Zeus: Locality-aware Distributed Transactions

Antonios Katsarakis[†*], Yijun Ma[‡], Zhaowei Tan[§*], Andrew Bainbridge, Matthew Balkwill, Aleksandar Dragojevic, Boris Grot[†], Bozidar Radunovic, Yongguang Zhang
[†]University of Edinburgh, [‡]Fudan University, [§]UCLA, Microsoft Research

Abstract

State-of-the-art distributed in-memory datastores (FaRM, FaSST, DrTM) provide strongly-consistent distributed transactions with high performance and availability. Transactions in those systems are fully general; they can atomically manipulate any set of objects in the store, regardless of their location. To achieve this, these systems use complex distributed transactional protocols. Meanwhile, many workloads have a high degree of locality. For such workloads, distributed transactions are an overkill as most operations only access objects located on the same server – if sharded appropriately.

In this paper, we show that for these workloads, a single-node transactional protocol combined with dynamic object re-sharding and asynchronously pipelined replication can provide the same level of generality with better performance, simpler protocols, and lower developer effort. We present Zeus, an in-memory distributed datastore that provides general transactions by acquiring all objects involved in the transaction to the same server and executing a single-node transaction on them. Zeus is fault-tolerant and strongly-consistent. At the heart of Zeus is a reliable dynamic object sharding protocol that can move 250K objects per second per server, allowing Zeus to process millions of transactions per second and outperform more traditional distributed transactions on a wide range of workloads that exhibit locality.

CCS Concepts: • **Computer systems organization** → *Reliability*; *Cloud computing*; *Availability*.

Keywords: locality, transactions, dynamic sharding, replication, availability, strict serializability, pipelining

ACM Reference Format:
Antonios Katsarakis, Yijun Ma, Zhaowei Tan, Andrew Bainbridge, Matthew Balkwill, Aleksandar Dragojevic, Boris Grot, Bozidar Radunovic, Yongguang Zhang. 2021. Zeus: Locality-aware Distributed Transactions. In *Sixteenth European Conference on Computer Systems (EuroSys '21), April 26–29, 2021, Online, United Kingdom.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3447786.3456234

EuroSys '21, April 26–29, 2021, Online, United Kingdom

ACM ISBN 978-1-4503-8334-9/21/04…$15.00
https://doi.org/10.1145/3447786.3456234

1 Introduction

Cloud applications over commodity infrastructure are becoming increasingly popular. They require distributed, fast and reliable datastores. Recent in-memory datastores that operate within a datacenter and leverage replication for fault-tolerance (FaRM [20], FaSST [34], DrTM [71]) offer strongly-consistent distributed transactions in the order of millions per second. They do not make any assumptions about the workloads and rely on highly-optimized remote access primitives (e.g., RDMA) to enable a variety of use cases.

These datastores run OLTP workloads with transactions involving a small number of objects. In addition, many applications have a high degree of locality. For example, many transactions in a cellular control plane involve one user always accessing the same set of objects (e.g., the nearest base station, or the same call forwarding number [49]). Many Internet middle-boxes mostly access the same state for all packets of a single flow (e.g., intrusion detection systems [72]). Bank transactions often recur between the same parties [10, 69, 75]. As Stonebraker *et al.* report [27], a transactional concurrency control scheme can derive significant benefit from leveraging application specific characteristics such as locality.

Existing works [20, 34, 71] can exploit locality through *static sharding* – iff all objects involved in each transaction are stored on the same node[1]. Consequently, static sharding only helps if the optimal placement is known a priori and never changes. However, this is often not the case for two main reasons. Firstly, the set of objects involved in a transaction may change over time. For instance, as a mobile phone user moves, her *cellular handover* transaction involves different base stations. Secondly, the popularity of each object changes in time, be it a network service or a financial stock. If several popular objects are located on the same server, the server becomes a bottleneck, and the popular objects should be spread across servers. In both cases, *the rate of changes in access locality is multiple orders of magnitude lower than the rate of processed transactions* (which is in millions per second). We describe these cases in more detail in Section 2.

In contrast, *dynamic sharding*, where objects are moved on-demand across nodes, helps both when the set of objects involved in a transaction changes or when object popularity shifts. In the first case, dynamic sharding ensures that all

*Part of this work was done when the author was in Microsoft Research.
[1]Throughout the paper we use the terms *node* and *server* interchangeably.

objects involved in a transaction are colocated, thus reducing expensive remote accesses. In the second case, dynamic sharding allows to quickly spread out the heavy-hitters, thus alleviating the bottlenecks. However, state-of-the-art works [20, 34, 71] do not support dynamic object sharding. Once the existing sharding is no longer optimal, they revert to remote transactions that are inherently slower. Remote transactions are slow because they impose the overhead of several round-trips both to execute a transaction via remote accesses and to atomically commit it. The source of the latter is the complexity of distributed atomic commit for conflict resolution under the uncertainty of faults.

Several systems propose application-level load balancer designs that let applications make a fine-grained decision regarding which node each transaction should be routed to [3, 5, 7, 8]. However, most of these systems rely on custom datastores that either do not provide strong consistency or are not as fast as the state-of-the-art datastores [20, 34, 71]. As argued by Adya *et al.* [2], there is a need for a general distributed protocol that would provide strongly-consistent transactions and better exploit dynamic locality.

In this paper, we address the problem of high-performance dynamic sharding for transactional workloads by presenting a novel distributed datastore called *Zeus*. The key insight behind Zeus is that, for many workloads, the benefits of local execution outweigh the cost of (relatively infrequent) resharding. Zeus capitalizes on this insight through two novel reliable protocols designed from the ground-up to exploit locality in transactional workloads. One protocol is responsible for reliable (atomic and fault-tolerant) object ownership migration requiring at most 1.5 round-trips during common operation. Using this protocol, while executing a transaction, Zeus moves all objects to the server executing it and ensures exclusive write access. Once that is done, and unless the access pattern changes, all subsequent transactions to this set of objects will be executed entirely locally and eschew the need for a costly distributed conflict resolution. The second protocol is a fast reliable commit protocol for the replication of localized transactions. By combining these two protocols, Zeus achieves performance and simplicity of single-node transactions with generality of distributed transactions. To further exploit locality, Zeus' reliable commit enables local yet consistent read-only transactions from all replicas.

Zeus design provides an extra benefit in that it allows easy portability of existing applications. Since most Zeus transactions are local, Zeus can pipeline executions without compromising correctness. A subsequent transaction does not need to wait for the replication of the current one. This is in contrast to the existing in-memory distributed transactional datastores [20, 34, 71], in which each transaction blocks until the replication is finished. To mitigate the effects of blocking, these datastores use custom user-mode threading that requires substantial effort to port existing applications onto. In contrast, Zeus transaction pipelining allows easy porting of legacy applications onto it, making them distributed and reliable while reaping the performance benefits of locality with minimal developer effort.

We implement Zeus and evaluate it on several relevant benchmarks: Smallbank [10], Voter [19], TATP [49]. We also introduce and implement a new benchmark which models handovers in a cellular network based on observed human mobility patterns. To demonstrate the ease of porting existing applications to Zeus, we port several networking applications that exhibit locality: cellular packet gateway [53], Nginx [50] and SCTP transport protocol [59].

In brief, the main contributions of this work are as follows:

- **Proposes *Zeus*, a reliable locality-aware transactional datastore** (§3) that replicates data in-memory for availability. Unlike state-of-the-art strongly-consistent transactional datastores, Zeus transactions are fast by virtue of exploiting dynamic sharding and locality that exists in certain transactional workloads (as demonstrated in §8).
- **Introduces two reliable protocols** (§4, 5). An *ownership protocol* for dynamic sharding that quickly alters object placement and access levels across replicas; and a transactional protocol for fast pipelined *reliable commit* and local read-only transactions from all replicas. Both protocols, which ensure the strongest consistency under concurrency and faults, are formally verified in TLA$^+$.
- **Implements and evaluates Zeus** (§7, 8) over DPDK on a six node cluster, using three standard OLTP benchmarks and a new cellular handover benchmark. For workloads with high access locality, Zeus achieves up to 2× the performance of state-of-the-art RDMA-optimized systems, while using less network bandwidth and without relying on RDMA. On the handovers benchmark Zeus performance with dynamic sharding is just 4% to 9% from the ideal of all local accesses. It also shows the ease of portability by porting three legacy applications showing scalability and reliability with little or no performance drop.

2 Objectives and motivation

We first describe high-level objectives that a data center operator and an application developer desire in a datastore. We next discuss the opportunities that arise with local access patterns and why they have not been explored fully before.

2.1 Datastore design objectives

Our goal is to design an intra-datacenter shared-nothing transactional database for OLTP workloads that allows programmers to deploy their software on top of a distributed infrastructure without needing to re-architect the application. More specifically, we want to provide the following:

Performance and reliability. Our target is to have a reliable datastore that can process millions of operations per second. Furthermore, to remain available despite node failures, each state update must be replicated across nodes.

Transactions. A single operation may arbitrarily access or modify multiple objects. A notion of transaction guarantees that either all modifications are committed, or none. This is in contrast to many widely used in-memory key-value stores (e.g., [56]) that essentially provide only single-object atomic abstractions and some generalizations as an afterthought.

Strong consistency. We want to provide a simple programming model where a programmer has the intuitive notion of a single-copy of state, despite the state being replicated for reliability. This model requires strongly-consistent distributed transactions guaranteeing strict serializability [62]. Informally, with strict serializability all transactions appear as if they are atomically performed at a single point in real-time to all replicas in-between their invocation and response.

Support for legacy applications. State-of-the-art in-memory datastores [20, 34, 71] meet the above criteria. However, when executing remote transactions, they block the associated threads. To mask the performance cost of blocking, they rely on transaction multiplexing and user-mode threads [34]. However, this makes porting existing applications on top of these frameworks difficult. Our goal is to provide a datastore that allows legacy applications to run on top of it without mandating modifications to the existing architecture.

2.2 A case for access locality

As noted in Section 1, many real-world applications exhibit transactional access patterns with a high degree of locality. In these cases, data is usually sharded for efficiency. However, the optimal sharding may change in time for two reasons. One is due to changes in object popularity and the other one due to changes in access locality. We use the term *locality* to refer to the temporal reuse of transactions between (spatially related) objects that reside on the same node.

Let us consider changes in locality via an example of call handovers in a cellular network. Every time a phone wakes up to process data traffic (a *service request*) or goes to sleep (a *release request*), the cellular control plane updates various objects related to the phone and to the base station this phone is attached to. This is an example of data access locality, where each consecutive operation on the same phone accesses the same two objects (the phone and the base station contexts).

However, the access locality may slowly and gradually change in time due to mobility. Every time a cellular user moves from one base station to another, her phone performs a *handover* operation. This is a transaction that involves three entities, the phone, the old base station the user is leaving, and the new base station the user is connecting to. As the user travels (e.g. during a daily commute), her phone will perform many such transactions, each involving one object that stays the same (the phone context) and two other objects that continuously change (contexts of the base stations on the way). Once the user finishes the commute, the access locality will resume, and every subsequent *service request* and *release* for the user will again involve a single base station (the one the user is currently attached to, which is different from the one at the beginning of the commute).

This change is slow in time. People are stationary most of the time. A study [12] shows that an average person makes five one-way trips per day with a total length of 100km for drivers and 20km for non-drivers (on average). Consequently, handover requests are only between 2.5% and 5% of service and release requests [45, 55], while the vast majority of service and release requests repeatedly include the same base station. Another fact that further improves locality in this case is that a base station will only take part in handovers with other base stations that are geographically close to it.

The optimal sharding should adapt to keep the relevant objects together in the same node. In this particular example, it should strive to keep the context of a phone and of the base station it is associated to on the same node. However, based on the above observations regarding user mobility, re-sharding will occasionally need to happen, though only for a single-digit fraction of transactions. We further discuss and evaluate this example in Section 8.

Another example of access locality are peer-to-peer financial transactions. Several studies of the popular peer-to-peer mobile payment system Venmo [69, 75] show that the transactions mainly occur among groups of friends, and that the transaction graph exhibits a higher local clustering than Facebook and Twitter graphs. Moreover, as noted by Unger *et al.* [69], the network remains largely consistent across the studies, indicating slow temporal change in the interaction graph. We study this case using publicly available data from a recent Venmo study [60] and evaluate it on a popular financial transactions benchmark Smallbank [10] in Section 8.

The optimal sharding may also change due to a shift in object popularity. One example of this can be found in the Voter benchmark [19], which we evaluate in Section 8. In a long-lasting online public contest (e.g., Eurovision), many users vote for a few contestants. The optimal sharding should spread the load evenly, and would ideally put the most popular contestants each on a separate server, while potentially grouping the least popular contestants together on a single server. However, the popularity of each contestant changes in time, and as she gets more or fewer votes, the optimal sharding changes as well. As in the previous example, each transaction involves only a few objects (a voter and a contestant) and the frequency of change in the optimal sharding is much lower than the frequency of the voting transactions.

Another example is stock exchange. Between 40–60% of the volume on the New York Stock Exchange occurs on just 40 out of 4000 stocks [66]. Stock popularity changes at the granularity of hours or days, whereas daily trading volume is on the order of 5-10 billion shares [48]. Thus, while transaction volume is high, the change in popularity is slow. Similar to the handover case, the re-sharding will need to happen, but relatively infrequently.

Existing works [17, 37, 61, 66] propose dynamic sharding to adapt to these kinds of changes. However, their datastore designs that support re-sharding and provide strong consistency operate at a sub-Mtps throughput. For instance, Squall [23] and Rococo [47] report up to 100 Ktps per server, and Rocksteady [37] up to 700 Ktps per server.

Meanwhile, state-of-the-art reliable in-memory datastores (e.g., FaRM, FaSST) reach millions of tps per node but have limited support for changes in locality. For instance, FaRM only supports static location hints. If the access locality changes, both FaRM and FaSST must execute remote transactions. Some domain-specific datastores have been built that exploit locality, but they do not meet all design objectives. For example, S6 [72] does not offer replication (a must for availability), while FTMB [63] runs only on one node and replicates on non-volatile storage. Overall, to the best of our knowledge, there is no in-memory datastore that meets all our design objectives and effectively exploits locality.

3 Design overview

We start this section by outlining the Zeus datastore system architecture. We then present a high-level overview of the key part of Zeus — a pair of protocols that exploit locality for high-performance transaction processing with fault-tolerance, strong consistency and programmability.

3.1 Zeus system architecture

Zeus exploits request locality and uses an application-level load balancer to enforce it. External requests issued to Zeus are issued through a load balancer. The load balancer can extract the application level information, locate relevant object keys and always forwards requests with the same set of keys to the same server. Application-level load balancers are not a new concept. Several previous systems have demonstrated such load balancers [3, 5, 7, 51]. We implement a simple one using a distributed, replicated key-value store based on Hermes [35]. We extract a key from each request and look it up in the key-value store. If not found, we pick a destination Zeus node at random, store it in the load balancer's key-value store and forward the request. If the key is found, we forward the request to the corresponding destination.

Zeus considers a non-byzantine partially synchronous model [22] with crash-stop node failures and network faults including message reordering, duplication and loss. It implements a reliable messaging protocol with low-level retransmission to recover lost messages. Zeus uses a reliable membership with leases to deal with the uncertainty of detecting node failures. Each membership update is tagged with a monotonically increasing epoch id (*e_id*) and is performed across the deployment only after all node leases have expired. This provides the same consistent views of *live nodes* across the deployment despite unreliable failure detection (similar to Zookeeper [31] with leases). For data reliability, Zeus

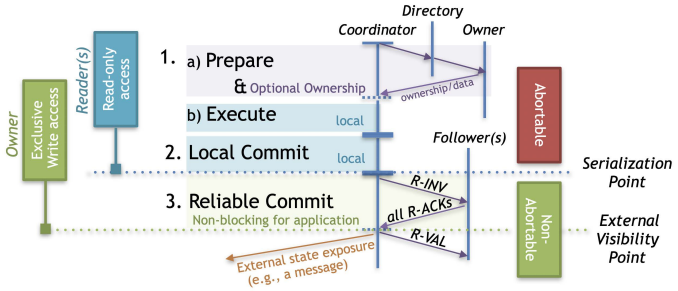

Figure 1. Zeus' locality-aware distributed transactions.

maintains replicas of each object. The replication degree is configurable; however, the higher the degree of replication, the greater the CPU and network overhead, and the lower is the throughput of transactions that modify the state.

3.2 Zeus protocols overview

Zeus is efficient in executing distributed transactions by forcing them to become local. At the heart of Zeus are two separate, loosely connected reliable protocols. One of them is the *ownership protocol* responsible for the on-demand migration of the object data from one server to another and changing the access rights (read or write) of servers storing the replica of an object. The other one is the *reliable commit protocol* for committing the updates performed during a transaction to the replicas. As these two protocols are only loosely connected, they can be optimized, verified and tested independently.

Zeus, inspired by hardware transactional memory [29], executes and commits each transaction locally, on a server designated to be the *coordinator* for that transaction. While executing a transaction, the coordinator has to secure the appropriate ownership level for each object involved in the transaction. This is the task of the ownership protocol. Once the coordinator acquires the required ownership levels and finishes execution, it commits the transaction locally. Subsequently, it copies the state of modified objects to backup servers, also called *followers*. The latter is the task of the reliable commit protocol. Crucially, the ownership protocol is invoked only the first time a node accesses an object. Subsequent transactions proceed without involving it, until another node takes over the ownership (i.e., locality changes).

At a high level, a transaction in Zeus is carried out through the following three steps (also shown in Figure 1):

1. **Prepare & Execute**: While the coordinator executes a transaction, prior to accessing an object, it verifies that it holds the appropriate ownership level (read or write) for that object. If not, it acquires the appropriate ownership level via the *ownership protocol* (described in Section 4) and continues execution. Before performing its first update to an object, the coordinator creates a private (to the transaction) copy of the object. This private copy is then used for all accesses of the transaction to the object.

2. **Local Commit**: The coordinator tries to serialize the transaction locally via a traditional single-node commit. This commit is local and unreliable but it does not expose any updated values yet to other servers. We implement a simple multi-threaded local commit that resolves contention across threads using a simplified, local version of the ownership protocol (details in Section 7).
3. **Reliable Commit**: If the local commit is successful, the coordinator pushes all updates to the followers for data reliability. In case the coordinator fails in the middle of this process, the followers recover by safely replaying any pending reliable commit of the failed coordinator. Both backup and recovery actions are performed by the *reliable commit protocol* (details in Section 5).

Zeus allows only a single server to modify an object at any time. This server is called the *owner* and is the only node able to use the object to execute write transactions (transactions modifying at least one object). Each object is replicated on one or more backup servers. These backups are active and are called the *readers* of the object; they can perform read-only transactions but not write transactions using the object[2]. Only the owner and the readers store the content of the object. The owner (as a coordinator of write transactions) updates all readers during the reliable commit phase. A user can specify and dynamically change the number of readers (i.e., replicas) of each object, making a trade-off between reliability and replication overhead.

Zeus avoids the conventional distributed commit protocols [46, 64] which are complex [9] because they need to deal with distributed conflict resolution and the uncertainty of commit or abort after faults. Zeus sidesteps these challenges through a simple invariant that an initiated reliable commit is idempotent and cannot be aborted by remote participants. This is accomplished via the exclusive write access of the coordinator and the use of *idempotent invalidations* (§ 5.1), which are sent to all of the remote participants at the start of the reliable commit. In case of a fault, any of the participants can replay the invalidation message which contains enough data to finish the transaction.

Zeus further introduces two key optimizations. Firstly, it supports efficient strictly serializable read-only transactions. Any node that is a *reader* of all objects involved in a read-only transaction is able to execute it without invoking the ownership protocol. A read-only transaction does not require a reliable commit phase; as such, it is light-weight and incurs no network traffic. Consistency of read-only transactions is enforced through invalidation messages, as a read-only transaction cannot execute on an object that is invalidated.

Secondly, a transaction coordinator in Zeus pipelines local execution and commit with the reliable commit, as shown in Figure 2. This is possible because no other server can update

[2]Note that a *reader* is per object, whereas a *follower* is per transaction (potentially spanning multiple objects).

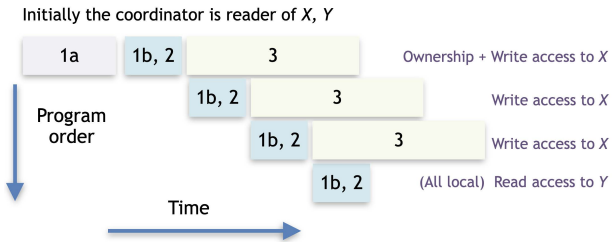

Figure 2. Zeus' pipelined execution of transactions for objects X and Y, on the same coordinator (labels in boxes are the same as in Figure 1).

the objects at the same time. This is guaranteed by the ownership protocol, which ensures that only one node (the current owner) may modify an object. It is thus safe for the coordinator to keep modifying the same object without waiting for the reliable commit to finish. As a consequence, any local transactions to objects for which permissions have already been acquired will not block the application execution.

We also note that we made a conscious design trade-off to make the ownership protocol blocking, to simplify application portability and to make Zeus transactions (the most frequent operations) non-blocking. This means that the application thread stalls when executing an ownership request (phase 1(a) in Figure 1). Such a design is justified because ownership requests are much less frequent than transactions, as discussed in Section 2. It would be straightforward to improve the performance of the ownership protocol, e.g., via a user-mode thread scheduling framework as in [34]; however, that would increase the burden on the developer and likely require re-architecting the application, thus invalidating a key design requirement as laid out in Section 2.

Finally, we specified Zeus' ownership and reliable commit in TLA$^+$ and model checked them. The details are in Section 8.

4 Reliable ownership

The reliable ownership atomically alters object access rights and transfers content between nodes. We start by introducing the main terminology used in the protocol. We then overview its operation without faults and contention, and follow by discussing these other cases.

Access levels, directory and metadata. A node can be the *owner*, a *reader* or a *non-replica* of an object. Each object has at most one owner at any time that has an exclusive write and (non-exclusive) read access to it. An object can also have several other readers with read access. Both the owner and the readers store a replica of the object. A non-replica node has neither the access rights nor the data for the object.

Zeus maintains an *ownership directory* where it stores ownership metadata about each object. This directory is replicated across three nodes for reliability (even if a Zeus deployment has more nodes). The nodes that store directory information are called the *directory* nodes.

	directory	owner	reader(s)	non-replica
data		✓	✓	
ownership metadata	✓	✓		
ownership levels	-	w/r	r	-

Table 1. Data and metadata stored by each node along with their read (r) and exclusive write (w) access permissions.

The directory stores the following metadata for an object:

- *o_state*: the ownership state of the object, which can be *Valid*, *Invalid*, *Request* or *Drive*;
- *o_ts* =<*obj_ver*, *node_id*>: ownership timestamp comprising a monotonically increasing number and a node id;
- *o_replicas*: denotes all nodes storing a replica of the object and their access rights (i.e., the owner and readers).

These ownership metadata are also stored by each object's owner node. The summary of the above is given in Table 1.

4.1 Reliable ownership protocol

Failure- and contention-free operation. An ownership request is illustrated at the top of Figure 3. The coordinator that starts a request is called a *requester* node. The requester assigns a locally unique request id to the request (to be able to match the response) and sets the object's local *o_state* = *Request*. It then sends a *request* (REQ) message with the request id to an arbitrarily chosen directory node, and this node becomes the *driver* of the request. The directory nodes and the object owner help arbitrating concurrent ownership requests to the same object, and are called *arbiters*.

Upon reception of a REQ message, the driver assigns an ownership timestamp *o_ts* to the object and sets its local state to *o_state* = *Drive* ❶. It also sends an *invalidation* (INV) message containing both the request id and ownership metadata to the remaining arbiters (including the current owner) ❷. Assuming no contention for the ownership of the object, each arbiter sets the object's local state to *o_state* = *Invalid*, updates its local *o_ts* and *o_replicas* and responds with an ACK message directly to the requester. Note that we optimize the ownership latency by sending the responses directly to the requester instead of passing via the driver. If the requester is a non-replica and does not have the data of the object, the current owner includes the data in her ACK.

When the requester receives all expected ACK messages, it applies its request locally before responding to all arbiters with a *validation* (VAL) message ❸. To apply the request, it updates the *o_replicas* to specify itself as the new owner, and sets its object's local *o_state* = *Valid*. Finally, upon reception of the VAL message, each arbiter also applies the request in the same way and the request is finished ❹.

Notice that to keep *o_replicas* consistent with the replica placement and the access levels of the object, the requester must apply the request before any of the arbiters. Moreover, once the requester receives all the ACK messages, it unblocks the application. Thus, the application resumes its transaction after 1.5 round-trips, as shown in the top part of Figure 3.

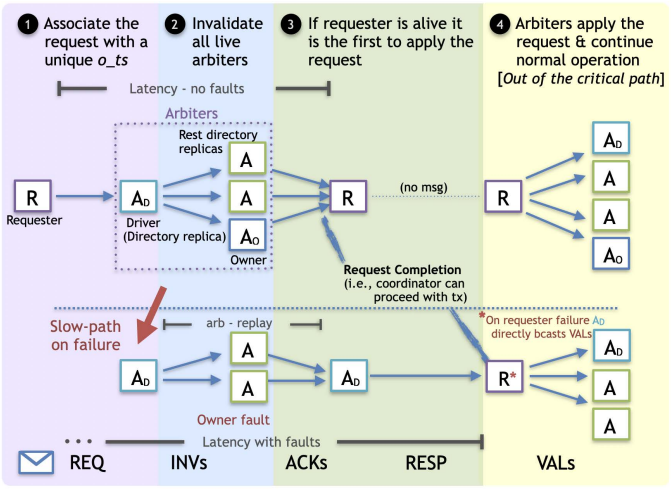

Figure 3. Zeus' ownership protocol with and without faults.

Contention resolution. Zeus uses the *o_ts* timestamp to resolve contending requests. Multiple nodes may issue an ownership request for the same object concurrently through different drivers. Each driver creates a per-object unique timestamp for the request *o_ts* =<*obj_ver* + 1, *node_id*>, using its previous local *obj_ver* and own *node_id* ❶. In case of contention, a driver of one of the contending requests will receive an INV message of another contending request (for the same object) ❷. It will only process the INV message if the *o_ts* in the message is lexicographically larger than its own *o_ts* for the object. This guarantees that there is one and only one winner of each contention. All the drivers whose requests fail send a NACK message to their requesters. Similarly, the owner responds with a NACK directly to the requester if the requested object is involved in a pending transaction (Section 5). Upon receiving a NACK the requester either aborts its ownership request or retries it later.

Failure recovery. The failure recovery procedure starts when the reliable membership is updated after fault detection and the expiration of leases. Each live directory node (and the live owners) update their *o_replicas* removing any non-live nodes. The objects whose owners died will be taken over by a new owner on the next write transaction. After the membership update which increases the epoch id (*e_id*), requests from previous epochs are ignored. This is achieved by including the *e_id* of the current epoch in the INV and ACK messages. The requester and arbiters ignore these type of messages when their *e_id*s differ from their local ones.

A node fault followed by a membership update can leave arbiters of a pending ownership request in *Invalid o_state*. Nevertheless, any arbiter has all the information to replay the idempotent arbitration phase of the ownership request (dubbed *arb-replay*) between the live arbiters and unblock. A blocked arbiter acts as the request driver and initiates an *arb-replay* by constructing and transmitting the same exact INV message using its local state. During *arb-replays*

some arbiter may receive an INV message for a request it has already applied locally (with same *o_ts*). In this case, the arbiter simply responds with an ACK. A basic recovery path from an owner failure is illustrated at the bottom of Figure 3.

Note that in the recovery process the arbitration phase of an ownership request is finalized with ACK messages sent from the arbiters to the driver instead of the requester, as shown in Figure 3. This is done in order to have a single recovery process that covers failures of all nodes including the requester. If the requester is not live the driver directly sends VAL messages to unblock the other live arbiters. Otherwise, for safety, as in the failure-free case, the requester must be the first to apply the request. To achieve that we introduce a new RESP message which confirms the win of the arbitration to the requester; who can then apply the request prior to sending VAL messages to the live arbiters, as before.

4.2 Fast scalable ownership

The Zeus ownership protocol is *scalable* since it 1) does not store directory metadata for each object at every transactional node; 2) does not broadcasts to every transactional node to locate an object's owner. Zeus' ownership protocol has a latency of at most 3 hops (without faults and contention) to reliably acquire the ownership regardless of the node requesting the ownership. We believe this to be the lowest possible latency for a *scalable* ownership protocol. The worst-case latency is incurred when an ownership request originates from a non-replica node where neither the owner nor the requester are co-located with the object's directory metadata. To proceed, the requester must receive the latest value of the object. In order to locate the object, the requester should first contact the directory. The directory will forward the request to the owner, which, in turn, will send the value to the requester, resulting in 3 hops. Note that if the requester is co-located with a directory replica, the first hop is eliminated and ownership is acquired after just one round-trip (2 hops) to the owner.

5 Reliable commit

Zeus reliable commit protocol is responsible for propagating the updates made by a local transaction to all of the followers (illustrated in Figure 4). For clarity, we start by describing the information maintained by the protocol. We next overview the operation without faults, and then discuss the case with failures. Finally, we present two optimizations: pipelining and local read-only transactions from all replicas.

(Meta)data. Each replica (i.e., the owner and readers) keep the following information for an object:

- *t_state*: the state of the object, which can be either *Valid*, *Invalid* or *Write*;
- *t_version*: the version of the object, which is incremented on every transaction that modifies the object;
- *t_data*: the data of the object stored by the application.

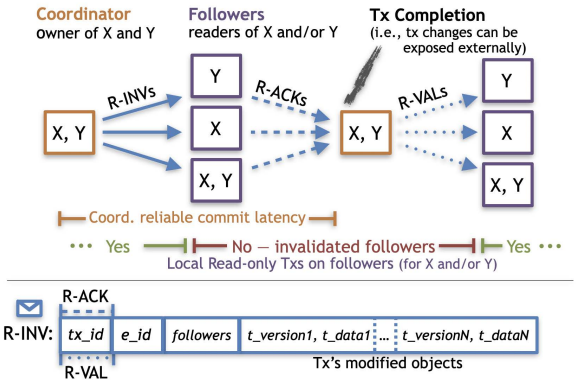

Figure 4. Zeus' reliable commit protocol and its messages.

For every transaction, at the beginning of reliable commit, the coordinator generates a unique *tx_id* = <*local_tx_id*, *node_id*>, where *node_id* is its own id and *local_tx_id* is a locally unique, monotonically increasing transaction id.

5.1 Reliable commit protocol

Failure-free operation. At the end of the Local Commit phase, the transaction coordinator updates the *t_data* of all modified objects with its private copies created during the Prepare & Execute phase. It also increments their *t_versions* and sets *t_state* = *Write* — for pending reliable commit.

At the beginning of the Reliable Commit phase, the coordinator broadcasts an *invalidation* (R-INV) message to all followers. As shown at the bottom of Figure 4, this message contains the *tx_id*, the current epoch id (*e_id*) and the *node_id*s of all followers. For each updated object, it also contains the new *t_version* and *t_data*. The coordinator temporarily stores the R-INV message locally.

Upon receiving an R-INV message, a follower checks if the received and the local *e_id* match, if not the message is ignored. If they match, the follower goes through each updated object and compares its local *t_versions* with that of the message. In case an object's local version is greater or equal, it skips the update of that object. Otherwise, it updates the local *t_data* (the actual content of the object) and *t_version* with the new ones from the message, and sets its local *t_state* = *Invalid* — denoting that the object has a pending reliable commit. A follower then responds to the coordinator with an R-ACK message containing the same *tx_id* and temporarily stores the R-INV.

Once the coordinator receives R-ACKs from all the followers, it reliably commits the transaction locally by changing the *t_state* of each updated object to *Valid*. Subsequently, the coordinator broadcasts a *validation* (R-VAL) message containing the *tx_id* to all followers and discards the previously stored R-INV message of the transaction. When a follower receives an R-VAL message for which it has already stored an R-INV message (with same *tx_id*), it sets the *t_state* of all objects previously updated by the transaction to the *Valid* state if and only if their *t_version* has not been increased. It then discards the stored R-INV message.

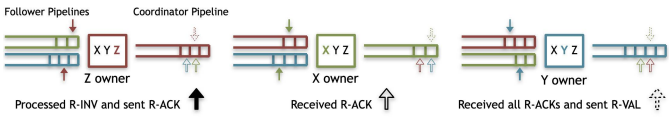

Figure 5. Zeus' per-node (in reality per-thread §7) pipelines.

Reliable replay under failures. A node failure triggers a membership reconfiguration, where the epoch id (*e_id*) is increased and the set of live nodes is updated. Subsequently, the ownership protocol stops accepting requests for objects whose owner node is not live in the current membership.

At this point, each locally stored R-INV message on any live node represents a pending transaction in the Reliable Commit phase. A live node, replays its own pending reliable commits and those from the failed nodes. This is accomplished by first updating the local pending R-INV messages (issued or received) with the new *e_id* and by removing all non-live nodes from followers. The messages are then re-sent and handled as explained before. A follower who receives an R-INV message with the latest *e_id* for a transaction (*tx_id*) that it has previously stored locally simply ignores its content and responds with an R-ACK . Although multiple nodes may replay the reliable commit phase of the same transaction, all relevant R-INV messages are idempotent containing the same *tx_id* (and *t_versions*) so only one can apply updates.

When a node has no more pending reliable commits (R-INV messages) from nodes that are not live, it informs the ownership protocol that it has finished the recovery (Section 4). Once all live nodes finish the recovery, the ownership protocol starts accepting again all ownership requests as normal.

5.2 Non-blocking transaction pipelining

We further introduce transaction pipelining to avoid blocking the application at the coordinator during replication (illustrated in Figure 2). This is possible because a locally (unreliably) committed transaction at the coordinator cannot be aborted. Thus, the coordinator can proceed using its locally committed values with certainty.

However, Zeus also needs to maintain the strict serializability on each backup replica. Thus, it requires that followers respect the pipeline order of the coordinators when applying updates. For this, Zeus uses *tx_id* =<*local_tx_id*, *node_id*> which is transmitted in every R-INV message and contains both the local transaction order within the node *local_tx_id* and the *node_id*. As a result, although there could be several pending causally-related reliable commits, all will be applied in the correct order as specified by the *local_tx_id*.

Note that the ordering is enforced only within each different pipeline as shown in Figure 5. This is because an object's owner change (i.e., when an object switches pipelines) is not approved until all pending reliable commits with that object have been completed (Section 4). Thus, an object cannot be involved in pending transactions from two different coordinator nodes and the ordering across coordinators does not matter. We further optimize this by enabling per-thread (instead of per-node) pipelines via our choice of local commit as explained in Section 7. The pipelining optimization also reduces the number of R-ACK and R-VAL messages, since sending a message with a *tx_id* implies the successful reception and processing of all previous messages in that pipeline.

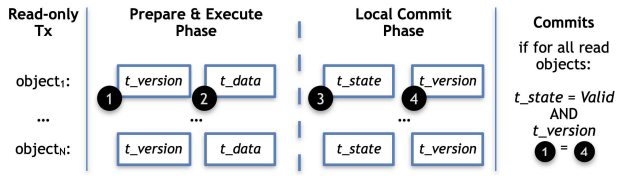

Figure 6. Zeus' consistent read-only transactions on readers.

A node may not be a follower of all R-INVs, and thus may receive only a partial stream of a pipeline. An extra condition is needed for when such followers can *apply* an R-INV. A follower applies an R-INV if for the previous *local_tx_id* (slot) of the pipeline it has either applied an R-INV or has received an R-VAL. The latter occurs for a transaction follower *F* who was not also a follower of the previous slot in the pipeline. To facilitate this, during the broadcast of an R-INV, the coordinator piggybacks a *prev-VAL* bit if it has broadcasted R-VALs for the previous slot. Otherwise, it includes *F* in the R-VAL broadcast of that previous slot. Finally, after a coordinator's failure, an R-INV is considered as a pending reliable commit and is replayed by a follower *iff* that follower has not only received but also applied the R-INV message.

5.3 Read-only transactions

Zeus optimizes read-only transactions by allowing them to be executed locally from any replica that stores all relevant objects, regardless of the ownership level (read or write), and without compromising strict serializability. This is enabled by three ideas. First, read-only transactions do not need to communicate any updates to other replicas. Second, a verification-based scheme can be applied to exploit the local object versioning and ensure a consistent snapshot across all reads of a read-only transaction. Finally, the reliable commit guarantees that all replicas are invalidated before any updated state is exposed externally by the readers. We elaborate on the latter before discussing the read-only protocol.

Invalidation-based reliable commit. A locally committed write transaction does not reliably commit on the owner unless it has invalidated all its followers (i.e., the readers of modified objects). As noted before, a reader which applies an invalidation to its local object also updates its object's local value with the newly received value. Thus, it cannot return neither the old value nor the new one as the object has been invalidated. The reader can return the new value only after it receives the R-VAL message and validates its local object.

Simply put, there is a transitioning period until a reader can safely return the new value. That period ends once all readers of a modified object have stopped returning the old value and have received the new one. If a reader was to prematurely return the new value (i.e., before receiving the

R-VAL message and the end of that period), two things could go wrong. First, another reader who has not yet invalidated the object could subsequently return the old value and compromise consistency. Second, if all nodes that have received the new (not yet reliably committed) value fail[3], then the prematurely returned value would be permanently lost.

Read-only protocol. Consequently, in Zeus, a read-only transaction completes after only two phases as shown in Figure 6 and described next. In the Prepare & Execute phase, the coordinator of a read-only transaction sequentially reads and buffers the $t\_version$ and the value (t_data) of each local object as specified by the transaction. In the Local Commit phase, the coordinator checks if all accessed objects are in $t\_state = Valid$ before verifying that all $t\_versions$ have remained the same. If so, the transaction commits successfully. Otherwise, there is an ongoing conflicting (local or remote) reliable commit and the read-only transaction is aborted.

Use-case. Apart from the obvious performance benefit, one example where the read-only optimization is useful is control/data-plane applications, such as in a cellular networks. There, write transactions are executed by a control-plane node (the Zeus owner), for instance to configure routing, while all data-plane nodes (i.e., Zeus readers) can perform consistent read-only transactions locally, e.g., for forwarding.

6 Discussion

6.1 Distributed commit vs Zeus

Traditional datastores statically shard objects and execute reliable transactions in a distributed manner across servers. This poses two challenges. The first is accessing the objects. Static sharding schemes do not guarantee that all objects accessed by a transaction reside on the same node. Frequently, one or more objects in a transaction are stored remotely. In this case the execution stalls until the objects are fetched – sometimes sequentially (e.g, pointer chasing or control flow).

The second challenge is handling concurrent transactions on conflicting objects. If two nodes try to commit transactions on conflicting objects simultaneously, one of them has to abort. Detecting and handling these conflicts under the uncertainty of faults needs extra signaling across nodes. Thus, transactional systems based on distributed commit need numerous round-trips to commit each transaction (e.g., see FaSST). Moreover, a node cannot start the next transaction on the same set of objects until the commit is finished, as it cannot be sure that it will not have to abort. This introduces several round-trips of delay in the critical path of the commit and significantly reduces the transactional throughput.

Zeus replaces remote accesses and distributed commit with its (occasional) ownership, local accesses and reliable commit to addresses the two main issues mentioned above and accelerate workloads with locality. Firstly, the ownership makes objects accessed by a transaction accessible locally most of the time, which avoids stalls during the execution. Secondly, only a single node (the owner) can execute a write transaction on an object at a time, so a transaction cannot be aborted remotely, commits after a single round-trip and is pipelined. Zeus' reliable commit also allows local and consistent read-only transactions from all backups.

Unlike distributed commit, Zeus' ownership is a protocol specialized for single-object atomic operations (including migration). Zeus resolves concurrent ownership requests in a decentralized way, and applies an idempotent scheme to tolerate faults without extra overheads on the common failure-free case. This makes acquiring ownership reliable yet fast (1.5 round-trips) during fault-free operation.

6.2 Other details

Cost of ownership vs. remote access. The object size influences the cost of acquiring ownership for it by a non-replica node similarly to a remote access, since in the fault-free case the value is included in a single ownership message as in the response of a remote access. A reader acquires the ownership without the value and thus is not influenced by its size. The reliability of Zeus' ownership comes with a higher message cost compared to a remote access. These are small constant messages with cost amortized over several local accesses in workloads with locality. Nevertheless, for workloads without enough locality, that cost renders Zeus less suitable than remote accesses and distributed commit.

Deadlocks. Zeus currently circumvents deadlocks via a simple back-off mechanism. For Zeus, such a situation may arise only early in a transaction (i.e., in the Prepare & Execute phase) – when requesting ownership for an object. This manifests with repeated failed ownership requests, after which Zeus aborts and retries a transaction with an exponential back-off. In practice, deadlocks in Zeus are rare because transactions on the same object are mostly executed on the same server by virtue of load balancing. For deployments where that is not the case, a more sophisticated scheme such as the one proposed by Lin *et al.* [40] may be considered.

Distributed directory. For simplicity, Zeus uses a single directory for all objects in the deployment. The directory is replicated for fault-tolerance, and the ownership protocol is lightweight and is designed to balance the load across all the directory replicas. However, a single replicated directory may become a scalability bottleneck at large deployment sizes or when locality is limited. In such cases, a distributed directory scheme (i.e., using consistent hashing on an object to determine its directory nodes) should be used instead.

Sharding request types. Zeus exploits the ownership protocol for other types of sharding requests, such as reliably removing a reader. For example, when a non-replica acquires the ownership of an object, the total number of replicas increases. To keep the initial replication degree and avoid increasing the cost of reliable commits, we invoke the ownership protocol out-of-the-critical-path to discard a reader.

[3]That is a smaller number of nodes than the replication degree.

Write transactions with opacity. Apart from strict serializability, Zeus provides an additional guarantee that all write transactions see a consistent snapshot of the database even if they abort. This is also referred to as *opacity* [26]. Opacity further enhances Zeus' programmability since by preventing inconsistent accesses in write transactions it relieves the programmer from the effort of handling those cases.

7 System

We have built a custom in-memory datastore and implemented the Zeus protocols on top of it. In this section we briefly discuss the implementation details.

An application communicates with the datastore through a transactional memory API that consists of primitives to create and manage memory objects of different sizes. This includes implementations of `malloc` (create an object), `free` (destroy an object), `tr_open_read` and `tr_open_write` (for marking object as used in a transaction for reading and writing). Each transaction starts with a create transaction call `tr_create` (for write) or `tr_r_create` (for read-only transaction), followed by an arbitrary code that can invoke the above APIs, and finishes with a `tr_commit` (or `tr_abort`), at which point the local commit starts (aborts). This is a low-level API, very similar to the one used by FaRM, and it allows great flexibility to build further abstractions on top of it.

The datastore is implemented in C over DPDK, and it consists of two parts. One part is the datastore module that runs as a separate process implementing the main datastore functionality. The other part is the Zeus library that is linked to any application over shared memory without limiting its architecture (it can be a separate process, container, etc).

The datastore module implements the Zeus protocols, the transactional memory API, and a reliable messaging between nodes. Zeus communicates between nodes using a custom reliable messaging library we built on top of DPDK. The datastore module also includes a customizable, application-aware load balancing functionality, as described in Section 3.

Both application and the datastore modules can run in multiple threads. In the evaluation, we use up to 10 application and 10 datastore worker threads. These threads are pinned to their own cores. We also use one core for DPDK.

We implement a simple multi-threaded Local Commit (Section 3) using the same intuition as for the overall Zeus. Each thread that executes a transaction has to become the owner of each object. However, this ownership is local and managed through standard locking. We leverage the aforementioned load balancer to enforce locality across the threads and increase concurrency. Apart from simplicity, this also enables transaction pipelining to be applied on a per-thread bases which increases the overall concurrency of reliable commits.

Currently, porting an application to Zeus requires manual code modification on pointer accesses, similarly to prior work (e.g., as in FaRM). However, this can be automatized at a compiler level, as performed by Sherry *et al.* [63].

	characteristic	tables	columns	txs	read txs
Handovers	large contexts	5	36	4	0%
Smallbank	write-intensive	3	6	6	15%
TATP	read-intensive	4	51	7	80%
Voters	popularity skew	3	9	1	0%

Table 2. Summary of evaluated benchmarks.

8 Evaluation

Formal verification. We specified the ownership protocol and the reliable commit of Zeus in TLA$^+$ and model checked them in the presence of crash-stop failures, message reorderings and duplication. We have verified them against several key invariants including the following:

- Live nodes[4] in $t\_state{=}Valid$ have always consistent data.
- All live arbiters in $o\_state{=}Valid$ agree and correctly reflect the owner and reader nodes of the object.
- At any time there is at most one owner and that owner stores the most up-to-date value of the object.

The detailed protocol specifications and the complete list of the model-checked invariants can be found online[5].

Locality in workloads. We begin by briefly analysing the locality of access patterns in workloads. For this, we report the fraction of remote transactions of three workloads, spanning the telecommunication, financial, and trade sectors.

- **Boston cellular handovers**: As explained in Section 2, in a cellular workload, remote transactions are caused by remote handovers. To evaluate the real-world frequency of remote handovers, we use the population and mobility model from Boston metropolitan area [12] with the reported averaged daily commute of 100km. We assume base stations are uniformly spread through the area at a distance of 1km (with a typical coverage of a macro cell [32] and a common ratio of cells per population [45]). These are sharded across all nodes in a deployment. As the number of nodes increases, the number of remote handovers also increases, up to 6.2% for six nodes. In summary, for a setup where 5% of all transactions are handovers and out of these 6.2% of handovers are remote (in a six node deployment), there are in total 0.31% remote transactions.
- **Venmo transactions**: We use the most recent public Venmo dataset [60] with more than seven million financial transactions to analyze the fraction of remote transactions. We partition the users to nodes, but still observe 0.7% and 1.2% of remote transactions for 3 and 6 nodes, respectively.
- **TPC-C**: We mathematically analyze the number of remote transactions in the TPC-C benchmark, which is considered representative for industries that trade products. In TPC-C, only a small fraction of new-order and payment transactions may result in remote accesses. We find that just 2.45% of the transactions in the benchmark are remote.

[4] By construction non-live nodes cannot compromise safety because *e_ids* prevent them from participating in either transaction or ownership requests.
[5] https://zeus-protocol.com

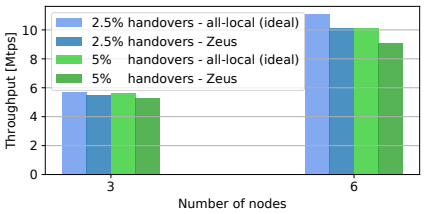

Figure 7. All-local (ideal) vs. Zeus for 2.5% and 5% handovers on 3 and 6 nodes.

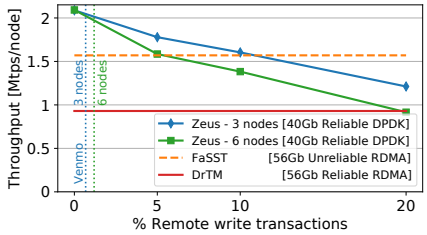

Figure 8. Smallbank while varying remote write transactions.

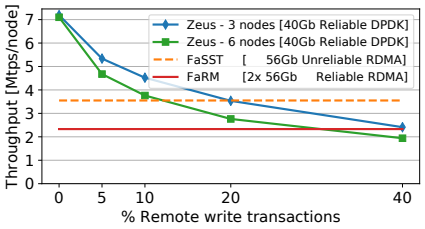

Figure 9. TATP while varying remote write transactions.

We empirically evaluate benchmarks related to cellular and financial transactions (i.e., Handovers and Smallbank). While promising in terms of locality, we leave the experimental evaluation of TPC-C for future work because our current implementation of Zeus does not support range queries.

Experimental testbed. We run all our experiments on a dedicated cluster with six servers. Each server has a dual socket Intel Xeon Skylake 8168 with 24 cores per socket, running at 2.7GHz, 192 GB of DDR4 memory and a Mellanox CX-3 card. We use and pin all our threads into the first socket only, where the network card resides. All servers communicate through a Dell S6100-ON switch with 40 Gbps links.

We first evaluate Zeus on several benchmarks (summarized in Table 2); including three benchmarks discussed in Section 2 and the TATP benchmark [49] to further study Zeus' limits over FaSST and FaRM. For benchmarks, as in prior work [34], we consider 3-way replication and enough colocated clients to saturate each evaluated system. The initial sharding of all systems is the same. Unlike Zeus, baselines do not support dynamic sharding (i.e., ownership). We were not able to run the baseline systems FaRM, FaSST and DrTM on our platform, but the hardware used in their evaluation is similar, so we report numbers from their papers [20, 34, 71]. We conclude by demonstrating the ease of porting legacy applications onto Zeus by porting and evaluating a cellular packet gateway, an Nginx server and the SCTP protocol.

8.1 Handovers

We start our evaluation with a cellular handovers benchmark. We evaluate three operations described in Section 2: a handover (consists of two transactions, one at the start and one at the end), a service request and a release (each a single transaction). We implement them as defined in 3GPP specification, on top of Zeus. All transactions are write transactions. A typical cellular phone context for these operation is large and many parts of it get modified so we need to commit about 400B of data per transaction.

Recall that mobile users perform both handovers and all other requests, while the stationary users only perform other requests (i.e., no handovers). In our evaluation, we vary the ratios of the total number of handovers versus the total number of requests (handovers, service requests and releases), each modeling different mobility speeds in the network. A typical cellular network has 2.5% handovers [45], and we also evaluate the 5% case corresponding to doubling the mobility.

We run a benchmark on a population of 2M users out of which 400k are mobile. We use the typical cell network provisioning as reported in [45, 55], scaled to 2M users (requiring 1000 base stations). Not all handovers will involve ownership transfers because some will occur between objects of the same node. For the ratio of remote handovers we use the numbers we analyzed from the Boston metropolitan area.

In our evaluation we vary the number of nodes in the system, and plot the total throughput for the two ratios as well as for all local transactions. This is shown in Figure 7. We see that the difference between Zeus and the perfect sharding is at most 9%. This is because there a large fraction of the transactions is local, and we have less than 0.5% ownership requests. We also see that the performance scales linearly with the number of nodes, even though there are more transactions with ownership transfers for a larger number of nodes. Lastly, we note that prior works have not studied the handover benchmark; as such, there are no published numbers for state-of-the-art systems to compare against.

8.2 Smallbank

Smallbank is a benchmark that simulates financial transactions [10]. It is write intensive with 85% write transactions. Out of them, 30% modify two objects and the rest modify 3 or more objects per transaction. All read transactions access 3 objects. We use the same access skew on objects as in FaSST.

Smallbank does not specify which pairs of users transact with each other, hence it cannot be used to infer real-world transaction locality. To understand how much the degree of locality affects Zeus, we start increasing the number of transactions that require an ownership change, until Zeus breaks even with the beaselines. This is shown in Figure 8. We see that running Smallbank with the real-world remote transactions, as observed in the Venmo, Zeus outperforms FaSST and DrTM by about 35% and 100%, respectively. Recall that neither FaSST nor DrTM support dynamic sharding so any small and gradual change in access pattern will eventually lead to an almost random placement and most requests being remote, which is what we show here. As expected, Zeus throughput drops as the remote transactions increase and the trend between three and six nodes remains the same. As

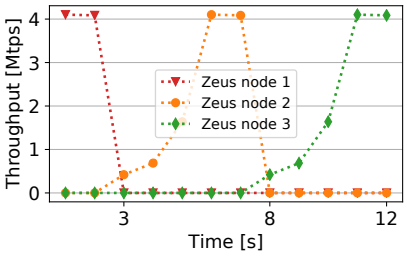

Figure 10. Voter Performance when moving 1M objects across nodes.

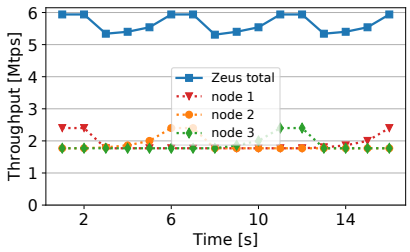

Figure 11. Voter Performance when registering votes and moving objects.

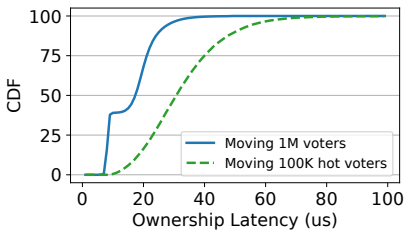

Figure 12. CDF of Zeus ownership request latency for Voter experiments.

long as less than 5% (20%) of transactions require ownership change, Zeus provides a benefit over FaSST (DrTM).

Reliable lower-end networking. Note that unlike FaSST, Zeus implements reliable messaging with its overheads. While this reduces Zeus' performance, it allows Zeus to gracefully tolerate message losses. In contrast, FaSST must kill and recover a node for each lost message. Also, FaSST uses 56Gb RDMA. DrTM similarly leverages 56Gb RDMA and relies on hardware transactional primitives for its performance. Zeus uses a 40Gb non-RDMA networking and it does not depend on hardware-assisted transactions for its performance.

8.3 TATP

We next evaluate the TATP benchmark [49], which gives us a second point of comparison with other state-of-art systems [21, 34]. It is read intensive, with 80% read and 20% write transactions. We use 1M subscribers per server, as in FaSST. Similarly to the Smallbank benchmark, we vary the fraction of transactions that require an ownership change. The total throughput is shown in Figure 9. We see that when the fraction of remote requests is small, Zeus outperforms FaSST and FaRM by up to 2× and 3.5× respectively.

As discussed in the Smallbank study, neither FaRM nor FaSST allow dynamic sharding so they end up issuing remote requests whenever there is a changing access pattern. Zeus keeps the requests local by moving objects, and it is especially effective for a read-dominant benchmark like TATP, since there is little overhead on reads. We also see that as long as there are fewer than 20% (40%) of write transactions with ownership requests, Zeus outperforms FaSST (FaRM). Again, these thresholds are higher than in the case of Smallbank due to read-dominant workload. The performance trend of Zeus for three and six nodes is the same as in Smallbank.

8.4 Voter

Voter is a benchmark that represents a real-time phone voting system [19]. Using three nodes, we simulate 20 contestants in a popularity show with 1M unique voters, each identified by their phone number. Each voter can vote for one contestant during one phone call and there is a limit how many times each voter may vote per unit of time. Therefore, each phone voting operation updates two objects: the total votes of a contestant and the voting history of the voter.

In this benchmark, we evaluate the ability of Zeus to move popular objects around, as discussed in Section 2. In the first experiment, we evaluate the performance of the ownership transfer protocol in isolation. We have 1M voters that generate 4M transactions per second (in comparison, E-store [66] evaluates up to 200Ktps). At time 2s, we move all voter objects from node 1 to node 2, and at time 7s, we move them again to node 3. The results are shown in Figure 10. We see that the full move takes 4s, implying that a single worker thread (out of ten) can move 25k objects per second.

In the second experiment, we evaluate the performance of ownership transfers concurrently with transaction processing. We have 1 very popular contestant that has 100k voters voting for her, generating 700Ktps. All other voters vote for other contestants and generate about 5.3Mtps in aggregate. In this experiment, a single application and worker thread process the popular voter. As in the previous experiment, at times 2s, 6s and 10s, we start moving the object corresponding to the popular contestant to another node. The results are shown in Figure 11. We see that the single worker thread still performs 25k ownership requests per second (moving 100k objects in 4s) while at the same time the rest of the system completes 5.3Mtps. This shows that the performance of ownership is not impacted by concurrent transactions.

Figure 12 shows the latency distribution of ownership transfer. This metric is important since an application thread is stalled during an ownership transfer, which allows easy porting of applications. We see that the mean latency and the 99.9th percentile are close during the first voter experiment; 17 and 36 μs, respectively. Under high load and while moving hot objects (during the second experiment) the mean latency is slightly higher at 29 μs, and the 99.9th percentile is 83 μs. This makes Zeus 3 times faster than Rocksteady[6] [37] in the 99.9th percentile despite moving hot objects under load.

8.5 Legacy applications

One of the advantages of Zeus is that it is easy to port existing applications on it. Different applications assume different multi-threading or multi-process models, with different role for each thread (process). They also often take dependencies on various external libraries and OS calls. FaRM, FaSST and

[6]Evaluated in similar setup with DPDK networking over 40Gb CX-3 NICs.

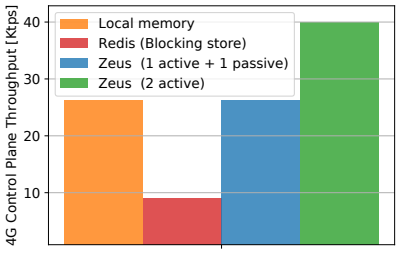

Figure 13. Cellular packet gateway control plane performance.

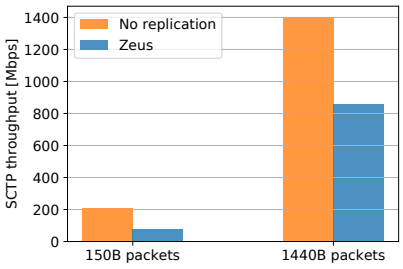

Figure 14. SCTP performance.

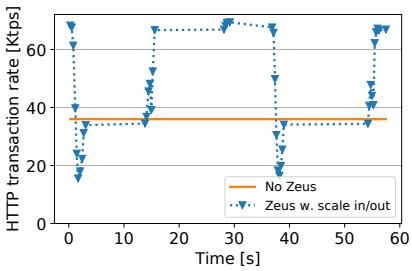

Figure 15. Nginx performance in a scale-in / scale-out scenario.

DrTM have to wait on each remote access. To mitigate this latency, they assume transaction multiplexing via custom user-mode threading (e.g., co-routines or Boost user-threads in FaSST); however, this makes it difficult to integrate with many legacy applications.

As explained in Section 3, Zeus takes a different approach. Since most transactions are pipelined and do not block the application thread, there is no need to re-architect a legacy application. Zeus only blocks the application during the ownership requests, which are infrequent.

In order to verify the claim about portability, we port and evaluate three existing applications on top of Zeus: the control plane of a cellular packet gateway, the SCTP transport protocol and an Nginx web server.

Cellular packet gateway. Cellular packet gateway is a virtual network function in a cellular network that forwards all packets from mobile users. It has a control and data plane. The control plane performs service request and release operations, as described in the handover benchmark (but not the handovers themselves). Each of these operations is one transaction. We use the OpenEPCv8 [53] 4G implementation of the cellular core control plane. We remove the legacy datastore and instrument every access to use Zeus. We use a custom load generator to create test workloads with service and release requests. We test the gateway without any datastore (all data in local memory and no replication), using an off-the-shelf Redis datastore without replication, and Zeus.

The results are shown in Figure 13. Requests to Redis are remote and, due to the OpenEPC design, the application thread blocks on every request. This is why Redis performance is lower than 10Ktps even without replication, and illustrates the challenges due to blocking when porting existing applications. Zeus with a single active node (and 1 passive replica) is as fast as the gateway with local accesses and no replication. This is because the bottleneck is in parsing and processing the signalling messages, not in the datastore access. When we use both nodes as active (being each other replica), the throughput is 60% higher. We are not able to scale beyond three nodes due to limitations of our signal generator, which cannot saturate more than two Zeus nodes.

SCTP transport protocol. SCTP is commonly used in the cellular control plane to offer a degree of fault tolerance on network issues. For fault tolerance, SCTP natively supports multi-homing and is able to switch from one access network to another in case of a network failure, without dropping a connection. However, current SCTP implementations cannot survive a node failure as the connection state is not replicated. If an SCTP connection fails, all active users drop calls. Moreover, it is not easy to virtualize SCTP state as the protocol is originally implemented as a part of a Unix kernel.

To demonstrate Zeus efficiency and the ability to support legacy applications, we port an implementation of SCTP protocol [59] to Zeus and replicate all changes to the connection states. We implement each packet transmission, reception and a timer event as a single transaction. Thus, any node failure will be perceived by the peers as a network loss, and dealt with by the protocol. SCTP uses standard BSD macros for basic data structures (e.g., lists, hash tables) that are compatible with Zeus memory interfaces (described in Section 7). We are able to keep the original SCTP design (timer, RX and TX threads) as we do not have to deal with thread blocking.

We use a standard iperf3 client to generate a single SCTP flow to a Zeus server running SCTP. All state is replicated on another Zeus server. Figure 14 shows the throughput of the single flow for different packet sizes. For large packet sizes, Zeus is 40% slower than vanilla SCTP with no modifications. This is because SCTP has a complex state that is modified for every packet and 6.8 KB of data has to be replicated (note that we have not spent any time optimizing state access and providing read-only accesses). The difference is higher for smaller packets because of the replication overhead. However, we argue that this is fine for the *control plane*, where the reliability is more important than speed. We also note that Zeus pipelined transactions are important for the SCTP case with a few flows because many consecutive transactions access the same object and do not have to wait for the reliable commit of the previous transaction (§ 5.2).

Nginx web server. Finally, we evaluate the session persistence routing mode [50] of an Nginx web server on top of Zeus. In this mode, Nginx runs as an application-layer load balancer. It looks up a specific cookie in an HTTP request and chooses an end destination based on its value. Session persistence is not available in the open source version of Nginx so we implement our own variant using the Zeus datastore.

If the cookie is found in the replicated datastore, we route the request to the destination stored in the entry. If not, we randomly select one of the two HTTP back-end servers and store it to the datastore (replicated over two nodes).

A client creates a number of requests for a single small HTTP page. Initially, all packets requests are processed by the same Nginx server node using a single core. We then emulate a scale-out and a scale-in by adding and removing another server node, and spreading the load across all available nodes. The number of forwarded HTTP requests processed by Nginx is shown in Figure 15. We see that the Nginx performance with Zeus is the same as without Zeus, showing that the bottleneck is in the application and not in the datastore. We also see that it seamlessly scales in and out as the number of servers change. Again, this illustrates an ease of portability of an existing legacy application to Zeus.

9 Related work

Recent works on in-memory distributed transactions present distributed commit protocols that leverage modern hardware to achieve good performance with strong consistency, but do not fully exploit locality [15, 20, 21, 34, 38, 70]. Some systems expose object locality which allows programmers to implement locality-aware optimisations [4, 20], but, unlike Zeus, object relocation is costly and burdens the programmer.

There are also works that mitigate the cost of distributed transactions but impose other constrains. For example, some mandate determinism [30, 41, 57, 68], and are limited to non-interactive transactions that require the read/write sets of all transactions to be known prior to execution [58]. Others adopt epoch-based designs to amortize the cost of commit across several transactions [16, 42, 43]. Contrary to those, Zeus enhances programmability and supports fully-general transactions that need not wait the end of epochs to commit.

Object partitioning has been used to improve performance of distributed transactions. Typically, objects are partitioned and migrated periodically to improve locality [1, 17, 23, 37, 39, 54, 61, 66]. In geo-distributed systems, object migration can significantly reduce WAN traffic [14]. Facebook's Akkio [7] splits data in μ-shards which migrates across datacenters to leverage locality in workloads. Similarly, SLOG [57] deploys a periodic remastering scheme over a deterministic database to reduce across-datacenter round-trips, but mandates coordination within a datacenter. Other works also exploit locality to reduce across-datacenter round-trips [25, 67, 74]. In contrast, Zeus infers locality and moves the object eagerly on the first access, supports non-deterministic transactions, and reduces coordination within the datacenter.

Zeus protocols bear similarity to cache coherence in multi-processor systems. Cache coherence protocols move the cache lines to the requesting node on access. Cache coherence protocols have been used to implement hardware transactions [29]. Zeus builds on ideas in Hermes [35], which adapted concepts from cache coherence and applied them to enforce strong consistency for replicated in-memory datastores. Hermes allows for local reads and fast reliable updates to individual objects from all replicas; however, it does not support multi-object reliable transactions or ownerships.

Distributed shared memory (DSM) provides an abstraction of single shared memory space built on top of a collection of machines (e.g. [6, 13, 65]). Similarly to Zeus, many DSMs use cache coherence protocols, moving data to the accessing node, but, unlike Zeus, most focus on single-object consistency. A few support transactions (e.g., [11, 73]) but relax consistency and/or forfeit availability for performance.

Several works on software transactions have used ownership-related ideas albeit on a single-node context [18, 28, 44]. L-Store [40] optimizes for locality using ownerships in a distributed local area setting, but only supports durable transactions (i.e., without replicas and availability). In contrast, Zeus enables strictly-serializable transactions and fast ownerships over a replicated deployment that facilitates availability and local read-only transactions from any replica.

Akin to Zeus' local commit, PWV [24] enables early write visibility, as soon as a transaction executes all statements that could cause it to abort. However, transactions in PWV forfeit strictness and need determinism. In contrast, Zeus transactions exploit locality and afford strict serializability.

An area that has looked into datastores with dynamic sharding are virtualized network functions. Several have built custom datastores to exploit locality (e.g., [63, 72]), but they do not deliver on other desired requirements – speed, availability or consistency. Others have forgone locality benefits [33, 36] and rely on external datastores (e.g., [52]).

10 Conclusion

Many real-world applications exhibit high access locality. Zeus leverages this to depart from the conventional distributed transaction design. Instead of executing a transaction across nodes, Zeus brings all objects to the same node and executes the transaction locally. It does so via two new reliable protocols: one for fast localized transactions with replication, and one for efficient object ownership. Another benefit of Zeus is the ease of porting existing applications on top of it, as localized transactions can pipeline replication without blocking the application. Zeus is up to 2× faster than state-of-art systems on TATP benchmark and up to 40% on Smallbank while using lower-end networking. It can move up to 250k objects per second per server and process millions of transactions per second. Zeus can run many industry standard applications without any re-architecting. We believe that Zeus can accelerate the uptake of reliable in-memory databases for a wide range of applications in the near future.

Acknowledgments. We thank our shepherd, Liuba Shrira, as well as Vitor Enes, Vasilis Gavrielatos, Vijay Nagarajan and our reviewers for their constructive comments and feedback. This work is supported by the EPSRC grant EP/L01503X/1 and by Microsoft Research via its PhD Scholarship Program.

References

[1] Michael Abebe, Brad Glasbergen, and Khuzaima Daudjee. 2020. DynaMast: Adaptive dynamic mastering for replicated systems. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE, 1381–1392.

[2] Atul Adya, Robert Grandl, Daniel Myers, and Henry Qin. 2019. Fast Key-Value Stores: An Idea Whose Time Has Come and Gone. In *Proceedings of the Workshop on Hot Topics in Operating Systems* (Bertinoro, Italy) *(HotOS '19)*. Association for Computing Machinery, New York, NY, USA, 113–119. https://doi.org/10.1145/3317550.3321434

[3] Atul Adya, Daniel Myers, Jon Howell, Jeremy Elson, Colin Meek, Vishesh Khemani, Stefan Fulger, Pan Gu, Lakshminath Bhuvanagiri, Jason Hunter, Roberto Peon, Larry Kai, Alexander Shraer, Arif Merchant, and Kfir Lev-Ari. 2016. Slicer: Auto-Sharding for Datacenter Applications. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (Savannah, GA, USA) *(OSDI'16)*. USENIX Association, USA, 739–753.

[4] Marcos K. Aguilera, Arif Merchant, Mehul Shah, Alistair Veitch, and Christos Karamanolis. 2007. Sinfonia: A New Paradigm for Building Scalable Distributed Systems. In *Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles* (Stevenson, Washington, USA) *(SOSP '07)*. Association for Computing Machinery, New York, NY, USA, 159–174.

[5] Mukhtiar Ahmad, Syed Usman Jafri, Azam Ikram, Wasiq Noor Ahmad Qasmi, Muhammad Ali Nawazish, Zartash Afzal Uzmi, and Zafar Ayyub Qazi. 2020. A Low Latency and Consistent Cellular Control Plane. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication* (Virtual Event, USA) *(SIGCOMM '20)*. Association for Computing Machinery, New York, NY, USA, 648–661. https://doi.org/10.1145/3387514.3406218

[6] Cristiana Amza, Alan L. Cox, Sandhya Dwarkadas, Pete Keleher, Honghui Lu, Ramakrishnan Rajamony, Weimin Yu, and Willy Zwaenepoel. 1996. TreadMarks: Shared Memory Computing on Networks of Workstations. *Computer* 29, 2 (Feb. 1996), 18–28. https://doi.org/10.1109/2.485843

[7] Muthukaruppan Annamalai, Kaushik Ravichandran, Harish Srinivas, Igor Zinkovsky, Luning Pan, Tony Savor, David Nagle, and Michael Stumm. 2018. Sharding the Shards: Managing Datastore Locality at Scale with Akkio. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation* (Carlsbad, CA, USA) *(OSDI'18)*. USENIX Association, USA, 445–460.

[8] Arijit Banerjee, Rajesh Mahindra, Karthik Sundaresan, Sneha Kasera, Kobus Van der Merwe, and Sampath Rangarajan. 2015. Scaling the LTE Control-Plane for Future Mobile Access. In *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies* (Heidelberg, Germany) *(CoNEXT '15)*. Association for Computing Machinery, New York, NY, USA, Article 19, 13 pages. https://doi.org/10.1145/2716281.2836104

[9] Carsten Binnig, Andrew Crotty, Alex Galakatos, Tim Kraska, and Erfan Zamanian. 2016. The End of Slow Networks: It's Time for a Redesign. *Proc. VLDB Endow.* 9, 7 (March 2016), 528–539. https://doi.org/10.14778/2904483.2904485

[10] Michael J Cahill, Uwe Röhm, and Alan D Fekete. 2009. Serializable isolation for snapshot databases. *ACM Transactions on Database Systems (TODS)* 34, 4 (2009), 1–42.

[11] Qingchao Cai, Wentian Guo, Hao Zhang, Divyakant Agrawal, Gang Chen, Beng Chin Ooi, Kian-Lee Tan, Yong Meng Teo, and Sheng Wang. 2018. Efficient Distributed Memory Management with RDMA and Caching. *Proc. VLDB Endow.* 11, 11 (July 2018), 1604–1617. https://doi.org/10.14778/3236187.3236209

[12] Francesco Calabrese, Mi Diao, Giusy Lorenzo, Joseph Ferreira, and Carlo Ratti. 2013. Understanding individual mobility patterns from urban sensing data: A mobile phone trace example. *Transportation Research Part C: Emerging Technologies* 26 (01 2013), 301–313. https://doi.org/10.1016/j.trc.2012.09.009

[13] John B. Carter, John K. Bennett, and Willy Zwaenepoel. 1991. Implementation and Performance of Munin. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles* (Pacific Grove, California, USA) *(SOSP '91)*. Association for Computing Machinery, New York, NY, USA, 152–164. https://doi.org/10.1145/121132.121159

[14] Aleksey Charapko, Ailidani Ailijiang, and Murat Demirbas. 2018. Adapting to access locality via live data migration in globally distributed datastores. In *2018 IEEE International Conference on Big Data (Big Data)*. IEEE, 3321–3330.

[15] Yanzhe Chen, Xingda Wei, Jiaxin Shi, Rong Chen, and Haibo Chen. 2016. Fast and General Distributed Transactions Using RDMA and HTM. In *Proceedings of the Eleventh European Conference on Computer Systems* (London, United Kingdom) *(EuroSys '16)*. Association for Computing Machinery, New York, NY, USA, Article 26, 17 pages. https://doi.org/10.1145/2901318.2901349

[16] Natacha Crooks, Matthew Burke, Ethan Cecchetti, Sitar Harel, Rachit Agarwal, and Lorenzo Alvisi. 2018. Obladi: Oblivious Serializable Transactions in the Cloud. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation* (Carlsbad, CA, USA) *(OSDI'18)*. USENIX Association, USA, 727–743.

[17] Carlo Curino, Evan Jones, Yang Zhang, and Sam Madden. 2010. Schism: A Workload-Driven Approach to Database Replication and Partitioning. *Proc. VLDB Endow.* 3, 1–2 (Sept. 2010), 48–57. https://doi.org/10.14778/1920841.1920853

[18] Dave Dice, Ori Shalev, and Nir Shavit. 2006. Transactional Locking II *(DISC'06)*. Springer-Verlag, Berlin, Heidelberg, 194–208. https://doi.org/10.1007/11864219_14

[19] Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudre-Mauroux. 2013. OLTP-Bench: An Extensible Testbed for Benchmarking Relational Databases. *Proc. VLDB Endow.* 7, 4 (Dec. 2013), 277–288. https://doi.org/10.14778/2732240.2732246

[20] Aleksandar Dragojević, Dushyanth Narayanan, Orion Hodson, and Miguel Castro. 2014. FaRM: Fast Remote Memory. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation* (Seattle, WA) *(NSDI'14)*. USENIX Association, USA, 401–414.

[21] Aleksandar Dragojević, Dushyanth Narayanan, Edmund B. Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. 2015. No Compromises: Distributed Transactions with Consistency, Availability, and Performance. In *Proceedings of the 25th Symposium on Operating Systems Principles* (Monterey, California) *(SOSP '15)*. Association for Computing Machinery, New York, NY, USA, 54–70. https://doi.org/10.1145/2815400.2815425

[22] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. 1988. Consensus in the Presence of Partial Synchrony. *J. ACM* 35, 2 (April 1988), 288–323. https://doi.org/10.1145/42282.42283

[23] Aaron J. Elmore, Vaibhav Arora, Rebecca Taft, Andrew Pavlo, Divyakant Agrawal, and Amr El Abbadi. 2015. Squall: Fine-Grained Live Reconfiguration for Partitioned Main Memory Databases. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (Melbourne, Victoria, Australia) *(SIGMOD '15)*. Association for Computing Machinery, New York, NY, USA, 299–313. https://doi.org/10.1145/2723372.2723726

[24] Jose M. Faleiro, Daniel J. Abadi, and Joseph M. Hellerstein. 2017. High Performance Transactions via Early Write Visibility. *Proc. VLDB Endow.* 10, 5 (Jan. 2017), 613–624. https://doi.org/10.14778/3055540.3055553

[25] Hua Fan and Wojciech Golab. 2019. Ocean vista: gossip-based visibility control for speedy geo-distributed transactions. *Proceedings of the VLDB Endowment* 12, 11 (2019), 1471–1484.

[26] Rachid Guerraoui and Michal Kapalka. 2008. On the Correctness of Transactional Memory *(PPoPP '08)*. Association for Computing Machinery, New York, NY, USA, 175–184. https://doi.org/10.1145/1345206.1345233

[27] Rachael Harding, Dana Van Aken, Andrew Pavlo, and Michael Stonebraker. 2017. An Evaluation of Distributed Concurrency Control. *Proc. VLDB Endow.* 10, 5 (Jan. 2017), 553–564. https://doi.org/10.14778/3055540.3055548

[28] Tim Harris and Keir Fraser. 2014. Language Support for Lightweight Transactions. *SIGPLAN Not.* 49, 4S (July 2014), 64–78. https://doi.org/10.1145/2641638.2641654

[29] Maurice Herlihy and J. Eliot B. Moss. 1993. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture* (San Diego, California, USA) *(ISCA '93)*. Association for Computing Machinery, New York, NY, USA, 289–300. https://doi.org/10.1145/165123.165164

[30] L. Hoang Le, E. Fynn, M. Eslahi-Kelorazi, R. Soulé, and F. Pedone. 2019. DynaStar: Optimized Dynamic Partitioning for Scalable State Machine Replication. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*. 1453–1465.

[31] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. 2010. ZooKeeper: Wait-Free Coordination for Internet-Scale Systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference* (Boston, MA) *(USENIXATC'10)*. USENIX Association, USA, 11.

[32] iWireless. 2020. Macrocell vs Microcell. https://www.iwireless-solutions.com/macrcocell-vs-microcell/. (Accessed on 10/06/2020).

[33] Murad Kablan, Azzam Alsudais, Eric Keller, and Franck Le. 2017. Stateless Network Functions: Breaking the Tight Coupling of State and Processing. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. USENIX Association, Boston, MA, 97–112.

[34] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2016. FaSST: Fast, Scalable and Simple Distributed Transactions with Two-Sided (RDMA) Datagram RPCs. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (Savannah, GA, USA) *(OSDI'16)*. USENIX Association, USA, 185–201.

[35] Antonios Katsarakis, Vasilis Gavrielatos, M.R. Siavash Katebzadeh, Arpit Joshi, Aleksandar Dragojevic, Boris Grot, and Vijay Nagarajan. 2020. Hermes: A Fast, Fault-Tolerant and Linearizable Replication Protocol. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 201–217. http://hermes-protocol.com

[36] Junaid Khalid and Aditya Akella. 2019. Correctness and Performance for Stateful Chained Network Functions. In *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation* (Boston, MA, USA) *(NSDI'19)*. USENIX Association, USA, 501–515.

[37] Chinmay Kulkarni, Aniraj Kesavan, Tian Zhang, Robert Ricci, and Ryan Stutsman. 2017. Rocksteady: Fast Migration for Low-Latency In-Memory Storage. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China) *(SOSP '17)*. Association for Computing Machinery, New York, NY, USA, 390–405. https://doi.org/10.1145/3132747.3132784

[38] Collin Lee, Seo Jin Park, Ankita Kejriwal, Satoshi Matsushita, and John Ousterhout. 2015. Implementing Linearizability at Large Scale and Low Latency. In *Proceedings of the 25th Symposium on Operating Systems Principles* (Monterey, California) *(SOSP '15)*. 71–86.

[39] Juchang Lee, Kyu Hwan Kim, Hyejeong Lee, Mihnea Andrei, Seongyun Ko, Friedrich Keller, and Wook-Shin Han. 2020. Asymmetric-Partition Replication for Highly Scalable Distributed Transaction Processing in Practice. *Proc. VLDB Endow.* 13, 12 (Aug. 2020), 3112–3124. https://doi.org/10.14778/3415478.3415538

[40] Qian Lin, Pengfei Chang, Gang Chen, Beng Chin Ooi, Kian-Lee Tan, and Zhengkui Wang. 2016. Towards a Non-2PC Transaction Management in Distributed Database Systems. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 1659–1674. https://doi.org/10.1145/2882903.2882923

[41] Yi Lu, Xiangyao Yu, Lei Cao, and Samuel Madden. 2020. Aria: A Fast and Practical Deterministic OLTP Database. *Proc. VLDB Endow.* 13, 12 (July 2020), 2047–2060. https://doi.org/10.14778/3407790.3407808

[42] Yi Lu, Xiangyao Yu, Lei Cao, and Samuel Madden. 2021. Epoch-based Commit and Replication in Distributed OLTP Databases. *Proc. VLDB Endow.* 14 (2021), 743–756. https://doi.org/10.14778/3407790.3407808

[43] Yi Lu, Xiangyao Yu, and Samuel Madden. 2019. STAR: Scaling Transactions through Asymmetric Replication. *Proc. VLDB Endow.* 12, 11 (July 2019), 1316–1329. https://doi.org/10.14778/3342263.3342270

[44] Virendra Jayant Marathe and Mark Moir. 2008. Toward High Performance Nonblocking Software Transactional Memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Salt Lake City, UT, USA) *(PPoPP '08)*. Association for Computing Machinery, New York, NY, USA, 227–236. https://doi.org/10.1145/1345206.1345240

[45] Ali Mohammadkhan, KK Ramakrishnan, Ashok Sunder Rajan, and Christian Maciocco. 2016. Considerations for re-designing the cellular infrastructure exploiting software-based networks. In *2016 IEEE 24th International Conference on Network Protocols (ICNP)*. IEEE, 1–6.

[46] C. Mohan, B. Lindsay, and R. Obermarck. 1986. Transaction Management in the R* Distributed Database Management System. *ACM Trans. Database Syst.* 11, 4 (Dec. 1986), 378–396. https://doi.org/10.1145/7239.7266

[47] Shuai Mu, Yang Cui, Yang Zhang, Wyatt Lloyd, and Jinyang Li. 2014. Extracting More Concurrency from Distributed Transactions. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation* (Broomfield, CO) *(OSDI'14)*. USENIX Association, USA, 479–494.

[48] Alex Nazaruk and Michael Rauchman. 2013. Big Data in Capital Markets. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data* (New York, New York, USA) *(SIGMOD '13)*. Association for Computing Machinery, New York, NY, USA, 917–918. https://doi.org/10.1145/2463676.2486082

[49] Simo Neuvonen, Antoni Wolski, Markku Manner, and Vilho Raatikka. 2009. Telecom Application Transaction Processing Benchmark. http://tatpbenchmark.sourceforge.netl.

[50] Nginx. 2021. High-Performance Load Balancing. https://www.nginx.com/products/nginx/load-balancing/. (Accessed on 16/03/2021).

[51] Binh Nguyen, Tian Zhang, Bozidar Radunovic, Ryan Stutsman, Thomas Karagiannis, Jakub Kocur, and Jacobus Van der Merwe. 2018. ECHO: A Reliable Distributed Cellular Core Network for Hyperscale Public Clouds. In *Proceedings of the 24th Annual International Conference on Mobile Computing and Networking* (New Delhi, India) *(MobiCom '18)*. ACM, New York, NY, USA, 163–178. https://doi.org/10.1145/3241539.3241564

[52] John Ousterhout, Arjun Gopalan, Ashish Gupta, Ankita Kejriwal, Collin Lee, Behnam Montazeri, Diego Ongaro, Seo Jin Park, Henry Qin, Mendel Rosenblum, et al. 2015. The RAMCloud storage system. *ACM Transactions on Computer Systems (TOCS)* 33, 3 (2015), 1–55.

[53] PhantomNet. 2021. OpenEPC Tutorial. https://wiki.emulab.net/wiki/phantomnet/oepc-protected/openepc-tutorial. (Accessed on 16/03/2021).

[54] Iraklis Psaroudakis, Tobias Scheuer, Norman May, Abdelkader Sellami, and Anastasia Ailamaki. 2016. Adaptive NUMA-Aware Data Placement and Task Scheduling for Analytical Workloads in Main-Memory Column-Stores. 10, 2 (Oct. 2016), 37–48. https://doi.org/10.14778/3015274.3015275

[55] A. S. Rajan, S. Gobriel, C. Maciocco, K. B. Ramia, S. Kapury, A. Singhy, J. Ermanz, V. Gopalakrishnanz, and R. Janaz. 2015. Understanding the bottlenecks in virtualizing cellular core network functions. In *The 21st IEEE International Workshop on Local and Metropolitan Area Networks*. 1–6. https://doi.org/10.1109/LANMAN.2015.7114735

[56] Redis. 2020. Redis. https://redis.io.

[57] Kun Ren, Dennis Li, and Daniel J. Abadi. 2019. SLOG: Serializable, Low-Latency, Geo-Replicated Transactions. *Proc. VLDB Endow.* 12, 11 (July 2019), 1747–1761. https://doi.org/10.14778/3342263.3342647

[58] Kun Ren, Alexander Thomson, and Daniel J Abadi. 2014. An evaluation of the advantages and disadvantages of deterministic database systems. *Proceedings of the VLDB Endowment* 7, 10 (2014), 821–832.

[59] I. Rüngeler and M. Tüxen. 2015. Socket API for the SCTP User-land Implementation (usrsctp). https://github.com/sctplab/usrsctp. (Accessed on 16/03/2021).

[60] Dan Salmon. 2020. sa7mon/venmo-data. https://github.com/sa7mon/venmo-data original-date: 2019-06-12T18:02:28Z.

[61] Marco Serafini, Rebecca Taft, Aaron J Elmore, Andrew Pavlo, Ashraf Aboulnaga, and Michael Stonebraker. 2016. Clay: fine-grained adaptive partitioning for general database schemas. *Proceedings of the VLDB Endowment* 10, 4 (2016), 445–456.

[62] Ravi Sethi. 1982. Useless actions make a difference: Strict serializability of database updates. *Journal of the ACM (JACM)* 29, 2 (1982), 394–403.

[63] Justine Sherry, Peter Xiang Gao, Soumya Basu, Aurojit Panda, Arvind Krishnamurthy, Christian Maciocco, Maziar Manesh, João Martins, Sylvia Ratnasamy, Luigi Rizzo, and et al. 2015. Rollback-Recovery for Middleboxes. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication* (London, United Kingdom) *(SIGCOMM '15)*. Association for Computing Machinery, New York, NY, USA, 227–240. https://doi.org/10.1145/2785956.2787501

[64] Dale Skeen. 1981. Nonblocking Commit Protocols. In *Proceedings of the 1981 ACM SIGMOD International Conference on Management of Data* (Ann Arbor, Michigan) *(SIGMOD '81)*. Association for Computing Machinery, New York, NY, USA, 133–142. https://doi.org/10.1145/582318.582339

[65] Robert Stets, Sandhya Dwarkadas, Nikolaos Hardavellas, Galen Hunt, Leonidas Kontothanassis, Srinivasan Parthasarathy, and Michael Scott. 1997. Cashmere-2L: Software Coherent Shared Memory on a Clustered Remote-Write Network. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles* (Saint Malo, France) *(SOSP '97)*. Association for Computing Machinery, New York, NY, USA, 170–183. https://doi.org/10.1145/268998.266675

[66] Rebecca Taft, Essam Mansour, Marco Serafini, Jennie Duggan, Aaron J Elmore, Ashraf Aboulnaga, Andrew Pavlo, and Michael Stonebraker. 2014. E-store: Fine-grained elastic partitioning for distributed transaction processing systems. *Proceedings of the VLDB Endowment* 8, 3 (2014), 245–256.

[67] Rebecca Taft, Irfan Sharif, Andrei Matei, Nathan VanBenschoten, Jordan Lewis, Tobias Grieger, Kai Niemi, Andy Woods, Anne Birzin, Raphael Poss, Paul Bardea, Amruta Ranade, Ben Darnell, Bram Gruneir, Justin Jaffray, Lucy Zhang, and Peter Mattis. 2020. CockroachDB: The Resilient Geo-Distributed SQL Database. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 1493–1509. https://doi.org/10.1145/3318464.3386134

[68] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J. Abadi. 2012. Calvin: Fast Distributed Transactions for Partitioned Database Systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data* (Scottsdale, Arizona, USA) *(SIGMOD '12)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/2213836.2213838

[69] Clive Unger, Dhiraj Murthy, Amelia Acker, Ishank Arora, and Andy Chang. 2020. Examining the Evolution of Mobile Social Payments in Venmo. In *International Conference on Social Media and Society* (Toronto, ON, Canada) *(SMSociety'20)*. Association for Computing Machinery, New York, NY, USA, 101–110. https://doi.org/10.1145/3400806.3400819

[70] Xingda Wei, Zhiyuan Dong, Rong Chen, and Haibo Chen. 2018. Deconstructing RDMA-Enabled Distributed Transactions: Hybrid is Better. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation* (Carlsbad, CA, USA) *(OSDI'18)*. USENIX Association, USA, 233–251.

[71] Xingda Wei, Jiaxin Shi, Yanzhe Chen, Rong Chen, and Haibo Chen. 2015. Fast In-Memory Transaction Processing Using RDMA and HTM. In *Proceedings of the 25th Symposium on Operating Systems Principles* (Monterey, California) *(SOSP '15)*. Association for Computing Machinery, New York, NY, USA, 87–104. https://doi.org/10.1145/2815400.2815419

[72] Shinae Woo, Justine Sherry, Sangjin Han, Sue Moon, Sylvia Ratnasamy, and Scott Shenker. 2018. Elastic Scaling of Stateful Network Functions. In *Proceedings of the 15th USENIX Conference on Networked Systems Design and Implementation* (Renton, WA, USA) *(NSDI'18)*. USENIX Association, USA, 299–312.

[73] Xiangyao Yu, Yu Xia, Andrew Pavlo, Daniel Sanchez, Larry Rudolph, and Srinivas Devadas. 2018. Sundial: Harmonizing Concurrency Control and Caching in a Distributed OLTP Database Management System. *Proc. VLDB Endow.* 11, 10 (June 2018), 1289–1302. https://doi.org/10.14778/3231751.3231763

[74] Irene Zhang, Naveen Kr Sharma, Adriana Szekeres, Arvind Krishnamurthy, and Dan RK Ports. 2018. Building consistent transactions with inconsistent replication. *ACM Transactions on Computer Systems (TOCS)* 35, 4 (2018), 1–37.

[75] Xinyi Zhang, Shiliang Tang, Yun Zhao, Gang Wang, Haitao Zheng, and Ben Y. Zhao. 2017. Cold Hard E-Cash: Friends and Vendors in the Venmo Digital Payments System.. In *ICWSM*. AAAI Press, 387–396.