3 \) [21]. However, it comes at the cost
of having to do more lookups and, in our case, increased synchronization effort on the
hash locations. Since grabbing mutexes over RDMA is expensive, we decided against it
in favor of the bucketized hash variant.

Choosing a good hash function is crucial to end up having a good distribution of the
entries in the hash table. While many hash families are proven to be 2-wise independ-
ent,\(^5\) there is no literature on hash functions ensuring independence from a second hash
function of the same family.\(^6\) We need such a hash family to gain a broad distribution
of the entries in case of a cuckoo situation. In fact, simply using some bits of the key
as the hash functions yields poor results due to the two functions being too dependent.
Thus, if two values happen to be hashed into the same bucket, their alternative buckets
likely match as well. This is bad for cuckoo relocations. Instead, we used the family
defined in [34, Def. 3.9] which yielded an even distribution in our experiments.

\(^4\)How full the table can be in expectation, before a cuckoo situation becomes unresolvable.
\(^5\)Choose \( x, y \) u.a.r.: \( x \neq y \) \( \Rightarrow \Pr[h(x) = h(y)] = \frac{1}{m} \).
\(^6\)Choose \( x, y \) u.a.r.: \( x \neq y, h_1(x) = h_1(y) \) \( \Rightarrow \Pr[h_2(x) = h_2(y)] = \frac{1}{n} \).
2.6 Nomenclature

Locks and Mutexes

Our Distributed Lock Manager (DLM) can be used for concurrency control of an application. The term lock thereby refers to a building block which ensures synchronized access to a shared resource used by that application. Whenever a client tries to use a resource it first has to acquire the corresponding lock by issuing a lock request. Once done, it releases the lock.

Internally, the DLM sometimes also demands mutual exclusive access to its own shared data structures. The used synchronization mechanism is conceptually a lock as well. However, we refer to it as mutex to avoid confusion with the client-facing locks. Mutexes are allocated before shared resource access and deallocated afterwards.\footnote{Most of the time we use the shorter terms allocated and deallocated.}

There is a third type of locks: The nodes of our distributed lock manager run several threads which may need to be synchronized as well. Here we did not enforce a special naming scheme: The terms lock and mutex are used interchangeably. Fortunately, in this work we will never become so detailed to ever reach the point where we would encounter them.

DLM vs. DHT

We refer to the system we introduce in this work as Distributed Lock Manager (DLM). On the other hand, we use the term Distributed Hash Table (DHT) for the data structure which is accessed by clients of the DLM to coordinate lock requests.
3 Design and Implementation

This chapter describes the design of our Distributed Lock Manager (DLM). Further, we give insights in why we chose that exact layout and provide an overview of the implementation where appropriate.

The goals of the DLM are:

- Low-latency, high-throughput management of locks and their requests.
- Clients should use RDMA operations to directly manipulate a shared data structure to ensure scalability. By leveraging one-sided RDMA we moreover completely exclude the server’s CPU from the locking process.
- Locks must be created upon acquisition and eventually be deleted again. Otherwise, their number would quickly exceed current systems’ memory capabilities, or at least prevent other applications from running on the same machine.
- Non-granted lock requests should be woken up to avoid polling of the clients.

3.1 Overview

An instance of the DLM consists of one server and several client nodes (cf. Figure 1). They are inter-connected by a fast network.

Server The server is owner of a set of concurrently accessed resources and employs locks for them.\(^8\) However, it is almost exclusively passive when it comes to clients accessing them. It is only responsible for registering memory that will be modified by the client nodes using RDMA. In some rare situations it has to resolve problems that the clients cannot handle on their own.\(^9\)

Client Client nodes compete for access to locks of a fixed server. They do so by modifying the server’s shared memory and possibly communicate to each other in case of a conflict. They might execute multiple instances that run in parallel – we call them competitors.

3.2 Distributed Mutex

The DLM is accessed highly concurrently by many competitors. They sometimes need to be synchronized s.t. consistent access to the remote data-structures is ensured. For that reason we designed the following two different types of distributed mutexes. Both of them only require an 8-byte value, accessed using the RDMA atomic operations Compare-And-Swap or Fetch-And-Add.

\(^8\)It is not necessary that the resources are co-located with the locks.
\(^9\)E.g., resolving a cuckoo situation or resizing the index.
3.2.1 CAS

Our CAS lock is an implementation of the MCS lock first described by [19]. It achieves mutual exclusion using a minimum number of messages. The shared atomic field is initialized to zero which implies that the mutex is unallocated. Competitor A tries to alloc the mutex and issues $tail = CAS(0, A)$ (cf. Figure 2a). Since the mutex is available, the atomic value is set to $A$ and the returned $tail$ is set to 0. $A$ got the mutex.

The next competitor $B$ issues $CAS(0, B)$. However, he ends up with $tail = A$ since $A$’s id is currently stored in the atomic field, and the swap did not take place. $B$ has to re-try to emplace his id into the atomic field by issuing $tail = CAS(A, B)$. This time the CAS succeeds which is indicated by $tail = A$. The atomic value now contains $B$. This is when $B$ learns that it’s his turn once $A$ deallocates the mutex. Hence, he sends a message to $A$ requesting a notification as soon as $A$ is done using the mutex. He then waits for that notification to arrive. Note that the second CAS issued by $B$ could also fail, if in the meantime either $A$ deallocated the mutex, or another competitor $C$ already has successfully inserted his id into the alloc queue. This requires $B$ to repeat the CAS call until it succeeds.

Eventually, $A$ deallocates the mutex. It might be that the notification request from $B$ did not yet arrive. In this case $A$ issues $tail = CAS(A, 0)$. There are two possible outcomes of that operation: Either $tail = A$, in which case $A$ was the only competitor, and the mutex is now free again. Or $tail = X$, indicating a notification request to eventually arrive. Note, $X$ is the last competitor in the alloc queue and not necessarily the next in line. This is why $A$ has to locally mark the mutex as free and is only able to send the notification message once the request message arrives.
Figure 2: Contention on a distributed (a) CAS, (b) FADD mutex. Node B has to wait for A to dealloc the mutex.

**Improvement using masked CAS** As noted above, B possibly needs to issue many CAS calls if the contention on the mutex is high. B might even be unable to add himself into the alloc queue if he happens to have a higher latency to the server than all other competitors: His first CAS probably fails due to the high contention. But, until he gets the result of that CAS call and is able to re-issue a new one, a faster competitor succeeds to insert himself into the queue. The next call of B will fail again. This might go on forever.

If masked atomic operations are available,\(^\text{10}\) this problem can be avoided. By using a compare mask of zero, *Compare-And-Swap* degrades into a *Swap* operation. It atomically sets the given competitor’s id and returns the old tail no matter what currently is written into the field. That reduces the number of atomic operations per competitor to exactly one. Also, since the swap always succeeds, there is no way that a competitor can starve. Moreover, the smaller number of atomic operations allows for a higher alloc throughput on systems that are throughput-bound by atomic operations.\(^\text{11}\)

### 3.2.2 FADD

Using a ticket-based system forms a second approach to distributed mutexes. Upon allocation, a competitor fetches the next ticket and waits until it is his turn. This is

\(^{10}\)See Section 2.4.

\(^{11}\)See Section 4.1, Mellanox ConnectX-3.
implemented by a 64-bit atomic field that is split into two parts (cf. Figure 3). One represents the ticket that is currently holding the mutex – it gets incremented upon dealloc. The other one is incremented during alloc to get a competitor's ticket id. If a competitor realizes that it is not his turn yet, he periodically polls the holder-counter for an update.

As depicted in Figure 2b, during alloc a competitor issues a FADD +1 to the next ticket id counter. This atomically increments the ticket id, and returns the whole atomic field containing the ticket id of the mutex holder as well as the acquired ticket id. If the two ticket ids match the alloc is granted. Otherwise, the competitor has to wait until the holding ticket id counter gets updated to his acquired ticket id. This is done by periodically polling its value. If a competitor senses that more than two tickets are in front of him he waits some time before issuing the first READ. This avoids too much READ traffic in case of a congested mutex. The wait time is estimated to be the queue length (minus two) times the round-trip time (RTT) on the network.\(^{12}\) It is ensured that at least this much time is required by the others: Each update to the holding ticket id takes half an RTT plus an already in-flight READ, that arrives just after the update, takes another half RTT to arrive at the next competitor. After that timeout the next READ to the holding ticket id is issued, and the wait time is adjusted or the mutex granted if the ticket id matches.

Since the two counters are stored in the same atomic field, special care has to be taken to avoid malicious overflows when increasing them. Fortunately, adding to the counter that is stored in the more significant bits never overflows into the less significant bits, since one does not add ‘1’ but ‘1 \(\ll 32\)’. On the other hand, adding to the less significant counter could in fact overflow into the more significant one. However, we chose the holder ticket to be stored there. During deallocation time it gets incremented to the next ticket id by the current holder of the mutex. Since he knows the current value – his own ticket id – he is able to tell if the add would result in an overflow. If so, an atomic subtraction to zero is issued, avoiding the overflow.

---

\(^{12}\)If we are the next in line we should not wait but repeatably poll to get notified about our grant as fast as possible. Minus two is due to our view of the value being half a RTT old. Right after our read, the release-FADD of the current holder might have arrived at the server. That enables the competitor in front of us to have realized that it is his turn at the same time we got our read. His immediate release of the mutex will arrive another half RTT later at the server. That is when our read will also arrive there, if we are issuing it now.
### 3.2.3 Comparison

The advantage of FADD mutexes over CAS mutexes is that a competitor gets his ticket id using only one atomic operation compared to potentially many for the CAS mutex. This argument gets invalidated when masked atomics are used.

For the CAS mutex the wakeup time is dependent on the latencies between two competitors, and the one between the competitor and the server for the polls to the FADD mutex value. In terms of network round-trips, the CAS release message only needs half a RTT until the next holder becomes aware of the release. The FADD release-write needs half a RTT to the server before an already in-flight read is able to see the update. It takes another half RTT before the next holder gets his read response from the server.

Also, CAS mutexes are more versatile due to their support for the `TRY_ALLOC` operation. It allocates the mutex if it is available and otherwise leaves it unchanged. This is achieved by issuing `tail = CAS(0, competitor_id)` even if masked atomics are available. In case of the mutex being free, the tail is set to 0. Otherwise, the comparison fails and the mutex value stays untouched. FADD mutexes do not support this operation since each alloc increments the next ticket id counter. It is not safe to decrement that value later, since in the meantime other competitors could also have issued FADDs and increased the next ticket id even more. At that point, the holding ticket id must eventually be increased even if the `TRY_ALLOC` operation failed.
3.3 Distributed Hash Table

The Distributed Hash Table (DHT) forms the heart of the Distributed Lock Manager (DLM): It is the data structure shared by the clients. The table allows fast and highly concurrent access to lock entries while constructing them only when needed. It consists of the following three building blocks (see Figure 4):

![Figure 4: Overview of the distributed hash table.](image)

**Lock entry** Represents a lock that is addressed by its unique lock-id. Locks are created and deleted as needed. The number of granted requests is kept in the lock entry.

**Request** Several competitors might concurrently want access to the same lock. These requests are of possibly conflicting types. The granted as well as the waiting requests are stored in a linked list of fixed-sized request buckets, forming a history.

**Index** Ensures fast lookup of lock entries by hashing their id. To support fast insertion of new requests, the pointer to the head request bucket is also kept in the index.

3.3.1 Components

We first have a closer look at these components before explaining how locks are acquired and released. Figure 5 depicts the DHT in more detail.

**Index** The index employs a cuckoo hash table to support fast lookup of lock entries (cf. Section 2.5). We went with the bucketized variant and chose the bucket size to be 5. Five slots result in a bucket size of 128 bytes. That exactly fits into two cache lines on our machines. It can be changed for different setups.

When searching lock entries in the index, the slot of the primary hash function is always checked first. Only if we do not find the entry in it we force a lookup into the slot of the secondary hash function. Unfortunately, that makes the first half of the index more contended since the primary index always points into it. This is why we hash the lock id with a third function and swap the primary and secondary slots if its outcome is odd. This evens out the accesses to both halves of the index.

---

13See Section 3.3.2.
The lock-id is used as the key of the index entry. It points to both, the according lock
entry and the request bucket which forms the head of the request queue. Concurrent
accesses to the hash table are synchronized by employing a distributed CAS mutex per
index bucket. One needs to alloc it before reading or writing any of its entries. We chose
the CAS mutex because of its support for the \texttt{try\_alloc} operation. By using it, we
are going to allow faster search times.

The result is the data structure in Listing 3.

\textbf{Lock entry} Each lock that is kept in the index has a corresponding \textit{lock entry}. Its
only properties are the number of currently granted requests by lock mode. This allows
to tell if a new request can be granted, without having to check through potentially
many linked request buckets. It turns out that counters for exclusive lock modes can
be omitted since only ever one request of them can be granted. The lock information is
stored outside the index on purpose: Its memory location is fixed even if a relocation
happens on the index. That allows accessing it at release time without having to search
for the lock-id in the index.

The server preallocates a fixed number of lock entries in contiguous memory. Note
that this is conceptually different from preallocating space for each lock at the server.
The containers can be re-used once a lock is removed from the index. Therefore, the
pool only needs to contain the maximum number of concurrently held locks. Entries are
referenced by their offset in that pool. Concurrent access to the pool of locks is managed
by a free-list.\footnote{See Section 3.4.}
struct Index {
    struct Bucket {
        using ptr_t = uint64_t;

        struct Entry {
            using idx_t = uint64_t;

            uint64_t lock_id;
            Lock::ptr_t lock_ptr;
            RequestBucket::ptr_t head_request_bucket_ptr;
        }

        uint64_t mutex;
        Entry entries[5];
    }

    Bucket buckets[kNumIndexBuckets];
};

Listing 3: The structure of the DHT index.

struct LockPool {
    struct Entry {
        using ptr_t = uint64_t;

        uint64_t num_granted[kNumLockModes - kNumExclusiveLockModes];
    }

    FreeList<Entry::ptr_t, kNumLockPoolEntries> free_list;
    Entry locks[kNumLockPoolEntries];
};

using Lock = LockPool::Entry;

Listing 4: The structure of the DHT lock pool and its entries.

Requests  Every competitor, trying to get access to a lock, creates a request entry. It consists of its lock mode, an identifier of the competitor’s node as well as the status of the request.

Several requests are stored in per-lock buckets which are linked to form a queue. That results in a history of requests. A new entry is appended to the head. Therefore, it is either inserted to the head request bucket or, if full, a new bucket is added to the queue. The most significant bit (MSB) of the next pointer indicates whether a request bucket is the tail of the queue.

Thereby, requests are always in one of five possible states. Initially, a bucket con-
```cpp
struct RequestBucketPool {
    struct Bucket {
        using ptr_t = uint64_t;

        struct Entry {
            using idx_t = uint64_t;
            struct ptr_t {
                RequestBucketPool::Bucket::ptr_t bucket_ptr;
                RequestBucketPool::Bucket::Entry::idx_t slot_idx;
            };

            uint16_t node_id;
        }

        enum Status {
            STATUS_UNUSED,
            STATUS_WAIT_FOR_GRANT,
            STATUS_GRANTED,
            STATUS_ABORTED,
            STATUS_DELETED,
        } status;

        uint8_t lock_mode;
    };

    uint64_t mutex;
    Entry requests[kNumEntriesPerRequestBucket];
    ptr_t is_tail_and_next_bucket_ptr;
}

FreeList<Bucket::ptr_t, kNumRequestBucketPoolEntries> free_list;
Bucket buckets[kNumRequestBucketPoolEntries];
using RequestBucket = RequestBucketPool::Bucket;
using Request = RequestBucketPool::Bucket::Entry;
```

Listing 5: The structure of the DHT request bucket pool, the request buckets and its entries.

tains only UNUSED entries. New lock requests consume a slot by marking it as WAIT_FOR_GRANT. Once conflicting requests are released, it is moved to the GRANTED state. Alternatively, a client is able to ditch a request that was waiting for too long by setting its state to ABORTED. A request is marked as released by updating its state to DELETED. Additionally, a garbage collecting operation eventually moves ABORTED requests to that same state.

Concurrent accesses to a request bucket are synchronized by a FADD mutex. It has to be grabbed before reading or writing the bucket. Its holder-id forms a convenient way of getting a unique value during garbage collection. However, it could also be constructed
alternatively, allowing to use CAS mutexes for request buckets.

A request bucket consists of twelve request entries. This results in a total size of 64B per bucket which fills a cache line on our machines. Again, for different setups this number can be adjusted.

As for lock entries, a fix-sized pool of request buckets is preallocated by the server and managed by a free-list. Pointers to request buckets store the index in that pool.

3.3.2 Acquiring a lock

Searching for the lock in the DHT Since locks are created as needed, accessing one requires to locate it in the DHT first (cf. Figure 7). After calculating the two hash values for its lock-id, the mutexes of the index buckets need to be allocated. In the first round it is assumed that we get them right away. This allows us to pipeline the CAS call to the mutex and the READ operation for the index bucket. These calls are guaranteed to not being reordered by the RNIC. Therefore, successfully allocating the mutex implies that the read fetches consistent data. Since we do not know if a cuckoo situation had to be resolved, and therefore the lock’s index entry had to be moved to its alternative bucket, we try to alloc both of them right away. Moreover, we issue a TRY_ALLOC to the mutexes. If we happen to only get one of them, but it’s bucket contains the lock, we do not have to wait for and eventually deallocate the other one. Figure 6 depicts the first alloc round to the DHT.

After waiting for the first read to return, we are able to tell if we got the mutex to the first index bucket. If so, and if the read reveals the searched lock-id, we are done. Otherwise, we wait for the second CAS to complete. If that alloc succeeded we wait for its corresponding read and check if the lock-id can be found in the second index bucket. If so, we are done.

In case we were not able to get both mutexes, we have to wait for them to be sure that no
other competitor concurrently creates or deletes our searched lock. Special care has to be taken to avoid deadlocks: Assume two competitors A and B concurrently searching for the same lock. A manages to get the mutex of index bucket $h_1(lock\_id)$ but fails to get the mutex of bucket $h_2(lock\_id)$. Competitor B grabs the second mutex but fails to get the first one. A deadlock occurs if they now both simply wait on the mutex they did not get. (Re-)allocating mutexes in order ensures deadlock avoidance.

This is why we sort the mutexes by their offset in the DHT and wait for them in that sequence. If we previously only managed to alloc the mutex with bigger offset in the index, we first have to deallocate it before being allowed to wait for the one with lower offset. When we eventually get a mutex, we re-issue the READ to its corresponding index bucket. If it happens to contain the lock-id, we are done. A new lock entry has to be created if we got both mutexes and both index buckets do not contain the lock-id we were looking for. In such a situation we choose any empty slot from the index bucket that is less occupied.

Finally, the client is left with either the index bucket that contains the lock-id or the one
where a new entry should be placed. We then deallocate the mutex of the other request bucket, if we hold it. Note, this reduces contention on that bucket while correctness is ensured: No other competitor can concurrently decide to create a lock entry for the same lock-id, nor remove an existing one. For these operations one would need to hold both mutexes, which is impossible since we still have one allocated.

Creating a new lock entry  If the lock has not been found in the DHT, a new entry has to be created. To avoid extra communication with the server during this part of the acquire operation, we fetch a free lock entry and request bucket from a local cache.\textsuperscript{15,16}

The next pointer of the request bucket is set to nil while the MSB of the pointer is set to 1 since that bucket forms the tail of the request bucket queue. The first request of the bucket is filled: The competitor's id is written into node_id, the mode is set to the given one, and the status is marked as STATUS_GRANTED.

If the lock mode is non-exclusive the corresponding counter in the lock entry is set to 1. Otherwise, all counters are initialized to zero.

Finally, the properties of the index bucket slot are set. The lock_id is updated to the given one, while the just filled entries are linked into the lock_ptr as well as the head_request_bucket_ptr fields.

All the changes are written back to the server by pipelined calls and the index bucket's mutex is deallocated. Both the lock entry as well as the request bucket are new entries to the DHT.\textsuperscript{17} Therefore, no concurrent accesses are possible and their mutexes do not need to be held for their modification.

This completes lock acquire, and the request is granted.

Appending to an existing lock entry  Inserting the request to an existing lock is slightly more involved than creating a new one, since it has to be checked if we can grant it. We directly get access to the head request bucket as its pointer is stored in the index. No concurrent acquireys can happen due to our ownership of the index bucket mutex. Nevertheless, other competitors could try to release requests at the same time while avoiding accesses to the index. As these requests might be stored in the head request bucket, we have to allocate its mutex to get exclusive access. We do this in the same manner as we did before when allocating the index buckets’ mutexes: An optimistic read of the request bucket is sent along with the atomic operation that initializes the mutex allocation. Here, we immediately issue an ALLOC since we want access for sure. If the allocation succeeds, we shortly after receive the contents of the request bucket.

\textsuperscript{15}Fetching them means that the offset to a free entry on the server is obtained.
\textsuperscript{16}See Section 3.4 for a discussion about how the local cache works and how free entries are pulled from and pushed back to the server.
\textsuperscript{17}They are not linked by the index.
Otherwise, we poll for the grant of the FADD mutex and eventually re-read the request bucket.

Next, we look for a free slot in the head request bucket: Starting from the last one, we search for the first entry that is either marked as WAIT_FOR_GRANT or GRANTED. We use the slot right after it. If we do not find such an entry we use the first slot which is then ensured to have status DELETED or ABORTED. On the other hand, it can occur that the slot we found is the last one in the bucket: There is no slot after it and the bucket is full. We fetch an unused request bucket from our cache and use its first slot. We update the next pointer in the old head request bucket, ensure that the next pointer in the new head is nil and its MSB is set to zero. Moreover, we update the head_request_bucket_ptr in the index. We are allowed to do these operations since we hold all required mutexes.\textsuperscript{18}

Requests can only be granted if the preceding request is granted. We call the most recently granted request in the history the granted-boundary. Thus, if the status of the preceding request is WAIT_FOR_GRANT we are not at the granted-boundary and set our status likewise. There is the situation where we use the first slot of the (old) head request bucket. We don’t know the status of the preceding request since we did not load that bucket. However, the status of our slot is ensured to be either DELETED or ABORTED. In the former case, it once had to be of status GRANTED, and we are at the granted-boundary. The latter status indicates an abort of a request that was not granted. We are not at the granted-boundary and have to wait.

Being at the granted-boundary does not necessarily mean that our request can be granted – its mode might conflict with already granted requests. Therefore, we look for any request in the (old) request bucket that is granted, and whose mode conflicts with ours. If we find one we have to wait and set our status to WAIT_FOR_GRANT. It remains the problem of becoming aware of conflicting requests in different buckets. This case can safely be ruled out if the (old) head request bucket is marked as also being the tail. In that case we can grant the request. Otherwise, we need to read the counters of the lock entry. Only if they do not indicate any granted requests in a conflicting mode we are allowed to grant our request.

If we have decided to grant our request, and its lock mode is non-exclusive, we have to increase the mode’s counter in the lock entry. If the (old) request bucket happened to be the tail as well as the head of the request queue, we know that no other competitor is able to concurrently modify the counter values.\textsuperscript{19} Moreover, we see all currently present requests. For the sake of throughput we then count the granted requests having the same mode than ours, increment that value and WRITE it back to the server. Otherwise, we

\textsuperscript{18}Note, we delay write-backs to the server as much as possible. This avoids duplicate writes if values in the neighborhood are changed later on. Therefore, if we say we write the values we mostly do it locally and do the real write at a later time. However, since the exact moment is highly dependent on corner-cases of the algorithm, we omit explicitly stating when it is done.

\textsuperscript{19}If we do not have all request buckets of the history allocated, counters might be decremented due to parallel releases.
issue an atomic FADD +1 to the counter.

Finally, we write all changes to the server and deallocate both, the request bucket mutex as well as the index bucket mutex. If we were not able to grant the request we wait for the wakeup message.

**Lemma 1.** A granted request with exclusive mode is always preventing us from granting ours.

**Proof.** If we happen to be the neighbor of the granted request we notice the conflict and wait. Otherwise, its neighbor and therefore all following requests are not granted. Our preceding request is waiting, and we decide to do so as well. It cannot happen that we use the first slot of the (old) request bucket with the status `DELETED`. This would imply that this request once was `GRANTED`. This situation conflicts with the assumption that the granted-boundary is at the exclusive-mode request.

**Lemma 2.** The counters in the lock entry could be modified concurrently since we do not hold any mutex on them. However, such modifications never result in our request being granted too early or not at all.

**Proof.** Simultaneous acquires on the same lock are synchronized since we hold the mutex of the index bucket. Neither can releases of requests in the head request bucket take place since they need to grab its mutex which we are holding. However, releases in different buckets could happen at the same time. If they are of non-exclusive type they first atomically decrement the counter of their lock mode. Then they possibly start granting requests (cf. Section 3.3.3). We need to distinguish two cases:

a) Assume we wrongfully decided to grant our request. This means that we are at the granted-boundary and the lock counters we had read indicated a compatible situation. Since we are at the granted-boundary and hold the mutex of its bucket, no other competitors can be granting requests. The only modification of the counters can therefore be decrements. However, a compatible situation can never become incompatible by decreasing counters.

b) Assume we decided to not grant our request when we should. That means we are either not at the granted-boundary or we detected incompatible counter values. The decision whether we are at the granted-boundary is independent of the counter values. Therefore, we must assume that we are at it but decided to not grant the request. However, since we saw incompatible counter values there must have been releases in the meantime that decremented the values to a compatible state. But at least one of them must have started a grant operation which eventually will grant our request. This contradicts the assumption that our request is never going to be granted.

---

\(^{20}\)Non-blocking acquires are possible. The client is then informed that the request is not granted yet. That allows it to issue more lock requests and wait for them at a later time.
3.3.3 Releasing a lock

Releasing the request When we acquired the lock, we remembered the pointer to the lock entry as well as the bucket where our request is stored. This provides us with all the information we need to release, without having to touch the index. First, we have to get exclusive access to our request bucket by allocating its mutex. As before, we optimistically read the bucket right after issuing the FADD operation and re-read it if we were not able to get the mutex right away. Once we are the owner of the mutex, we switch our request status from GRANTED to DELETED (cf. Figure 8 (■)).

Granting waiting requests Our request could have been the last one preventing others from being granted. In that case we are responsible for waking them up. See Figure 8 (■) for a visualization of the following paragraphs.

If our request was of exclusive mode, we decide to start the wakeup process no matter what. Otherwise, we first decrease the counter of our lock mode. As when increasing it, we check if our request bucket is marked as the tail and head of the bucket queue. If so, we can update the counter by a WRITE. Otherwise, we have to issue a FADD -1. Only if the new counter value drops to zero the grant situation might have changed, and we decide to start the grant operation.

First, we need to get to the granted-boundary. We do so by iterating through the request bucket queue, allocating all bucket mutexes on the way. Moreover, we count the granted requests we come across by mode. When we eventually reach the granted-boundary, we use these counter values to check if they prevent the first waiting request from being woken up. If not, we read the complete counters from the lock entry to be sure. Note, there could be granted requests in buckets before our first one. We did not see them but they might prevent a wakeup. By reading the counters from the lock entry we get aware of them. We avoid the read if we allocated the tail request bucket and therefore have seen all granted requests.

We then wake up waiting requests until we reach an incompatible one or the end of the list. This might require us to further iterate through the request bucket queue. Finally, we update the granted counters in the lock entry. Again, if we hold the mutex of the tail request bucket, it is ensured that no other competitor concurrently modifies these values and we can update them with a WRITE. Otherwise, we need to issue a FADD per lock mode.

Deleting obsolete request buckets The only missing piece is how resources are recycled when they no longer are in use. Once a request bucket is full the queue is enlarged. From that point on no new requests get written into that bucket again. So, eventually all its requests are granted and released. We call such a bucket obsolete. Removing arbitrary buckets from the queue is complex to do asynchronously, time consuming and requires
Alloc & read RB

Set request’s status to **DELETED**

**Yes**

Is exclusive request mode?

**No**

Is RB tail & head?

**Yes**

Write new granted count

FADD new granted count

New granted count == 0?

**Yes**

At granted-boundary?

**No**

Is RB head of queue?

**Yes**

Alloc & read next RB

First waiting request potentially compatible?

**Yes**

Is tail RB allocated?

**No**

Read granted counters from lock

Wake up grantable requests in RB

Alloc & read next RB

**Done**

**No**

Is tail RB allocated?

**Yes**

Is current RB obsolete?

**No**

Flag RB as new tail

Is RB head of queue?

**Yes**

Flag RB as new tail; Write magic request

**No**

Add RB to cache

Alloc & read next RB

**Yes**

Is tail RB allocated?

**Yes**

Flag RB as new tail

**No**

Dealoc head RB

Push lock to background GC

**Yes**

Put lock to background GC

**No**

Get next GC entry

Nullify index entry;
Put lock entry & RB to cache

Valid magic?

**No**

Find lock in index

**Yes**

Alloc & read RB

Woken up last request in RB?

**No**

Write new granted count

FADD new granted count

Has next RB?

**Yes**

**No**

Is tail RB allocated?

**Yes**

**No**

**Figure 8:** The interaction between releasing a lock (■), granting waiting requests (■), deleting obsolete request buckets (■) and garbage collecting old locks (■).
it to be doubly linked. This is why we opted for deleting obsolete request buckets from the tail only. Thus, if we release the last request in the tail bucket we are responsible for cleanup – independent of the decision if we granted requests or not.

As depicted in Figure 8 (\(\text{\#}\)), cleanup is easy: Since no valid pointer references to obsolete buckets any longer, we simply put it into the local cache and are done. Once more, we iterate through the request bucket queue until we either reach the first granted entry or we hit the head bucket.\(^{21}\) In the former case we mark the bucket containing the granted request as being the new tail, write back that change to the server and are done.

If the head request bucket is obsolete, too, the lock needs to be completely removed. This includes removing its entry from the index, requiring us to fetch that bucket’s mutex for exclusive access. However, in the meantime a competitor could have started inserting a request to our lock. He already allocated the mutex of the index bucket, but has to wait for us to deallocate the head request bucket’s mutex. Fetching the index-entry’s bucket-mutex will result in a deadlock. Moreover, in this situation the lock clearly should not be removed.

This is why we first deallocate the head request bucket’s mutex. Our lock’s index entry could have been moved due to a cuckoo situation. Therefore, we have to search for it as during lock acquire. Once found, we have to re-alloc the head request bucket. Now we can decide whether the delete should still take place. If so, we nullify the index entry, and put the lock entry as well as the head request bucket to our local cache. Clearly, that operation is time consuming. Fortunately, it can be done asynchronously since no latency critical operations of other competitors are delayed (cf. Figure 8 (\(\text{\#}\))).

Deallocating the head request bucket’s mutex may result in many dangerous situations which require special care to be taken. If – as described above – a competitor was waiting for it to be deallocated, we might see a granted request in the head request bucket after re-allocating it. This clearly aborts the delete operation.

Moreover, that request could have been granted and already been released again. This is dangerous since it led that competitor to start a delete operation on his own. However, only one such operation is allowed to complete, to avoid the deleted entries being placed into two caches. Even worse, there are situations where the request bucket looks the same before the deallocation and after the re-alloc, although a request has been granted and released in the meantime.\(^{22}\) This is why we write a magic value into the first request, before deallocating the head request bucket’s mutex: We set its \textit{status} to \textit{DELETED} while using a special, unused value for the lock \textit{mode} indicating the magic request. To differentiate two delete operations we use the less significant two bytes of the holding ticket-id of the request bucket’s FADD mutex. We set it as the value for the \textit{node_id} field of the request. This is a strictly increasing value and uniquely identifies a delete

\(^{21}\)Note that we kept obsolete request buckets allocated after iterating through them while granting. In that case the second iteration does not take any time on the network.

\(^{22}\)Since the delete happens in the background, the same competitor could again issue the acquire & release of that request and enqueue a delete twice. Requests of the same competitor look alike.
operation. During lock acquire that slot will be used, and the magic value is overwritten. Thus, the pending delete operation will be aborted. If that request is released before our re-alloc, the magic value is written once more by that competitor. However, it will be different from ours and we abort the delete operation as well.

Two more situations could occur: a) The lock-id might not be found in the index. This indicates that a second delete operation of a situation described above already completed. Or b) we might see a different head request bucket being linked in the index, if many acquires took place in the meantime. Both situations allow an early abort of the delete without having to allocate the head request bucket.

### 3.4 Free-list

Both the lock entries as well as the request buckets are preallocated by the server in a contiguous pool. Competitors who want to create a new lock, or extend an existing one with an additional request bucket, need to fetch an unused entry from that pool. Upon cleanup of obsolete request buckets and garbage collecting old locks, these entries are eventually returned.\(^{23}\) We employed a circular buffer of free slots to allow grabbing them fast: a free-list.

![Image](image.png)

(a) Initial setting.  
(b) Situation after grabbing the 5 entries \(\{0, 1, 2, 3, 4\}\) from the buffer.  
(c) The buffer after giving back the entries \(\{3, 1, 4\}\).

Figure 9: The life of a free-list.

The list consists of the same number of slots than the pool – initially each containing an index of a free entry in the pool (see Figure 9a). Additionally, two pointers into the free-list define where the current head and tail of the buffer are. Whenever a client wants to grab an entry, he issues a FADD \(+x\) to the head pointer. This moves it forward while the client is presented with its old position (see Figure 9b). That enables him to then Read the free indexes from the range \([\text{old\_head}, \text{old\_head} + x)\). Giving back entries is

\(^{23}\)Note, they are simply marked as usable. The entries are never moved in memory.
done similarly: One issues a FADD \( + y \) to the tail pointer which enables the client to 
\text{Write} \text{ into} \quad [\text{old\_tail}, \text{old\_tail} + y] \quad (\text{cf.} \text{ Figure 9c}).

Clients maintain a cache of free request buckets and lock entries, since grabbing(releasing) 
them from(to) the server requires an atomic operation which is followed by a read(write). 
This is too slow to do at each acquire-release of a lock. Only if a given low(high)-
watermark is underrun(overrun) entries are fetched from(released to) the server. This is 
also more performant since multiple entries can be fetched(released) in bulk.

\textbf{Improvement using masked FADD} \quad \text{We would benefit from having the head and tail} 
\text{pointers in one atomic field: It allows us to check for an underflow while grabbing} 
\text{entries from the free-list. After increasing the head pointer we would also atomically} 
\text{get the value of the tail. Having both values at hand, enables us to notice whether the} 
\text{head stepped over the tail in the ring buffer. Unfortunately, as for FADD mutexes, the} 
\text{counters are eventually victims of overflows. The one that is stored in the LSBs would} 
\text{then write into the counter of the MSBs. Storing them in one field was possible for} 
\text{FADD mutexes since for one counter}\textsuperscript{24} \text{we always knew the old and new value. That} 
\text{allowed us to do a subtraction instead of an addition in case an overflow would happen.} 
Unfortunately, neither the value of the head pointer when grabbing nor the tail when 
returning elements to a free-list is known. However, having the experimental verbs at 
hand enables us to use masked FADDs. The mask splits one atomic field into several 
independent counters that overflow as expected – exactly what we need.

Overflow checks are currently not done if two atomic fields are used. To avoid under-runs 
of the buffer, one would need to use a mutex or involve the server. That could be done 
at limited cost since accesses to the free-lists are rare due to the caches at the clients.

\textsuperscript{24}The holder ticket id.
4 Evaluation

In this section we are going to look at the performance of our work. We ran the clients on 15 machines. 12 of them are equipped with two 2.4 GHz Intel Xeon E5-2630 CPUs with 8 cores and two threads per core (32 hardware threads per machine). They have access to 252 GiB of DRAM. Another three machines were used for client processes: They are equipped with four 2.4 GHz Intel Xeon E5-4650 CPUs with 10 cores and two threads per core (80 hardware threads). Their DRAM size is 504 GiB. These 15 machines are connected with a Mellanox ConnectX-3 56 Gbps Infiniband NIC. The server was either run on an identical machine or at one with four 2.4 GHz Intel Xeon E5-4640 CPUs with 8 cores and one thread per core (32 hardware threads). This machine has 504 GiB of memory and is equipped with a Mellanox Connect-IB NIC. We are going to state the server machine on each experiment by its RNIC model.

Confidence intervals are stated at the 95% confidence level. If not specified otherwise, experiments were issuing $n = 20\,000$ operations. They were running in three identical rounds where measurements were taken in the middle one, while the others served as warm up and cool down phases.

4.1 Performance of RDMA operations

First, we measured the performance of the different RDMA operations. We ran a server on an IB as well as a X3 machine. The remaining 15 X3 machines were running client instances which were issuing operations to the server. We tested the performance using different access patterns: In the no contention case each client was assigned with its unique memory location. For random contention, clients chose a new location from a 120 MB range uniformly at random for each operation. During the full contention experiments all clients were accessing the same location. They kept the send queue filled with eight (four for FADDs) outstanding operations when running the throughput experiment since that led to the best performance results. We conducted the experiments with a warm-up period of six seconds. Afterwards, the measurements were taken during two seconds, followed by a two-second cool down.

Reads are done to 64 B regions as that is the size of an index bucket and frequently done by our DLM. The 32 B Writes reflect modifications of the lock counters. The same operations to 1 B values might have performed slightly better due to the reduced amount of data that has to be transmitted over the memory bus as well as the network. We did not measure any differences between FADD and CAS atomic operations and therefore only show FADDs. Figure 10 presents the results.

Overall, random access has a significant impact on latency and therefore throughput. It always performs worse than the non-congested experiments. On the X3 NIC, random

\footnote{120 MB correspond to a DLM instance with an index size of $2^{16}$ buckets.}
Figure 10: The performance of RDMA operations. We ran the server on both the IB and X3 machine. The remaining 15 X3 machines were issuing Read (64 B), Write (32 B) and FADD operations to the server. They kept 8 (4 for FADDs) outstanding operations during the throughput experiment. Note the log-log scale and that the values are only defined at integer values on the x-axis. The connecting lines do not indicate continuity.
accesses hurt even more than full contention: Throughput is more than halved for Reads and FADDs. In case of Writes, full contended accesses achieve 1700% better throughput than their random counterparts. The IB RNIC is able to handle them better. Random accesses still take longer than in the non-congested experiments. However, full contention eventually becomes slower for bigger number of clients. That results in throughput of random accesses being way better than the one for full contention. We suspect this effect to be caused by implementation-specific details of the RNIC.

Upon saturating the NIC, we notice the latencies to be similar, no matter whether each client accesses its own memory location or a shared one (no vs. full congestion). However, the throughput limits are way smaller for contended accesses. Except for Writes to the X3 NIC which perform equally good in both experiments.

As shown in Figure 10b, we achieve a maximum throughput of 58.9 Mops/s on the IB NIC when performing Writes to non-contended memory locations. It takes 1.5 µs per operation. Contended Writes take the same amount of time. But, they only reach a maximum throughput of 10.5 Mops/s. The X3 card performs worse with a maximum Write throughput of 36.0 Mops/s at a slightly better latency of 1.4 µs without suffering from contention. It takes 8 clients to saturate the X3 card while 15 will do for the IB.

Reads are slower since each operation has to wait until the memory is actually fetched (cf. Figure 10a). Latency is roughly the same on both cards with 1.7 µs on the X3 and 1.8 µs on IB. However, they both suffer from contention: IB only reaches a maximum throughput of 5.3 Mops/s while X3 manages to handle 8.8 Mops/s. The maximum lies at 53.9 million Reads per second on the IB while the X3 supports only above half as much: 26.6 million per second.

Figure 10c presents our results for atomic operations. The highest throughput rate on the X3 card is 2.95 Mops/s at a latency of 1.6 µs. The card is already saturated by two connected clients – independent if the operations are congested or not. This suggest that the X3 card probably uses a global lock for atomic operations.

The IB card shows comparable performance in the full contention experiment: At a latency of 1.8 µs we get a throughput of 2.3 Mops/s. Already one client can fully load the card in that setting. However, performance gets way better when distinct memory locations are used (no contention). 30 clients are needed to saturate the card and a total throughput of 50.8 Mops/s is gained. They have to wait 1.9 µs for a response. That is comparable to the Read performance of the same NIC.

4.2 Acquiring and releasing locks

In this section we are going to examine the performance of our DLM. We compare the throughput and latencies which are achieved by different setups. If not noted differently we are specifying the latencies of acquiring a lock and releasing it separately. The throughput is given in units of operations that consist of both, the lock acquire as well
Figure 11: The latency of acquiring and releasing a lock request, split into sub-operations. We ran the server on the X3 machine and used a single client that slept for a few microseconds between consecutive acquire and release calls.

as the according release. The latencies are measured from the moment we do the call to the DLM until it returns. Hence, we do not include the time that is possibly required to wait for a grant message in case of congestion on a lock. Section 4.3 covers that aspect by analyzing the required time for issuing grants to a waiting competitor. In the following experiments the index consists of $2^{16}$ buckets. This allows for around 320 000 lock entries to be stored at a time, while consuming 100 MB of memory on the server. The lock ids are chosen uniformly at random. Thus, conflicting ids are rare resulting in low congestion on the locks. Section 4.2.3 evaluates the performance for non-uniform distributed accesses.

The first setup we are going to look at, is the time required to acquire a lock and afterwards releasing it again. We ran the server on a X3 machine and connected a single client. It issued only one acquire or release operation at a time, and slept for a few microseconds between them. That allows background processes to finish their work, and enables us to see how much time is spent in an uncongested environment. We further split up the measurements into the different sub-operations that are performed while acquiring and releasing a request. Clearly, the timings are going to vary for different setups. Nevertheless, they provide a sufficient basis for locating potential bottlenecks.
Figure 11 shows the results obtained by the previously described experiment. We assembled the sub-operations into three groups:

**Updating the state of the DHT (■)** Updating the state of the DHT on the server is equally performant for both acquires and releases. It consists of some writes followed by deallocating the single mutex that is held. These calls can be pipelined and there is no need to wait for their completion. That group forms, with around 1 µs, the quickest of all.

**Local overhead, such as synchronization efforts and enqueuing remaining jobs into work queues (■)** The local overhead is comparable for both operations, except that enqueuing garbage collection jobs during release takes about 0.4 µs longer than fetching entries from cache while acquiring a lock.

**Allocating mutexes and the initial read of the index (request bucket) for acquire (release) (■)** The initial allocation of the mutex takes most of the time in both situations: 2.6 µs for the CAS mutex on the index bucket versus 2.2 µs for the FADD mutex on the request bucket. There is no straightforward explanation for this difference as both accesses are uncongested and require only a single atomic operation. It is likely due to the random accesses on the server memory which results in fluctuating response times. Furthermore, clear separation of the sub-operation timing is difficult and a few misplaced cycles are possible. Finally, the crucial difference between the two operations forms the additional deallocation of the second index bucket mutex. These extra 0.9 µs during lock acquire form about the latency difference between the two operations: 6.8 µs for acquire and 6.0 µs for releases.

In the following sections we encounter the same pattern: Releasing a lock is always faster, and sometimes also scales better, than acquiring one. This is due to the reasons previously explained in this section. Moreover, mutexes of index buckets are allocated by all competitors whose lock-id hashes into it. On the other hand, request bucket mutexes are only ever fetched by competitors interested in that lock. That results in index bucket mutexes being under higher congestion pressure. While releases typically only need to allocate the request bucket mutex, acquires need exclusive access to one of the index bucket. Additionally, if the lock already exists, the mutex of the request bucket has to be fetched as well.

**4.2.1 Single client node**

In this section we have a look at how well a single client node is able to perform. For this we set up an experiment where the server is running on the IB machine while several competitors are spawned on the same client host. They repeatedly issue a lock request, wait for it to be granted and immediately release it again. We avoided the sleep time between two consecutive operations to explore the efficiency of highly utilized clients. Figure 12 compares the performance of running the competitors each in an own client process versus having them in separate threads of the same process.
Figure 12: The performance of a single client node repeatedly grabbing locks and releasing them right away. In a) a single process spawned the given number of competitor threads while in b) multiple client processes were competing. The server was running on the IB machine.
We note that the minimum latency of 11.1 $\mu$s for acquiring a lock and 6.6 $\mu$s for releasing it is higher than in the previous experiment. This is due to the increased congestion on the DLM, as well as the higher synchronization overhead on the client. The competitors issue lock requests at a very high frequency. Hence, already a single one is sufficient to cause the asynchronously running garbage collection to produce delays in lock acquire and release.

Sending messages between competitor-threads is possible without involving the NIC which makes it potentially faster than communication between processes. However, all threads share the same connection to the server as well as other resources like the garbage collection work queue. The concurrency control for them prevents scale-up: The latency increases linearly to the number of threads while the throughput is maximum when running three competitors only. The 70k operations per second are already beaten by two client processes. The process-version scales up to eight competitors where a throughput of 385 Kops/s is reached. Afterwards the client machine gets overloaded and throughput drops to around 150 Kops/s while the latency increases by 5 $\mu$s. Surprisingly, for lock acquire the latency goes down from 11.1 $\mu$s to 9.4 $\mu$s when using eight instead of only one client process. This pattern is not visible when running the server on a X3 machine (cf. Figure 13b). We explain it with the design of the IB NIC, but need to investigate further for its exact reason.

From now on we only compare competitors running in separate processes due to the worse performance of the multi-threaded alternative. Note, using threads instead of processes might still be a considerable choice in case of a lot of client nodes being used. The increased number of connections might become a limiting factor, or only infrequent access bursts might be experienced. The connection count increases exponentially in the former case due to client-to-client connections that are established for mutex wakeup and lock grant messages. In the latter case, resources are saved by the reduced number of processes.

4.2.2 Multiple client nodes

Since a single client machine might not be sufficient to fully load the server NIC, we went for running competitors on different hosts. Figure 13 compares the performance of an increasing number of clients versus the two different server RNICs. Competitors were evenly distributed among the client nodes which resulted in multiple processes per hosts for client counts bigger than 15. Otherwise, the test case performed as the one in the previous section.

For up to four clients both experiments show comparable results: Throughput grows almost linearly from 60k acquire&releases/s for one client to 220 Kops/s for four clients. The lock acquire latency is around 10 $\mu$s and releasing it takes 7 $\mu$s. For the IB NIC we see the acquire latency being reduced by 1 $\mu$s for multiple competitors as in the experiment before. Also, there is no latency related improvement of running up to eight
Figure 13: Comparison of running different number of clients against a server on a) the IB, and b) a X3 machine. The clients are evenly spread across our 15 ConnectX-3 machines. Every competitor repeatably grabs locks and releases them right away.
clients on different hosts vs. keeping them on the same one (cf. Figure 12b).

The real differences are revealed when considering bigger numbers of clients: Due to the limited throughput of atomic RDMA operations already four competitors are able to saturate the X3 server RNIC. They achieve a maximum throughput of 235 Kops/s at a median latency of 18.2 µs (15.6 µs) for lock acquire (release). The sweet spot is at four clients which are able to dispose 220 Kops/s at a latency of 10.0 µs. Comparing these numbers with Figure 12b shows us that a single machine which runs these four client processes is able to saturate the server in this setup. On the other hand, having the server on the IB NIC machine allows us to process operations of up to 30 clients before the latency starts to increase significantly. 1.37 Mops/s are processed at a total latency of 20.5 µs per operation: 11.5 µs for the acquire and 9.0 µs for the release. The maximum throughput of 1.41 Mops/s is reached with 45 competitors. The latency thereby increases to 16.3 µs for lock acquire and 13.8 µs for releasing a request. We moreover spot that 8 clients achieve with 460 Kops/s a higher throughput than in the single-host experiment where the same number of competitors only reached 385 Kops/s. Both the limited outgoing throughput of the client RNIC, as well as the increased pressure of having eight vs. one client processes running on a single host prevented higher throughput in the previous experiment.

4.2.3 Non-uniform lock access rates

In the experiments above, the lock ids for acquire were chosen uniformly at random. While forming a good baseline for experiments it likely does not reflect real-world scenarios where some locks are accessed more frequent than others. This is why we also ran the multi-client experiment from the last section (cf. Figure 13a) using Zipf distributed lock ids as described by YCSB [3]. We used a space of 10 billion lock ids and a distribution parameter of $\theta = 0.99$. The most popular lock is thereby accessed by 4% of the operations.

Figure 14 shows the results of this experiment. Up to eight clients perform similar to the results from Figure 13a: The median acquire latency is a little above 10 µs while the release time is around 7 µs. Throughput also increases as before. However, the maximum throughput is considerably smaller – 750 Kops/s are reached using 30 clients. Interestingly, the latency drops by more than 1 µs when running 30 vs. 15 clients while throughput is still slightly increasing. A second drop by again around 1 µs is experienced between 45 and 60 competitors. For release time the explanation is straight forward: When locks are more congested, releases are more likely to meet request buckets that are used by other requests. Grants might not be required which leaves the release operation with only a single mutex allocation, writing back the updated status and lock counter, and deallocating the mutex again. This is clearly more performant than having to grant waiting requests and starting the garbage collection process. Also, the increased number of lock entry re-usage reduces the pressure created by the garbage collectors. This again affects the speedup of lock acquire. But it becomes even more clear, when we recap the
Figure 14: The performance of increasing congestion against a server on the IB machine. The clients are evenly spread across our 15 ConnectX-3 machines. Every competitor repeatably grabs locks and releases them right away. The locks are chosen Zipf distributed at random from a space of $10^{10}$ entries using $\Theta = 0.99$. 

The way we chose to measure latency: We only account for the time spent in the DLM and not the time that is used waiting for locks currently held in incompatible modes. At higher congestion, competitors spend more time with waiting for grants. This means that the congestion which is experienced at the server stalls, or even goes down, for bigger numbers of clients. The result is very fast operations for 60 clients: 8.7µs for an acquire and 6.4µs for releases. However, reduced latency comes at the cost of fewer throughput. The many grant waits limit the number of operations that a client is able to dispose – throughput quickly drops when having more than 30 clients.\footnote{The dropping throughput for increasing congestion is as expected for the setup of this experiment. The most popular locks experience high access rates. Many requests are likely to not be compatible to each other. This results in a reduced number of grants no matter what lock manager is used.}
Whenever a lock request is incompatible with the granted set, it has to wait. This potentially results in large queues of stalled requests. We set up an experiment, where the first request was chosen to have exclusive mode. It avoided granting the following shared requests. We then measured the time needed to wake them up after the conflicting one was released. We repeated the experiment for different waiting queue lengths. As seen in Figure 15, wakeup for up to 8 requests takes $10\,\mu s$. That includes the time spent from the start of releasing the $X$ request up to the point where the last competitor received the wakeup. It jumps to $15\,\mu s$ for a queue length of 16 requests. In this scenario the waiting requests no longer fit into one request bucket and the linked one has to be allocated and processed during wakeup. For bigger queue lengths the time grows proportionally to the number of request buckets to be read.

We also measured the time spent by the competitor who is releasing the $X$ lock. It is $7\,\mu s$ when only one request bucket has to be processed. That is about $1\,\mu s$ slower than
without having to send grant messages (cf. Figure 11). Naturally, that time increases as well for bigger queue lengths. However, we notice a faster increase of release time with respect to the wakeup time. We can explain that additional latency by the growing effort which has to be made to delete obsolete request buckets as well as deallocating their mutexes. For a number of 128 outstanding requests – or 11 request buckets – it matches the wakeup time at 67\(\mu\)s.

### 4.4 TPC-C

**Setup** We also measure our DLM’s OLTP performance using the TPC-C [4] application benchmark. It simulates real-world workloads that consist of a mixture of read-only and write intensive transactions. Figure 16 depicts the layout of the TPC-C database: A company is separated into sales districts which are assigned to a home warehouse. Data lines, such as stock items, are stored per warehouse. Therefore, the table sizes are dependent on their number. The database’s total row count, and thus the lock count, is approximately \((1 + 5w) \times 10^5\).

![Diagram of TPC-C benchmark layout](image)

Figure 16: The layout of the TPC-C benchmark. Copied from [4].

Customers are in contact with the company. Most of their requests are towards the store of their district, and therefore their home warehouse. However, in some situations, e.g. orders of non in-stock items, remote warehouses are contacted. There are different transaction types, whereas only two of them ever access remote warehouses: New-order and payment transactions. They occur with a probability of 45% and 43% respectively. Remote accesses are defined to be at 1% for new-order transactions and 15% for payments. New-order transactions access 10 remote items on average. That results in an overall probability of around 11% for remote accesses.

Implementing our DLM into an existing database is out of scope of this work. That is why we ran the TPC-C benchmark against a modified MySQL server. It logged
the locks that had to be taken for the given query set. We then ran our experiments based on these traces. TPC-C dictated us to use MySQL’s *serializable* isolation level. Moreover, two-phase locking (2PL) is used by the database. The former avoids any locking optimizations during log generation. The latter allows arbitrary lock aborts and retries during our experiments without harming any consistency constraints.

In the standard TPC-C benchmark, multiple clients\(^{27}\) concurrently issue their queries to the database. Their accesses might conflict with each other which may lead to transaction aborts and retries. However, we did not want to let MySQL handle any such conflicts – that is the job of our DLM. Our goal was to generate complete, conflict-free lock access patterns for each client. That is why we ran the clients one after another. However, that imposed the next problem:

MySQL uses next-key locking to avoid phantom reads: During iteration over the index of a table, MySQL locks all rows that it encounters. Moreover, each lock is flagged to also be *next-key*. That effectively locks the gap between that row and the preceding one. Concurrent INSERTs to these ranges fail to take that gap-lock which stalls them.

Sequentially running the terminals, lets ones that were run later see the modifications of the earlier ones: Rows are possibly added or deleted during one of the different runs. Since locks are taken on them, the clients are left with a differing set of lock-ids: Assume the first transaction of each client being the same and conflicting with each other. Since their lock access patterns are consecutively generated, the locks that are taken by the first client might completely differ from the ones taken by the second client. When the traces afterwards are executed in parallel, they do not conflict even though they should.

We resolved this problem by issuing a ROLLBACK at the end of each transaction. Now, all the clients see the same static data during their whole execution. That still does not generate the exact same lock accesses as if the DLM were actually implemented into the MySQL server. E.g., modifications that would always succeed are now aborted, avoiding the alteration of the lock-id set. However, TPC-C is guaranteed to keep a fixed database size in expectation. Thus, eventually all the INSERTs and DELETEs are balancing out. Regarding lock accesses, the result is a comparable situation to the initial setting.

Previously to running the experiments, we preprocessed the lock access patterns that we received from the MySQL server: Since our DLM does not support lock upgrades,\(^{28}\) we were scanning all the lock accesses for each transaction. If a lock was fetched twice and their modes were incompatible, we modified the first request to use the upgraded mode and dropped the second request. Moreover, the row-locking manner of MySQL resulted in thousands of requested locks, e.g., during a table scan. Therefore, we created a second trace that limited the number of lock requests per table to 32. When exceeding that number, the row-locks were transformed into a single table-lock. Note, this makes them targets of increased contention. We are going to identify experiments with such

---

\(^{27}\)TPC-C uses the term *terminal* for what we call client. We use them interchangeably.

\(^{28}\)See Section 7.1.
modified traces limited.

Lock-ids were generated by hashing their index location in the MySQL database. Additionally, the ids were prefixed by the warehouse-id to avoid conflicts when being handled by the same server.

We ran our experiments in two different settings: First, we used a single, centralized server on the Connect-IB machine which handled all lock requests. Second, we ran a server on all the 15 ConnectX-3 machines. Thereby, we partitioned the locks by warehouses and equally distributed the warehouses to the servers. We also spread the clients over the same machines, ensuring the server of their home warehouse being run locally. 89% of clients’ lock-requests were towards their home server. However, due to the limitations discussed in Section 7.2 they needed to be processed by the local RNIC as well. For more than 15 clients, multiple processes were spawned per machine.

Transactions running in parallel are victims to deadlocks. We introduced a wait timeout for lock requests before assuming a deadlock. In that case, we canceled the transaction by releasing all the lock requests we already took in reverse order. The transaction was then retried. After three unsuccessful attempts, the transaction was definitely aborted and no longer retried. Care had to be taken s.t. after a deadlock was detected, it was unlikely for it to happen again. Having transactions with high number of locks, and therefore high run times, results in deadlocks being detected at approximately the same time by all parties. In that situation deadlocks are only successfully omitted by delaying all but one transaction by a bit. Therefore, we used an exponential growing per-lock waiting time window, starting at $1 \text{ ms}$. The actual waiting time was chosen u.a.r. from it. That shuffled the deadlock detection among the participants.

Results Figures 17 and 18 depict the results of our experiments. We ran them for 10,000 transactions in the limit testcases and 100 otherwise. Thereby, we counted the number of requests that were put into the DHT (locks taken). We also measured the amount of requests that were removed from the DHT because of transaction retries and aborts (releases due to TX retry). The wait-for locks counter kept track of the number of requests that were not granted right away. We calculated the latency per lock by dividing the duration of the experiment by the number of taken locks – it contains the time required to acquire and release a lock. Finally, we sensed how many transactions were retried and aborted (TX retries and TX aborts). All counters are depicted in ratios per second.

In all experiments we can clearly see, having only one warehouse does not scale at all: Congestion is too high on the few locks. The result is only around 3% of them being successfully taken. The rest was aborted after having to wait for too long for their grant. For more than two clients, over 95% of the transactions were retried. While the latency rapidly increases in the limit cases, it stays at a moderate level for non-
Figure 17: **Centralized** TPC-C benchmark with a single server running on the Connect-IB machine. Queries were issued to different warehouse counts. In the *limit* experiments we set the maximum number of locks per table per transaction to 32. Note the different y-axis ranges for the two rows and that the latencies are only defined at integer values on the x-axis. The connecting lines do not indicate continuity. The bars are overlayed and not stacked.
Figure 18: Distributed TPC-C benchmark where each of the 15 ConnectX-3 machines ran a server. The given number of clients were distributed among the same machines. Queries were issued to different warehouse counts. In the *limit* experiments we set the maximum number of locks per table per transaction to 32. Note the different y-axis ranges for the two rows and that the latencies are only defined at integer values on the x-axis. The connecting lines do not indicate continuity. The bars are overlayed and not stacked.
limited traces. In the former case (cf. Subfigures d), congestion on the DHT mutexes of the table locks is high. That does not trigger any aborts but consumes time. In the latter (cf. Subfigures a), the higher number of distinct lock ids allows for fast insertion into the DHT. However, deadlocks are now frequent and transaction retries happen way earlier.

Having more warehouses increases the lock-space which naturally results in reduced congestion. For fewer clients than warehouses, lock collisions are rare due to the small number of only 11% remote accesses per client. The limit experiment for 15 warehouses produces almost identical results in the centralized and distributed version of the experiment (see Subfigures e). Lock throughput increases linearly up to a number of 15 clients where it reaches 800 Kops/s at 19 µs. Having more clients results in multiple being assigned to the same warehouse. Thus, competition for the same locks grows a lot: Latencies explode while aborts become frequent. In the distributed version we are able to see the phenomenon described in the previous paragraph in more detail: In the limit experiment (Figure 18 d) lock throughput decays to almost zero due to the high congestion on the relatively few table lock entries in the DLM. The limit benchmark escapes this limitation as lock aborts become more frequent due to deadlocks. Quick retries result in an increasing number of placed locks per second for more clients than warehouses. Still, the overhead introduced by deadlock detection starts to increase latency.

Interestingly, Figure 17 b and c show unexpected results: Throughput is capped at around 200 000 locks/s while transaction aborts are nearly zero. Moreover, in the limit experiment no such shortcomings are visible. The reason was found in the bigger number of concurrently held lock requests in the no-limit benchmarks. The fuller DHT resulted in more cuckoo situations during acquire. Resolving them increased latency. We re-ran the experiment with a bigger index size. Unfortunately, that did not end up in better performance: The increased size of the DHT hit us due to the reduced random-access throughput of the RNIC (cf. Section 4.1).

Having 128 warehouses almost entirely avoids congestion. Therefore, we are able to reach maximum throughput: In the centralized, limit version up to 1.35 Mops/s are gained at 22 µs. These numbers match the micro-benchmark depicted in Figure 13a. Due to the few per-client table lock-ids, no cuckoo situations have to be resolved in that experiment. The locks are frequently re-used and easily fit into the DHT.

In the distributed runs, the throughput is no longer capped as in the no-limit experiment of the centralized version. That makes sense since now more indexes are used, distributed over the different machines. Cuckoo situations become rare, and a linear performance increase is visible. The maximum throughput is reached in the no-limit experiment with 128 warehouses: 950 Kops/s at 30 µs. That is a scale-up of 430% compared to the performance of a single server running on a ConnectX-3 machine (cf. Figure 13b). We are not able to reach the optimal 15x performance increase due to the machines now having to run both the server and clients. The RNICs are not able to handle that many requests.
5 Related Work

5.1 Lock managers

The work on lock managers can be split into three groups. The following sections explain them in more detail and show the current state-of-art.

Single-system Systems running on a single machine to provide locking amongst threads strive for ultra-low latency. Historically, they stored lock requests in a linked list somewhat similar to our approach (cf. [11]). Jung et al. [15] uses fine-tuned memory allocations to profit from cash locality, and a latch-free data structure to minimize congestion and make strong progress guarantees. While performing very well on a single machine where memory access is relatively cheap, latch-free algorithms tend to excessively swap pointers to elements. However, dereferencing a linked list using RDMA is very costly since for each hop one round trip on the network is required. Ren, Thomson, and Abadi [27] entirely removes the linked list of lock requests at the cost of reduced concurrency.

Yu et al. [35] analyzes existing concurrency control algorithms by evaluating their performance when running on 1024 cores. They find that all of them fail to scale, and provide potential solutions.

RDMA based DLM There is some work tackling distributed lock management using RDMA. Devulapalli and Wyckoff [6] show an implementation similar to our CAS mutex, providing exclusive-mode only locks. Narravula et al. [22] and Chung and Zamanian [2] introduce a DLM where locks can be acquired in either shared or exclusive mode. However, all of these solutions assume that all locks can be preallocated in memory. We remove this precondition and additionally provide the possibility of having any number of lock modes with an arbitrary compatibility matrix.

Transaction systems To cope with scalability issues, distributed NoSQL database systems, such as Amazon Dynamo [5] and Cassandra [17], were designed. They often employ a key-value store. There is much research going on in making them faster and increasing throughput. While being somewhat orthogonal to this work, the following solutions provide transactional access to data. That forms a subset of the problems which can be solved by a DLM.

Pilaf [20] started by performing gets using RDMA one-sided reads, while puts are done using Send/Recv to the server. HERD [16] optimizes the usage of RDMA Write and Send/Recv to minimize network round-trips and bringing down get latency to as low as 3.6µs. They heavily involve the server CPU during gets by Writeing the get request to a work queue on the server. FaRM-KV [8] uses RDMA Read for gets while
the server polls for PUT requests – similar to HERD’s GET. They moreover replicate updates to backup servers during PUT. RIFL [18] implements exactly-once semantic on-top of existing key-value stores. Finally, DrTM [33] combines RDMA with HTM (Hardware Transactional Memory) to allow fast local accesses to the store. The result is a powerful combination of advanced hardware features of which our work might also profit (see Section 7).

5.2 Hash tables

Nguyen and Tsigas [23] present a lock-free cuckoo hash table. They allow keys to co-exist in it, arguing that in that case the history is synchronizable. Therefore, one write came after the other, and the extra entries simply can be removed. Unfortunately, our model is different. We do not just store updateable values in the hash table, but use it as an index to elements that need to be unique. As soon as an entry exists in the table, all future requests must use it as well. Szepesi et al. [32] implement cuckoo hashing using RDMA, but still suffer from the same shortcoming.

FaRM [7] uses a modified version of hopscotch hashing [13]. They show that on average only 1.04 RDMA reads are needed per lookup.\(^{30}\) On the other hand, writing the index is exclusively done by the server since it is considerably more complex than lookups. Their experiments scale as they assume update rates lower than 5%. That is not comparable to what we expect: In our use-case unpopular lock entries come and go at a very high frequency. Since that would involve our server too much, we opted for our distributed cuckoo hash table that is updateable by the clients.

\(^{30}\)With a neighborhood size of \(H = 8\) at 90% occupancy.
6 Conclusion

In this work, we have designed and implemented a distributed lock manager. By exclusively using one-sided RDMA operations, we were able to move all the work from the server to the clients. Our solution allows for a huge number of locks by not having them all maintained in memory. Hence, we constructed a cuckoo hash table which supports very fast insertion, lookup and removal of locks via RDMA. To ensure synchronized access to it, we presented two different mutex implementations which leverage RDMA atomic operations.

Measuring the performance of our DLM, we found that acquiring and releasing locks is possible at latencies as low as $6\mu s$. Up to 1.4 million acquire/release cycles are served by a single server. We performed an evaluation of concurrency control for Online Transaction Processing (OLTP) workloads: We generated TPC-C lock traces and our approach achieved a maximum of $2.7M$ tpmC. In a symmetrical distributed setup, where clients were running on the same 15 machines as the servers, we were able to observe a throughput scale-up by 430%.

7 Future Work

We presented a ready-to-use version of an RDMA-based DLM. However, there is some missing functionality and even better performance might be achieved by leveraging advanced hardware features. In this section we give a short overview of the things that might be interesting to look at in the future.

7.1 Lock upgrades

When multi-granular locks are used, one usually acquires locks in the minimum mode that is required to complete a job. However, sometimes the job is not known to full extent from the beginning of acquiring locks. A stricter mode may later be required. Getting the lock in that mode from the one that is already held is called an upgrade. It is preferable over releasing the currently held request and re-acquiring it with the adjusted mode: When multiple dependent locks are held, releasing one – especially for 2PL – requires the release of all of them.

Lock upgrades are currently not supported by our work. Upgrades need to be granted before ordinary requests to avoid deadlocks. However, this requires a check for waiting upgrades on each insert of a new ordinary request that might be granted. Moreover, one must be cautious when starting a grant operation after a lock release: Waiting upgrades of request that are stored in request buckets previous to the one that is released are possible. We present two possible approaches:
• Each request contains an additional field of a requested upgrade mode. A pointer
to the bucket containing the first upgrade is stored in the lock entry. Hence, a
quick decision whether granting is allowed during insert is possible: If the pointer
is set the request cannot be granted. Moreover, it allows a release operation to
know about upgrade requests in buckets which would not be traversed otherwise.\textsuperscript{31}

• A separate per-lock upgrade queue is introduced. It resembles the request bucket
queue but only stores upgrade requests. Again, a pointer to it is stored in the
lock entry. Compared to the previous solution, traversing the separate list during
granting might be more performant. On the other hand, aborts might become
more involved.

\section*{7.2 Improving local accesses}

Common RNICs do not have a memory cache and do not participate in the cache-
coherence protocol. They therefore resolve atomic accesses locally. This is why atomic
operations to the same memory location by e.g., the CPU are not synchronized with
the RNIC. Unfortunately, this prevents clients that reside on the same machine as the
server – or the server itself – from directly accessing the local memory. Instead they
have to go all the way over the NIC by using a connection to themselves. Although
avoiding the network round-trip, RDMA accesses to local memory is still an order of
magnitude slower than direct DRAM access. There might be three possible solutions to
this problem:

\textbf{Cache-coherent RNICs} The Mellanox ConnectX-4 RNIC has built-in support for CAPI.
That attaches the card to the memory domain and makes it part of the cache-
coherence protocol. In theory that allows for system-wide atomics. Unfortunately,
we do not know if this features is exploited.

\textbf{PCIe atomic operations} Having both a RNIC and PCI root device which support PCIe
atomic operations would also result in them being system-wide.

\textbf{HTM} Hardware Transactional Memory brings the notation of transactions to memory
featuring atomicity, consistency and isolation. Processes can start transactions
and then modify memory. Commits fail, if the memory was concurrently modified
by other processes, and the transaction’s memory modifications are not applied.
Wei et al. [33] already make use of this technology. While being out-of-scope for
this work, it is clearly a candidate for local accesses support.

\textsuperscript{31}Upgrades of requests that are stored in request buckets closer to the tail than the one that contains
the released request.
7.3 Fault tolerance

We did not yet implement any fault tolerance measures. However, we present here some ideas how it could be done.

There are three fault scenarios: Either a) the server or b) a client crashes, or c) a node is disconnected from the network. In all the scenarios RDMA connections to the faulty node are ensured to eventually go down. Moreover, in situation c) the disconnected node sees a drop of all its connections. Therefore, a disconnect of a node can be seen as if it crashed.

Crash recovery of the resources that are managed by the Distributed Lock Manager (DLM) is orthogonal to our work. We therefore rely on the existence of such a mechanism.

**Client crash**  Lock requests of a crashed client must be released if they are granted or, if waiting for grant, marked as invalid. Unfortunately, the client might have left the Distributed Hash Table (DHT) in inconsistent state if it was acquiring or releasing a lock request while crashing. In these situations it might have partly written some updates to the DHT while others are outstanding.

Each client is connected to the server. Therefore, at least the server notices the crash. The server sending a crash-notification message to all clients is the easiest way of resolving any irregularities. They then abort all currently running transactions, triggering the crash recovery that is employed on the managed resources. The DHT is then reset by the server, and normal execution is resumed. However, this possibly leads to many aborted transactions which might be problematic if client crashes are somewhat frequent.

Alternatively, the server might try to fix the irregularities on the fly. If the client was not manipulating the DHT at the time of the crash, the server can traverse its memory while looking for lock requests of the client. It can then simulate the release operations that the client would have done. However, it is neither known if an operation was executed on the client at the time of the crash, nor where it stopped. Fixing partially executed operations is involved. We do not go into too much details about the full recovery process, but here are some issues and how they might be solved. Since heavy access to the DHT is required, it might be inevitable to block clients from writing to it for the time of recovery.

- Creating a new lock entry has been started: Its index bucket entry points to an invalid lock entry or head request bucket, or their content is inconsistent. The server can detect these issues and reset the entries.
- Appending to an existing lock has been started: The granted counters are corrected, and the client's lock request is removed from the request bucket. The head request bucket pointer might need to be reset to the old one if a new RB was
• Releasing a lock has been started: Waiting clients need to be woken up, and garbage needs to be collected.

• In all previous situations, the client might have been enqueued for a CAS mutex or hold it: The server collects the predecessors of all mutexes from all alive clients to reconstruct the mutex queues. The clients are notified about their new predecessor.

• In all previous situations, the client might have been enqueued for a FADD mutex or hold it: The server collects the ticket ids from all alive clients for all mutexes to find the tickets the crashed client was holding. The clients are notified about their new ticket ids.

• Unused lock entries and request buckets have been cached by the crashed client: The server collects the cached entries from all alive clients and looks into the DHT for currently used ones. The others are given back to the free-lists.

**Server crash** Each client notes a server crash since the connection to it drops. A replica server must then take over to be able to continue work. Unfortunately, the old server took the DHT with it. As before, the easiest recovery is to abort all transactions on all clients, and to restart all work once the replica server catches up. There are three alternative approaches that enable the replica to continue where the old server stopped.

Each write to the DHT is not only done to the main server, but also replicated to backup servers. Upon a crash, the new leader already has an exact copy of the DHT, and any lock request can seamlessly be continued. Note, mutexes only need to be taken on the leader server since all the writes are then synchronized on all servers. However, after a crash the mutex values might then not be correct on the replica. The current mutex holder can keep the ownership of the mutex while the waiting queue has to be rebuilt. Or at least the tail has to be found to be able to reconstruct the value of the atomic field. Moreover, mutex accesses can be distributed to all server replicas if server crashes are rare. \(^ {33}\) Reads can be arbitrarily distributed as a consistent state is guaranteed after releasing a mutex. \(^ {34}\) This increases throughput for server machines that are throughput-bound by atomic or read operations (cf. Section 4.1).

The solution of the previous paragraph allows seamless continuation after a crash. However, it requires each write to be applied to each replica. Thus, outbound writes might become a new bottleneck of clients, resulting in reduced performance. There are also more packets circulating on the network, potentially congesting switches. Therefore, a second approach would be to write a journal about completed operations (cf. [24]). The replicas can then replay that log to end up with a copy of the DHT. It is moreover pos-

---

\(^ {32}\) Scanning all request buckets reveals the one pointing to the invalid tail.

\(^ {33}\) E.g., by choosing the responsible replica by \(\text{mutex\_id} \mod |\text{replicas}|\).

\(^ {34}\) E.g., by selecting the closest server.
sible to only write the journal entries to a majority of the replicas which then populate the update amongst them – possible in bulk.

As non-volatile DRAM is getting cheaper, a third option would be to store the DHT in such memory (cf. [8]). Once a server crashes, the DHT could be copied from its memory and transferred to the new leader. Thus, continuation is rendered possible with minimal overhead since an exact clone of the DHT is made without introducing any additional messages during normal operation.
**Glossary**

**2PL** Two-phase locking  
**CAPI** Coherent Accelerator Processor Interface  
**CAS** Compare-And-Swap  
**DHT** Distributed Hash Table  
**DLM** Distributed Lock Manager  
**DMA** Direct Memory Access  
**FADD** Fetch-And-Add  
**HTM** Hardware Transactional Memory  
**MSB** Most significant bit  
**NIC** Network Interface Controller  
**OLTP** Online Transaction Processing  
**RB** Request Bucket  
**RDMA** Remote Direct Memory Access  
**RNIC** RDMA NIC  
**RoCE** RDMA over Converged Ethernet  
**RTT** Round-trip time  
**U.a.r.** Uniformly at random

**References**


Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor’s thesis, Master’s thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

A Scalable Distributed Lock Manager using One-Sided RDMA Atomic Operations

Authored by (in block letters):

For papers written by groups the names of all authors are required.

Name(s):
Zehnder

First name(s):
Sämy

With my signature I confirm that
− I have committed none of the forms of plagiarism described in the ‘Citation etiquette’ information sheet.
− I have documented all methods, data and processes truthfully.
− I have not manipulated any data.
− I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date
Zurich, December 15, 2015

Signature(s)

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.