Rethinking Database High Availability with RDMA Networks

Erfan Zamanian[1], Xiangyao Yu[2], Michael Stonebraker[2], Tim Kraska[2]
[1] Brown University [2] Massachusetts Institute of Technology
erfanz@cs.brown.edu, {yxy, stonebraker, kraska}@csail.mit.edu

ABSTRACT

Highly available database systems rely on data replication to tolerate machine failures. Both classes of existing replication algorithms, active-passive and active-active, were designed in a time when network was the dominant performance bottleneck. In essence, these techniques aim to minimize network communication between replicas at the cost of incurring more processing redundancy; a trade-off that suitably fitted the conventional wisdom of distributed database design. However, the emergence of next-generation networks with high throughput and low latency calls for revisiting these assumptions.

In this paper, we first make the case that in modern RDMA-enabled networks, the bottleneck has shifted to CPUs, and therefore the existing network-optimized replication techniques are no longer optimal. We present *Active-Memory Replication*, a new high availability scheme that efficiently leverages RDMA to completely eliminate the processing redundancy in replication. Using Active-Memory, all replicas dedicate their processing power to executing new transactions, as opposed to performing redundant computation. Active-Memory maintains high availability and correctness in the presence of failures through an efficient RDMA-based undo-logging scheme. Our evaluation against active-passive and active-active schemes shows that Active-Memory is up to a factor of 2 faster than the second-best protocol on RDMA-based networks.

PVLDB Reference Format:
Erfan Zamanian, Xiangyao Yu, Michael Stonebraker, Tim Kraska. Rethinking Database High Availability with RDMA Networks. *PVLDB*, 12(11): 1637-1650, 2019.
DOI: https://doi.org/10.14778/3342263.3342639

1. INTRODUCTION

A key requirement of essentially any transactional database system is *high availability*. A single machine failure should neither render the database service unavailable nor should it cause any data loss. High availability is typically achieved through distributed data replication, where each database record resides in a primary replica as well as one or multiple backup replicas. Updates to the primary copy propagate to all the backup copies synchronously such that any failed primary server can be replaced by a backup server.

The conventional wisdom of distributed system design is that the network is a severe performance bottleneck. Messaging over a conventional 10-Gigabit Ethernet within the same data center, for example, delivers 2–3 orders of magnitude higher latency and lower bandwidth compared to accessing the local main memory of a server [3]. Two dominant high availability approaches, *active-passive* and *active-active*, both adopt the optimization goal of minimizing network overhead.

With the rise of the next-generation networks, however, conventional high availability protocol designs are not appropriate anymore, especially in a setting of Local Area Network (LAN). The latest remote direct memory access (RDMA) based networks, for example, achieve a bandwidth similar to that of main memory, while having only a factor of $10\times$ higher latency. Our investigation of both active-passive and active-active schemes demonstrates that with a modern RDMA network, *the performance bottleneck has shifted from the network to CPU's computation overhead.* Therefore, the conventional network-optimized schemes are not the best fit anymore. This calls for a new protocol design to fully unleash the potential of RDMA networks.

To this end, we propose *Active-Memory Replication*, a new high availability protocol designed specifically for the next-generation RDMA networks in the LAN setting. The optimization goal in Active-Memory is to minimize the CPU overhead of performing data replication rather than minimizing network traffic. The core idea of Active-Memory is to use the one-sided feature of RDMA to directly update the records on remote backup servers without involving the remote CPUs. One key challenge in such design is to achieve fault tolerance when the CPUs on backup servers do not participate in the replication protocol. To address this problem, we designed a novel *undo-logging based replication protocol* where all the logic is performed unilaterally by the primary server. Each transaction goes through two serial phases: (1) undo logging and in-place updates and (2) log space reclamation, where each update is performed by a separate RDMA write. We have proved that the protocol has correct behavior under different failure scenarios.

We compared Active-Memory with both active-passive (i.e., log-shipping [26, 17]) and active-active (i.e., H-Store/VoltDB [16, 35] and Calvin [37]) schemes on various workloads and system configurations. Evaluation shows that Active-Memory is up to a factor of $2\times$ faster than the second-best baseline protocol that we evaluated over RDMA-based networks.

Specifically, the paper makes the following contributions:

- We revisit the conventional high availability protocols on the next-generation networks and demonstrate that optimizing for network is no longer the most appropriate design goal.

Proceedings of the VLDB Endowment, Vol. 12, No. 11
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3342263.3342639

- We propose Active-Memory, a new replication protocol for RDMA-enabled high bandwidth networks, which is equipped with a novel undo-log based fault tolerance protocol that is both correct and fast.
- We perform extensive evaluation of Active-Memory over conventional protocols and show it can perform 2× faster than the second-best protocol that we evaluated.

The rest of the paper is organized as follows: Section 2 describes the background of the conventional high availability protocols. Section 3 analyzes why conventional wisdom is no longer appropriate for modern RDMA-based networks. Section 4 describes the Active-Memory replication protocol in detail, and Section 5 demonstrates that the protocol is fault tolerant. In Section 6, we present the results of our performance evaluation. Section 7 reviews the related work and Section 8 concludes the paper.

2. HIGH AVAILABILITY IN DBMSS

Database systems experience failures for different reasons: hardware failures, network communication failures, software bugs, human errors, among others. Highly available database systems ensure that even in the face of such failures, the system remains operational with close to zero downtime.

High availability is typically achieved through replication: every record of the database gets replicated to one or more machines. To survive k machine failures, the system must make sure that for each transaction, its effects are replicated on at least $k+1$ machines. This is known as the k-safety rule. For example, for $k = 1$, each record is stored on two different machines, so that a failure of either of them does not disrupt the continuous operation of the system.

According to the widely-cited taxonomy of Gray el al. [10], replication protocols can be either *eager* or *lazy* (which pertains to *when* the updates are propagated to the replicas), and be either *primary copy* or *update anywhere* (which concerns *where* data-modifying transactions must be issued). Lazy replication is often used in data stores where strong consistency is not crucial, and the possibility of data loss can be accepted in exchange for possibly better throughput, such as in Amazon Dynamo [7] and Facebook Cassandra [20]. However, it has been acknowledged that abandoning consistency by lazy replication introduces complexity, overheads, and costs that offset its benefits for many applications [24].

In this paper, we target eager, or **strongly consistent** replication in shared-nothing architectures, which are suitable for databases which aim to offer high availability without compromising consistency and correctness. These databases make sure that all of the updates of a transaction reach the backup replicas before the transaction is considered committed [10]. Strong consistency makes it easy to handle machine failures, since all of the copies of a record are identical at all times. A failover can be as simple as informing every surviving server about the new change in the cluster and allowing them to reach an agreement on the current configuration. Reconfiguration of the cluster can be done by a cluster configuration manager such as Zookeeper [12].

Strongly consistent replication is implemented in databases using various schemes, which can be broadly categorized into two groups, namely *active-passive* and *active-active*, with each having their own many variations and flavours. Here, we abstract away from their subtleties and provide a simple and general overview of how each one delivers strongly consistent replication, and discuss their associated costs and limitations.

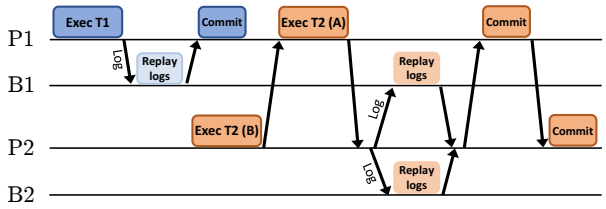


Figure 1: Active-passive replication using log shipping.

2.1 Active-Passive Replication

Active-passive replication is one of the most commonly used replication techniques. Each database partition consists of one active copy (known as the *primary* replica) which handles transactions and makes changes to the data, and one or more *backup* replicas, which keep their copies in sync with the primary replica. When a machine p fails, the system maintains its high availability by promoting one of p's backup nodes as the new primary and fails over to that node. There are many different implementations of active-passive replication both in academic projects [9, 15, 18, 39] and in commercial databases [4] (such as Postgres Replication [17], Oracle TimesTen [19], and Microsoft SQL Server Always On [26]). Active-passive schemes are often implemented through *log shipping* where the primary executes the transaction, then ships its log to all its backup replicas. The backup replicas replay the log so that the new changes are reflected in their copy.

Figure 1 illustrates how log shipping is used in an active-passive replication scheme. Here, we assume that the database is split into two partitions ($P1$ and $P2$), with each partition having a backup copy ($B1$ is backup for $P1$, and $B2$ is backup for $P2$). In practice, $P1$ and $B2$ may be co-located on the same machine, while $P2$ and $B1$ may reside on a second machine. $T1$ (colored in blue) is a single-partition transaction which touches data only on $P1$. Therefore, $P1$ executes this transaction, and before committing, it sends the change log to all its backup. Upon receiving all the acks, $P1$ can commit. Transactions that span multiple partitions, such as $T2$ (colored in orange), follow the same replication protocol, except that an agreement protocol such as 2PC is also needed to ensure consistency.

Wide adoption of active-passive replication is due to its simplicity and generality. However, two factors work against it. First, the log message contains each data record that the transaction has updated and is therefore may be big in size [33]. This becomes a serious bottleneck for conventional networks due to their limited bandwidth. Second, the communication between primaries and replicas not only imposes long delays to the critical path of transactions, but perhaps more importantly, it consumes processing power of machines for replaying logs and exchanging messages.

2.2 Active-Active Replication

The second group of eager replication techniques is update everywhere, or active-active protocols [10]. These systems allow any replica to accept a transaction and then broadcast the changes to all the other replicas. Synchronizing updates between replicas require much more coordination than active-passive, due to possible conflicts between each replica's transactions with the others. Main modern active-active databases solve this issue by removing coordination and replacing that with determinism of execution order among replicas [16, 34, 35, 37].

Deterministic active-active replication seeks to reduce the network communication needed to ship logs and coordinating with other replicas in an active-passive scheme. Specifically, the transactions are grouped into batches where each batch is executed by

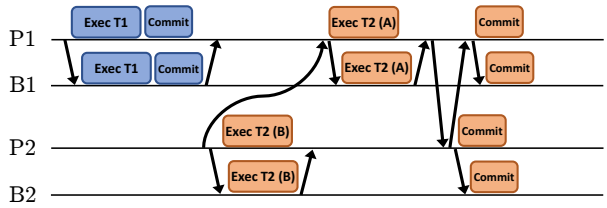


Figure 2: Active-active replication in H-store/VoltDB

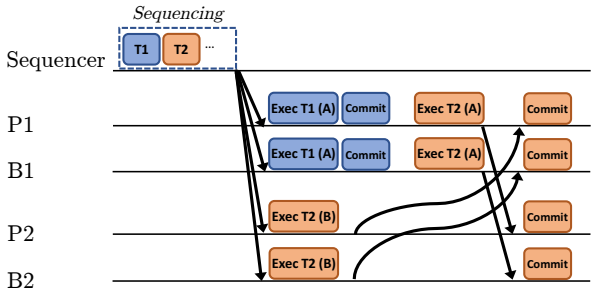


Figure 3: Active-active replication in Calvin

all replicas of the database in the same deterministic order, such that all the replicas end up with identical states once the batch is executed. Replicas in an active-active database only coordinate to determine the batches, but do not coordinate during the execution of transactions, which significantly reduces the network traffic between replicas. Two prominent examples of databases which use active-active replication are H-store [16] (and its commercial successor VoltDB [35]) and Calvin [37].

H-Store's replication protocol is illustrated in Figure 2. In H-Store, all transactions have to be registered in advance as stored procedures. The system is optimized to execute each transaction from the beginning to completion with minimum overhead. In H-Store, transactions do not get record locks, and instead only lock the partition they need. Transactions are executed in each partition sequentially, without getting pre-empted by other concurrent transactions. For a single-partition transaction such as $T1$, the primary replicates the ID of the invoked stored procedure along with its parameters to all its backup replicas. All replicas, including the primary, start executing the transaction code in parallel. Unlike in log shipping, here the replicas do not need to coordinate, as they execute the same sequence of transactions and make deterministic decisions (commit or abort) for each transaction. For multi-partition transactions, one of the primaries acts as the transaction coordinator and sends the stored procedure and its parameters to the other participating partitions. At each partition, an exclusive lock is acquired, so that no other transaction is allowed to be executed on that partition. Each partition sends the stored procedure to all its backup replicas so they run the same transaction and build their write-set. Finally, the coordinator initiates a 2PC to ensure that all the other primaries are able to commit. H-Store performs extremely well if the workload consists of mostly single-partition transactions. However, its performance quickly degrades in the presence of multi-partition transactions, since all participating partitions are blocked for the entire duration of such transactions.

Calvin [37] is another active-active system that takes a different approach than H-Store to enforce determinism in execution and replication (Figure 3). All transactions first arrive at the sequencer which orders all the incoming transactions in one single serial history. The inputs of the transactions are logged and replicated to all the replicas. Then, a single lock manager thread in each partition scans the serial history generated by the sequencer and acquires all the locks for each transaction, if possible. If the lock is already held, the transaction has to be queued for that lock. Therefore, Calvin requires that the read-set and write-set of transactions are known upfront, so that the lock manager would know what locks to get before even executing the transaction (This assumption may be too strict for a large category of workloads, where the set of records that a transaction is read or modified is known throughout executing the transaction). Those transactions which all their locks are acquired by the lock manager are then executed by the worker threads in each replica without any coordination between replicas. For multi-partition transactions, the participating partitions communicate their results to each other in a push-based manner (instead of pull-based, which is common in the other execution schemes).

Compared to Calvin with its sequencing overhead, H-store has a much lower overhead for single-partitioned transactions. Calvin, on the other hand, benefits from its global sequencing for multi-partition transactions.

3. THE CASE FOR REPLICATION WITH RDMA NETWORKS

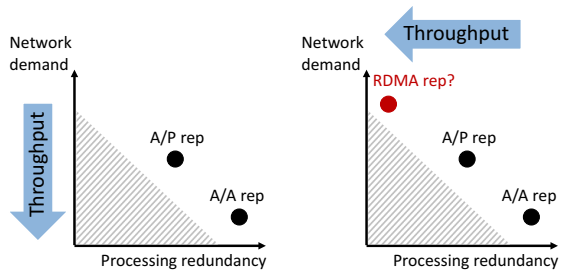
With the fast advancement of network technologies, conventional log-shipping and active-active schemes are no longer the best fits. In this section, we revisit the design trade-offs that conventional schemes made and demonstrate why the next-generation networks call for a new design of high availability protocol in Section 3.1. We then provide some background on RDMA in Section 3.2.

3.1 Bottleneck Analysis

The replication schemes described in the previous section were designed in a time that network communication was the obvious bottleneck in a distributed main-memory data store by a large margin. Reducing the need for accessing the network was therefore a common principle in designing efficient algorithms. Both classes of techniques approach this design principle by exchanging high network demand with more processing redundancy, each to a different degree. This idea is illustrated in Figure 4a. In log shipping, the logs have to be replayed at each replica, which may not be much cheaper than redoing the transaction itself for some workloads. Active-active techniques reduce the need for network communication even further and thus improve performance when the network is the bottleneck but impose even more redundancy for computation.

In these networks, communication during replication is considered expensive mainly due to three factors. (1) *Limited bandwidth* of these networks would be easily saturated and become the bottleneck. (2) *The message processing overhead* by the operating system proved to be substantial [3], especially in the context of many OLTP workloads which contain simple transactions that read and modify only a few records. (3) *High latency* of network communication increases the transaction latency, contributing to contention and therefore impacts throughput.

With the emergence of the next-generation of RDMA-enabled networks, such as InfiniBand, these assumptions need to be re-evaluated. (1) *Network bandwidth* has increased significantly, and its increase rate does not seem to be slowing down [13]. For example, a Mellanox ConnectX-4 EDR card offers 100× bandwidth of a typical 1Gb/sec Ethernet found in many public cluster offerings (including our own private cluster). (2) The *RDMA feature* open new



(a) Conventional TCP/IP networks

(b) Modern high bandwidth RDMA-enabled networks

Figure 4: The blue arrows indicate the throughput is increased by reducing which axis (i.e. reducing the network communication in conventional networks, and reducing processing redundancy in RDMA-enabled networks).

possibilities to design algorithms that eschew not only the message processing overhead of existing methods, but also the actual processing redundancy attached to replication (i.e. replaying logs in the log shipping scheme or executing transactions multiple times in deterministic schemes). (3) RDMA can achieve much *lower latency* compared to Ethernet networks, owing to its zero copy transfer and CPU bypass features. While the latency of RDMA is still an order of magnitude higher than main memory latency, it can be significantly masked by efficiently leveraging parallelism and concurrency in software [15].

Consequently, the modern RDMA-enabled networks have been shifting the bottleneck in the direction of CPU rather than the network, as depicted in Figure 4b; creating an identical memory copy can be done with very little CPU overhead. In this new environment, the old replication techniques are not optimal anymore, as their inherent processing redundancy underutilizes the processing power of the cluster, without efficiently leveraging modern networks. This calls for a new replication protocol which is designed specifically for the new generation of networks.

Note that in a shared-nothing environment, where nodes do not share memory or storage, nodes either have to replicate their new states to the replicas after performing one or a batch of transactions (resulting in relatively higher bandwidth requirement, as in the case with log shipping and Active-Memory) or they must enforce the same transaction schedule at all replicas (resulting in higher processing redundancy). This explains that shaded area in Figure 4 is very unlikely to achieve in a shared-nothing system.

3.2 Background for RDMA

The RDMA feature allows a machine to directly access the main memory of another machine without the involvement of the operating systems of either side, enabling *zero-copy* data transfer. RDMA recent popularity in the database community is mostly due to the fact that the network technologies which support RDMA have become cost competitive with Ethernet [3]. InfiniBand, RoCE, and iWarp are currently the main implementations of RDMA networks. Bypassing the OS and entirely offloading the networking protocol onto the network cards allow RDMA to have high throughput, low latency and low CPU utilizations. For example, the latest Mellanox ConnectX-6 RNICs can deliver 200Gb per second data transfer, have latency of below 1μs, and are able to handle up to 200 million messages per second.

The RDMA interface provides two operations types: one-sided (Read, Write, and atomic operations) and two-sided (Send and Receive). One-sided operations bypass the remote CPU and provide user-level memory access interface, where the remote memory is directly read from or written to. Two-sided operations, on the other hand, provide a message-passing interface for two user-level processes to exchange RPC messages. Unlike one-sided operations, two-sided communication involves the CPUs of both sides.

Two processes communicate to each other through *queue pairs*, which have different modes. In Reliable Connected (RC) queue pairs, packets are delivered in order and without any loss. These two properties are the key to our replication and fault-tolerance protocol, as we will see in Sections 4 and 5.

4. Active-Memory: RDMA-BASED REPLICATION

In this section, we start with an overview of Active-Memory Replication, our RDMA-based high availability solution, and then present the replication algorithm in detail.

4.1 Concurrency Control and Replication Assumptions

We present Active-Memory in the context of a partitioned and distributed shared-nothing database. Similar to many other data stores (such as FaRM [9], RAMCloud [30, 31], and FaSST [15]), we assume striped master partitions. Each data record has one primary copy on the *master node* and multiple replica copies on each of the *backup nodes*. Each node in the cluster is the master for a fraction of data records and the backup for some other records. A transaction accesses only the primary copy of a record. The backup copies are therefore only accessed and/or modified during the replication protocol. This is necessary to avoid having transactions reading uncommitted data on the backups.

While Active-Memory is orthogonal to the employed consistency protocol, in this paper we focus on two-phase locking (2PL) with a NO_WAIT policy [2]. However, we do require that every data structure change is made atomic within the boundaries of a transaction. Without this requirement, a direct memory copy with RDMA could replicate uncommitted changes. Our implementation guarantees this requirement using exclusive latches on the shared data structures (e.g., buckets in a hash-table). However, many alternative designs exist. Finally, we assume that each node is equipped with Non-Volatile Memory (NVM) similar to other recent high-performance OLTP systems [8, 15, 41].

4.2 Overview

Active-Memory adopts a primary-backup replication scheme that is specifically designed for RDMA-enabled networks. Instead of shipping logs to the backup nodes, the coordinator directly writes each of the transaction's changes to the main memory of the backup nodes using the *one-sided* write operations that RDMA supports. Therefore, the CPUs at the backup nodes are no longer involved in the data replication logic and therefore can be spent on processing new transactions. Specifically, for each transaction, the replication protocol involves two serial phases: (1) UNDO log transfer and in-place updates and (2) log space reclamation.

Overall, Active-Memory has the following salient features.

Strong Consistency: Following our discussion in Section 2, Active-Memory provides strong consistency replication. The data in the backup nodes always reflects the changes up to the last committed transaction and do not lag behind. Fresh backups enable fast and straight-forward fail-over.

Zero Processing Redundancy: In contrast to log shipping and active-active schemes, Active-Memory entirely eliminates the processing redundancy in the replication protocol. The transaction

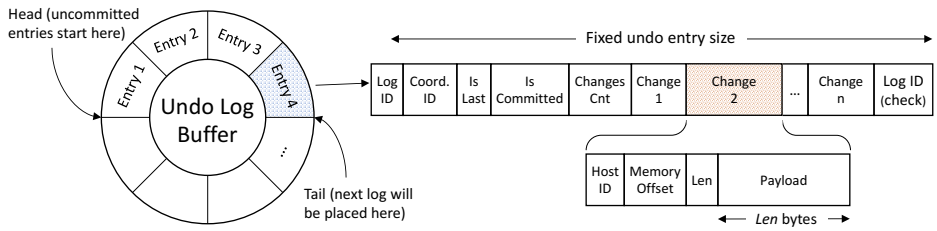


Figure 5: The structure of the log buffer and the log entries. Each node has a private log buffer on every other node.

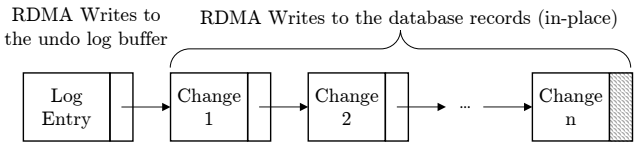


Figure 6: The linked list of RDMA messages sent to an active node

logic is executed only once. The backup nodes neither execute the transaction code nor replay the logs. Their CPU time is used to process new transactions.

Zero Overhead of Message Handling: By fully relying on one-sided RDMA operations, Active-Memory is able to remove much of the overhead caused by sending and receiving RPC messages, including the TCP/IP overhead, and message dispatching mechanism inside each database node.

Simple and Fast Fault-Tolerance: No matter how well a replication protocol works in the normal execution mode, recovery from failures must be reliable and consistent, and preferably fast. As we will see in Section 5, our protocol takes advantage of RDMA to provide an intuitive and fast mechanism for fault-tolerance.

The reduction of CPU time in Active-Memory comes at the cost of increased network traffic. As we will demonstrate in Section 6, however, this is not a problem with the new-generation RDMA networks due to their high bandwidth. By efficiently leveraging RDMA, Active-Memory can significantly outperform both log shipping and active-active schemes.

4.3 Design Challenges

Although the key idea behind Active-Memory is simple, what makes the design challenging is supporting fault tolerance and non-blocking fast recovery. For correctness, the coordinator must make sure that either all its changes are replicated on all the backup nodes, or none of them has been replicated. Different from a conventional log-shipping protocol where the primary and backups coordinate to achieve this goal, the backup nodes in Active-Memory do not actively participate in the replication protocol. Therefore, the coordinator has to unilaterally guarantee fault tolerance properties, which makes it more challenging.

To achieve this goal, Active-Memory relies on an undo logging mechanism, rather than traditional redo logging. The coordinator writes undo log entries to a backup node before directly updating any memory state on that node. Section 4.4 describes how undo logging occurs in Active-Memory in more details.

Another challenge in Active-Memory is non-blocking recovery, which requires the system to quickly recover to a consistent state when one or multiple coordinators or backup nodes fail. Active-Memory ensure that there is always enough information available to at least one of the surviving nodes so that it is able to recover the correct state of the modified data records and the ongoing transactions. Section 4.5 describes more details on this topic.

4.4 Undo Log Buffers

As stated before, Active-Memory uses an RDMA-based undo logging mechanism to ensure failure atomicity. Each server node has a pre-allocated RDMA buffer hosted on every other machine, which contains a certain number of fixed-size log entries, and implements a circular buffer primitive, as shown in Figure 5. Each log buffer can *only* be modified by a single remote server node. Thereby, there is no concurrent updates to a log buffer. This significantly simplifies the replication algorithm of Active-Memory (Section 4.5) and its fault-tolerance mechanism (Section 5).

Each node maintains the list of its available undo log entries on every other server machine. That is, a server knows what the head and tail pointers are for each of its remote log buffers. A log entry is placed in a remote buffer by issuing an RDMA Write operation. Implementing a circular buffer primitive means that the log entries which are not needed anymore can be re-used. In other words, the log buffer merely needs to have as many entries as the number of open (i.e. live, uncommitted) transactions initiated by each node at a time (for example in our system, this number is at most 20).

The layout of a log entry is illustrated on the right side of Figure 5. Each entry stores the pre-update contents of the attributes of the records modified by the transaction. For example, a transaction that modifies two attributes of three records will contain 6 changed attributes (`ChangesCnt` = 6), and for each of them, the entry will contain its `HostID`, the attribute's memory offset in that host, its size in bytes (`Len`), and its old content in `Payload`. Storing the values of changed attributes as opposed to the entire record content minimizes the required log size, and therefore minimizes the number of log entries to be sent per transaction. Each entry stores a locally unique identifier `LogID`. If a log exceeds the entry's fixed size, a follow-up log entry is sent with the same `LogID`. An entry with `IsLast` set to `false` signifies that a follow-up entry should be expected. The coordinator sets the same log ID at the end of each entry as well (`LogID_Check`). A log entry is considered correct and usable only if `LogID` and `LogID_Check` have the same value, otherwise it is considered corrupt. This is because at the receiver side, most NICs guarantee that RDMA Writes are performed in increasing address order [8] (i.e. writing of `LogID_Check` does not happen before writing of `LogID`). Therefore, this mechanism makes each log entry self-verifying. Finally, `IsCommitted` indicates the commit status of a transaction.

For some workloads, it may be possible that some few transactions have such large write-sets that they would not fit in the undo log buffer. Such an uncommon case can be supported by relying on the RPC-based log shipping; the coordinator sends an RPC message to each replica, with all its log entries concatenated to each other. The replicas apply the updates and send back acks to the coordinator. In general, the system must make sure that such cases remain rare for a given workload, and if not, it should increase the size the log buffers to better accommodate the write set of most transactions.

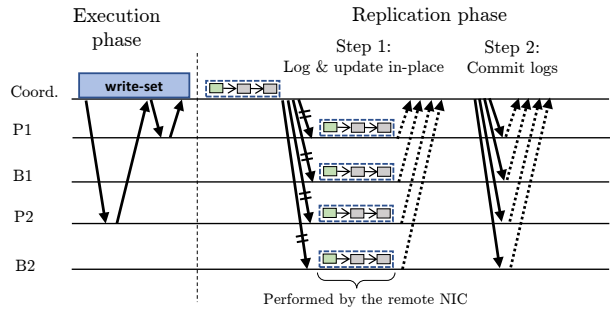


Figure 7: Active-Memory replication: Step 1 is replicating the undo logs and in-place updates to the backups (Section 4.5.2). Step 2 is marking the log entries as committed (Section 4.5.2).

4.5 Replication Algorithm

We now describe Active-Memory replication protocol. The coordinator initiates this protocol once it has built its read-set and write-set, which we refer to as the execution phase. Active-Memory assumes that for each transaction, its coordinator maintains a local write-set (`WS`), which contains a list of unique record keys and their new values which the transaction intends to write. At the end of its execution, the transaction is ready to commit and apply the updates in `WS` to the database. This is when the coordinator initiates the replication protocol which consists of two steps (Figure 7).

4.5.1 Step 1: Undo Log and In-place Update

The goal of the first step is to 1) replicate the undo logs, and 2) directly modify the records in the write-set in-place. These two substeps must be performed for all the involved partitions in the transaction and their replicas, henceforth referred to as *active nodes*. The algorithm must make sure that for each active node, in-place data update does not happen without undo logging.

Listing 1 shows the algorithm for this step. In summary, the coordinator scans its write-set and forms a linked list of RDMA operations for each active node. The first message of this linked list is the undo log RDMA Write operation, and the rest are the RDMA Writes for in-place attribute updates for records hosted on that partition, as shown in Figure 6. Because of the in-order message delivery guarantee of reliable connected queue pairs, the log message is received by the destination NIC *before* the in-place update messages. Such a guaranteed delivery order is crucial in our fault tolerance protocol, as we shall see later in Section 5.

As shown in Listing 1, the coordinator performs this process for each active node p in the transaction. It first retrieves and updates the tail pointer of the log buffer hosted on node p. Then, it initializes a log entry which is to be replicated later on p and its replicas (the `IsCommitted` field is set to `false`, indicating that the transaction has not yet committed). It retrieves the records in the write-set which are hosted on p (line 6), and adds their changed attributes to the log entry (lines 8 to 13). It then initializes a list of RDMA messages by adding the undo log entry (lines 16 to 18) and the data updates (lines 20 and 25). The resulting list is similar to Figure 6. This linked list is then issued to p and all its replicas (lines 27 to 30). Posting a linked list of RDMA operations in one call, as opposed to issuing them separately, allows the low-level driver to perform optimizations that result in less CPU usage on the sender side, and therefore improves performance [23, 1].

For the sake of brevity, here we assumed that all changed attributes can fit into one log message. Multi-entry log messages are handled in the same manner with one difference. `IsLast` in all the log entries except for the last one will be set to `false`.

```
1  for (p in active_nodes) {
     // update the tail pointer
3    tail = undoBuffers[p].tail++;
     log = init_log_entry();
5    // add the changes to the log msg
     recs = get_records_in_ws_in_partition(p);
7    i = 0;
     for (r in recs) {
9      for (attr in r.changedAttrs) {
         log.changes[i] = init_change(attr);
11       i++;
       }
13   }
     // prepare RDMA msgs
15   msgs = linkedList();
     ml = prep_log_RDMA(log);
17   // add the log entry as the *first* msg
     msgs.append(ml);
19   // add all updates to the msgs list
     for (r in recs) {
21     for (attr in r.changedAttrs) {
           md = prep_data_RDMA(attr);
23         msgs.append(md);
       }
25   }
     // send RDMA msg list to p and its replicas
27   issue_RDMA(p, msgs);
     for (rep in replicas[p]) {
29     issue_RDMA(rep, msgs);
     }
31 }
```

Listing 1: Step 1 of the replication protocol, i.e. undo logging and in-place data modification.

Once the log and change messages are sent, the transaction has to wait until their acks are received. These acks are indicators that the transaction log and its updates are replicated on all active nodes, so the coordinator proceeds to the second step.

4.5.2 Step 2: Commit the Log Entries

The goal of this step is to ensure that the commit decision will survive $f-1$ failures in a replica set of size f before reporting to the user (the k-safety property). To accomplish that, the coordinator first sets `IsCommitted` attribute of the log entry at each active node to `true` using an RDMA Write, as shown in Figure 7. Once the coordinator NIC finishes sending these RDMA Write messages, it locally increments the *head* pointer of the corresponding undo log buffer for each involved node, indicating that this entry can be re-used by future transactions. It also logs to its local NVM, releases its locks (if the transaction is multi-partition, informing every participant to release their locks) and returns to the user.

In summary, Active-Memory takes advantage of the in-order message delivery of reliable connected queue pairs in RDMA. Using this connection type, it is guaranteed that messages are delivered to the receiver's NIC in the same order that they are transmitted by the sender (even though that they may be applied to the receiver's main memory out of order [11], but this does not pose a problem for our protocol, as will be discussed in Section 5). The algorithm leverages this fact by issuing the undo log RDMA message *before* the in-place updates. This guarantees that even if the coordinator fails in the middle of the replication protocol, the in-place update RDMA messages will not take effect on the receiver without the undo log entry being present.

5. FAULT TOLERANCE

In this section, we discuss how our protocol guarantees fault tolerance in various failure scenarios without sacrificing either correctness or high availability. For the ease of explanation, we first describe our recovery mechanism for single-partition transactions,

Table 1: Different scenarios for the coordinator failure. Each row describes a scenario and how transaction state and records are recovered.

Time of Coordinator Failure	Coordinator Status	Transaction Status in T_P	Transaction Recovery	Data Recovery	Case ID
Before step 1	Uncommitted	No log	Abort	Data is untouched.	1
During step 1	Uncommitted	No log	Abort	Data is untouched.	2a
		Corrupt log	Abort	Data is untouched.	2b
		Logged	Abort	Data may be untouched, corrupt, or updated. Recover from log.	2c
After step 1 (received all acks) and before step 2	Uncommitted	Logged	Abort	Data is updated. Recover from log	3
During step 2	Uncommitted	Logged	Abort	Data is updated. Recover from log.	4a
		Commit-ready	Requires consensus: If all agree, commit. Otherwise, abort.	Data is updated. If the consensus is abort, recover from log. Otherwise, do nothing.	4b
After step 2 (received all acks)	Uncommitted	Commit-ready	Since all are committed, the consensus will be commit.	Data is up-to-date.	5a
	Committed	Commit-ready	Since all are committed, the consensus will be commit.	Data is up-to-date.	5b

and later extend it to multi-partition transactions. For single partition transactions, the coordinator is the primary replica for all records that the transaction accesses. We explain the failure of primary replicas and backup replicas separately. In the event of a machine failure, each surviving node S performs the procedure in Section 5.1 for transactions that S is a primary of (and therefore is the coordinator), and the procedure in Section 5.2 for the transactions that S is among the backups.

5.1 Recovery from Backup Failures

Handling failures of backup machines is fairly straightforward, since the backups do not perform any replication processing and the replication state is maintained by the coordinator, which is S. In our protocol, a coordinator returns the result to the client only when it has successfully replicated the updates on all its backups. Therefore, any backup failure which happens before S has successfully completed step 2 of the replication process (i.e. commit and release its log entries) will prevent S from returning to the client. Instead, it has to wait for the cluster manager to broadcast the new configuration, upon receiving which, S will know either it can commit or it needs to replicate on more machines. Note that the coordinator never rolls back a transaction for which it is already in the middle or at the end of step 2.

5.2 Recovery from Primary Failures

Recovery of transactions whose coordinator has failed is more challenging. The undo log messages are the footprints of the coordinator on the backups. Using these logs, the backups are able to decide to either rebuild and commit or discard and abort the transaction in order to maintain the system's transactional consistency. More specifically, after a machine P is suspected of failure, S will perform the following procedure.

1. S closes its RDMA queue pair to P, so that even if P returns from failure and issues new RDMA operations, it will not be successful once S starts the recovery protocol.

2. S checks the P-owned log buffer on S's local memory, and records all the transactions in the buffer, committed or uncommitted, to which we refer as T_P.

3. For each transaction t in T_P, S checks the data records pointed by the `Change` entries in t's undo logs.

4. S constructs a *status message* and broadcasts it to all surviving nodes. This status message contains P's and S's ID and also contains two entries per each transaction t in T_P: (t's id, t's status), where the first one is the unique ID of the transaction, and the second one is the current status of the transaction on S. t's status can be one of the following 4 cases (the 4th one will be described later):
i) *Corrupt log*: for at least one of the log entries of t, `LogID_check` does not match `LogID` (i.e. self-verifying check fails), or the log does not end with an entry with `IsLast=true` (i.e. not all logs have been received). In this case, the data records must be untouched, since the undo logs are delivered before updates.
ii) *Logged*: the log entries are correct, but the value of `CommitBit` is 0. In this case, the data may be untouched (i.e., the update messages are not received), corrupt (i.e., the update messages are partially received), or fully updated (i.e., the update message are fully received).
iii) *Commit-ready*: the value of `CommitBit` is 1. In this case, the undo logs must all be received and the data records are fully updated.

5. S broadcasts its status message to all surviving nodes, and receives their statue messages in return, which may have some overlap with T_P, but also may contain new transactions. These transactions have left no footprint on S, so S adds them to T_P and set their statuses to the fourth state: *No log*.

6. Upon receiving all the status messages, S commits a transaction in T_P if all involved nodes of that transaction are in *Commit-ready* status. Otherwise, S aborts the transaction, reverts its modified data using the corresponding undo log, and frees the P-owned log buffer on its local memory.

Once all backup nodes recovered the state and commit (or abort) the ongoing transactions, they inform the cluster manager to elect a new primary and the cluster continues regular processing.

Proof of Correctness — We now provide some intuition on how the recovery protocol ensures correctness. Table 1 summarizes the different failure scenarios that may befall the coordinator. We will refer to each scenario by its case ID which is in the rightmost column. The coordinator may fail *before* or *during* either of the two steps of the replication protocol presented in Section 4 (cases 1-4), or it may fail *after* completing the second step (case 5). In all these 5 cases, the recovery protocol guarantees correctness by satisfying these properties:

(1) Unanimous agreement: All involved nodes must reach the same agreement about the transaction outcome. This is achieved by steps 5 and 6 of the recovery protocol in Section 5.2. If there is at least one surviving node that has not gone through step 2 of the replication protocol, all nodes will unanimously abort the transaction. Otherwise, all nodes will reach the same decision.

(2) Consistency for aborted transactions: For an aborted transaction, the data in all involved nodes must be reverted to its prior consistent state, in the presence of any failure. In-order RDMA message delivery of reliable connected channels guarantees that for any surviving active node, if there is any data modification (complete or corrupt), there must be a complete log present at that node. Combined with the first property (unanimous agreement), this ensures that the recovery protocol always brings the database back to the last consistent state.

(3) Consistency for committed transactions: A transaction whose coordinator has failed can commit only in cases 4b (if all nodes agree), 5a, and 5b. What all these three cases have in common is that all active nodes have gone through step 2 of the replication protocol (even if the coordinator failed before receiving acks) and their logs are in *Commit-ready* status. Therefore, the data records on all the active nodes must be already successfully updated, as it is done in step 1 of the replication protocol.

(4) Consistency in responses to the client: The new elected coordinator will not report a different outcome to the client than what the original failed coordinator might have already reported. This property is guaranteed by the following reasoning: First, the coordinator commits the transaction only if it has completed step 2 on all active nodes. Second, in the event of the coordinator failure, the surviving nodes will reach a commit consensus only if the failed coordinator has completed step 2 on all active nodes. As a result, the transaction outcome is always preserved; If the coordinator assumed the transaction as committed, the consensus will be commit. Otherwise, it will be abort (cases 5a and 5b in Table 1).

5.3 Recovery of Multi-Partition Transactions

A multi-partition transaction accesses data from multiple primaries, with one partition acting as the coordinator. During the replication phase, the coordinator is in charge of both constructing the log entry for each accessed data partition, and in-place updating the data records in all nodes involved in the transaction — both the primary and backup nodes. A multi-partition transaction commits only after all the nodes have acknowledged the coordinator.

The recovery process of a multi-partition transaction is largely the same as a single-partition transaction. If the failed node is not the coordinator of an active transaction, the coordinator decides whether to replicate on more machines after the cluster reconfiguration, which is the same as in Section 5.1. If the the failed node is the coordinator, all the other machines locally construct and broadcast transaction status messages in the same way described in Section 5.2, with the only difference being that the commit decision is made if nodes from *all* involved partitions (rather than one partition) are in *Commit-ready* status.

6. EVALUATION

In this section, we evaluate Active-Memory and compare it to the three replication schemes that we introduced previously. In particular, we aim to explore the following two main questions:

1. How does the new-generations network change the design trade-off of high availability protocols?
2. how well does Active-Memory scale under different loads compared to the other active-passive and active-active replication protocols?

6.1 Experiment Settings

6.1.1 Setup

Our cluster consists of 8 machines connected to a single InfiniBand EDR 4X switch using a Mellanox ConnectX-4 card. Each machine is equipped with 256GB RAM and two Intel Xeon E5-2660 v2 processors, each with 10 cores. All processes were mapped to cores on only one socket which resides in the same NUMA region as the NIC. The machines run Ubuntu 14.04 Server Edition as their OS and Mellanox OFED 3.4-1 driver for the network.

6.1.2 Workload

For the experiments, we use YCSB (Yahoo Cloud Serving Benchmark [5]) which models the workload for large-scale online stores. It contains a single table, with each record containing a primary key and 10 string columns of size 100 bytes each. For all experiments, the YCSB table is hash partitioned by the primary key, where each physical machine contains 5 million records ($\sim$ 5 GB). Each YCSB transaction in our experiments consists of 10 operations. Unless otherwise stated, the set of records for each transaction are selected uniformly either from the entire database (for the distributed transactions) or from the local partition (for the single-node). Also, for most experiments, each operation in a transaction reads a record and modifies it. Also, the replication factor is set to 3 (i.e. 2-safety) for all experiments unless otherwise stated. We will inspect each of these settings in more detail.

6.1.3 Implementation

To achieve a fair comparison between the replication schemes, we implemented all of them in a unified platform.

The system consists of a number of server machines and one client machine. The database is partitioned horizontally into multiple partitions. Each machine is the primary for at least one partition and the backup for several other partitions, i.e. the striped master model. Furthermore, each machine contains the same number of primary and backup partitions.

Within each machine, there are multiple worker threads working on the same partitions, except for H-Store (in which each single thread owns one partition and has exclusive access to it). To extract maximum concurrency, each thread uses multiple co-routines, so that when one transaction is waiting for a network operation, the currently running co-routine yields to the next one, who will then work on a different transaction. This way, threads do not waste their time stalling on a network operation, and are always doing useful work. We found that using 5 co-routines per thread is sufficient to extract maximum concurrency in our implementation.

The implementations for all the replication protocols share the same storage layer and access methods. Besides, the RPC subsystem is implemented using RDMA Send and Receive. Therefore, all replication schemes use the same high-bandwidth RDMA-enabled network, and do not have the TCP/IP messaging overhead.

We now briefly describe our implementation for each of the replication protocols that we will compare in the following sub-sections.

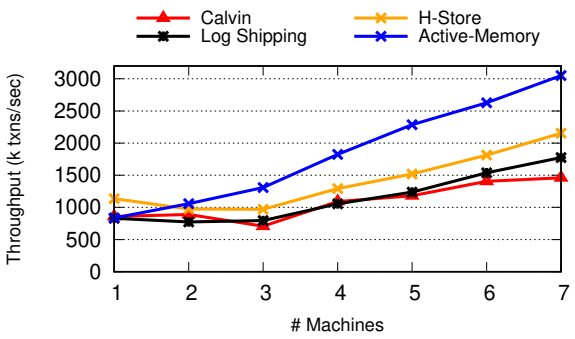


Figure 8: Scalability – The throughput of different replication protocols on different cluster sizes. All transactions are single-partition, with each reading and modifying 10 records.

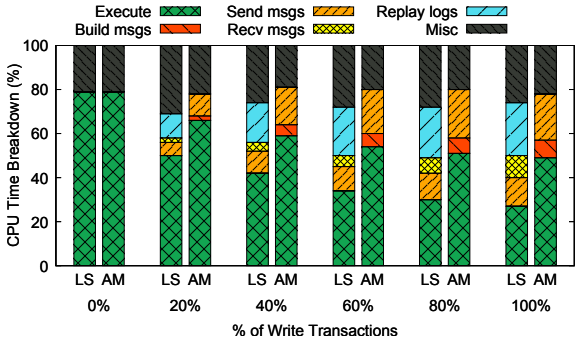


Figure 9: CPU time breakdown for log shipping and Active-Memory with varying percentage of write transactions. Cluster size is 5 with 2-safety. Total system utilization in all cases is 100%.

Active-Memory uses two-phase locking for concurrency control, and 2PC for multi-partitioned transactions. During transaction execution, remote records are accessed by issuing an RPC request to the record's primary partition. Once the transaction has successfully built its read-/write-set, the replication protocol described in Section 4 is performed using one-sided RDMA write operations.

Log shipping shares everything with Active-Memory, except its replication protocol which is based on sending log messages via RPC and replaying those logs at the receiver side. The transaction coordinator returns the result to the client only when it has received acks from all backup replicas. For distributed transactions, the coordinator sends the log message to all the backup replicas of the participating nodes.

H-Store replicates the transaction statement to all the replicas. For single-node transactions, each replica executes the transaction to completion without needing to acquire any locks, and returns the outcome to the coordinator. For distributed transactions, the coordinator locks all the participating nodes and engages them using 2PC. While one transaction is executing, no other transaction can be run in any of these partitions.

Calvin uses a partitioned and distributed sequencer, where each client sends its transactions to one of the them. At the beginning of every epoch, each sequencer replicates its transaction inputs on its replicas, and then sends it to all the schedulers on all the machines in its replica. The lock scheduler thread on each machine then scans all the sequenced transactions, and attempt to get all the locks for each transaction. Transactions that have acquired all of their locks are then processed by the worker threads. As stated earlier, any communication between two nodes is done via two-sided RDMA Send and Receive.

6.2 Single-Partition Transactions

We begin by examining the throughput of different replication schemes as the number of servers increases. The result is shown in Figure 8. In this experiment, all transactions are single-partition, and the replication factor is 3 (i.e. 2-safety). However, since in our platform the replicas are forced to be placed on different machines, replication factor of 3 is not possible when the number of machines is 1 or 2. For these two configurations, the replication is 0-safety (i.e. no replication) and 1-safety, respectively.

When the cluster size is 1, all schemes perform similarly since there is no replication involved, except for H-store, which achieves a higher throughput. This is because H-store eliminates locking overhead due to its serial transaction execution. With 2 machines (therefore 1-safety replication), the throughput of all schemes except Active-Memory drops, which is due to the overhead of replication in these protocols. In Calvin and H-Store, the same transaction has to be executed redundantly by all the replicas. For single-partition transactions, replaying the updates for 10 records in log shipping is not so much different than re-executing the transaction which is why its performance is very close to that of Calvin.

Table 2: The median latency per transaction.

H-Store	Calvin	Log shipping	Active-Memory
$85\mu s$	$1253\mu s$	$142\mu s$	$121\mu s$

The throughput of Active-Memory, in contrast, increases and further goes up with 3 machines. The reason is that replication in Active-Memory does not involve the CPU of the backup nodes at all. So, by increasing the cluster size, the processing power to handle new transactions proportionally increases. The asynchronous nature of RDMA operations means that the primary replica can issue the RDMA operations for a transaction, proceed to the next transaction, then come back later to pick up completions for the former transaction and commit it.

This experiment shows that even for a workload with single node transactions, which is the sweet spot of H-Store, its replication overhead dominates the benefit of its efficient single node transaction execution. Calvin not only has this overhead, but also has the cost of sequencing transactions and scheduling their locks, thereby achieving a lower throughput than H-Store. In general, Active-Memory can achieve 1.5x to 2x higher throughput compared to the other three protocols as the cluster scales out.

The average latency in the four replication protocols for a cluster size of 5 with 2-safety is reported in Table 2. Even though that Active-Memory involves two network roundtrips, it has lower latency compared to log shipping, which involves only one network roundtrip. This is because Active-Memory bypasses the RPC subsystem and also does not involve replaying logs at the replicas. Calvin has an order of magnitude higher latency due to the batching nature of its sequencing step. H-Store has the lowest latency among the others, owing to its simple command replication protocol.

To understand where the performance gain in Active-Memory comes from, Figure 9 illustrates the breakdown of CPU time for log shipping (LS) and Active-Memory (AM). Here, we fixed the number of machines to 5 with 2-safety, and varied the percentage of write transactions from zero (i.e. only read-only transactions) to 100% (the same workload in Figure 8). When there is no write transaction, both schemes perform similarly since no replication is involved, with 80% of the CPU time spent on performing transaction logic, and 20% on other tasks (including receiving

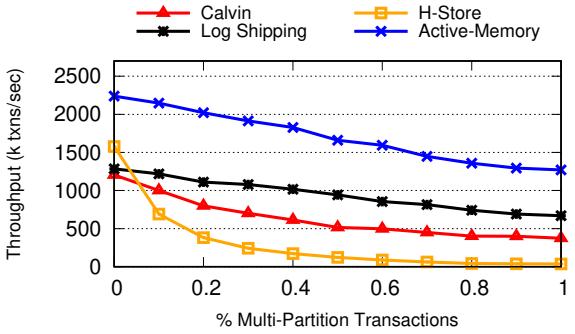


Figure 10: Multi-partition transactions – Eight records are read and modified locally, and two records are read and modified remotely. Cluster size is 5 with 2-safety.

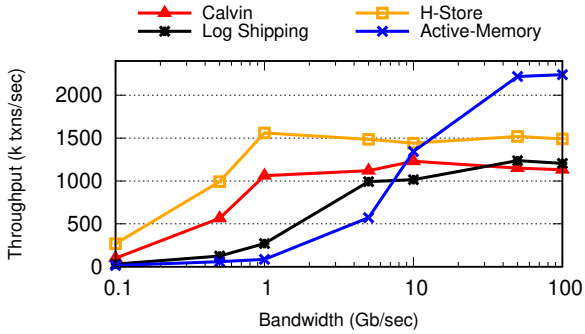


Figure 11: Network Bandwidth – The measured throughput of the replication schemes on 5 servers, with replication factor set to 2-safety, and 10 read-modify records per transaction.

client requests, copying each request to the corresponding worker's buffer, orchestrating the RPC messages sent to and received from the other servers, and polling acknowledgements). As the ratio of write transactions increases, however, receiving the log messages (in yellow) by the replicas and replaying them (in blue) incur significant overhead for log shipping; a cost that is non-existing in Active-Memory. Even though that relative to log shipping, Active-Memory spends more time on initializing the RDMA messages (in red) and sending them (in orange), all in all much less CPU time is left for performing the transaction logic in log shipping.

6.3 Multi-Partition Transactions

We next compare the impact of multi-partition transactions on the throughput of different replication protocols. The cluster size is set to 5, and the replication factor is 2-safety. While each single-partition transaction chooses all of its 10 records from the same partition, a multi-partition transaction selects 2 out of 10 of its records randomly from different partitions other than the rest of the 8 records (we experimented with different numbers of remote records per transaction, and observed similar results. Therefore their plots are omitted due to space constraints).

Figure 10 show the measured throughput of the four protocols with varying percentage of multi-partition transactions. H-Store performs better than log shipping and Calvin when there is no distributed transaction. However, with only 20% distributed transactions, the performance of H-Store drops to half of Calvin. This is because in H-Store, all participating nodes and their replicas are blocked for the entire duration of the transaction, which in this case takes one network roundtrip. The performance of all the other three replication protocols also drop with more multi-partition transactions. While the relative throughput of Active-Memory to that of log shipping remains the same with different percentage of distributed transactions, the relative throughput of Calvin to Active-Memory reaches from $\frac{1}{2}$ for single-partition transactions to $\frac{1}{3}$ for 100% distributed transactions.

6.4 Network Bandwidth

Our next experiment analyzes the impact of network bandwidth on throughput. We throttled the bandwidth by running background network data transfer jobs and made sure that during the course of the experiment, they consumed the requested portion of bandwidth.

Figure 11 shows the throughput of different protocols on 5 servers with 2-safety replication for single-partition transactions. When the available bandwidth is less than 10Gb/sec, the active-active schemes perform much better than the other two protocols. By enforcing a deterministic order of executing transactions at all replicas, these techniques can effectively minimize the network communication, which is key in delivering good performance in a network with very limited bandwidth. In such low bandwidth networks, Active-Memory is outperformed by log shipping as well by a factor of 2. To inspect this result in more depth, we also measured the network traffic per transaction in each of these replication schemes (Table 3). For a single-partition transaction, both H-Store and Calvin require relatively very little network traffic. Active-Memory, on the other hand, requires 15×, 9× and 1.8× more traffic than H-Store, Calvin and log shipping respectively. This explains the reason for this lower throughput compared to the other schemes when the network is slow: Active-Memory becomes network-bound when the bandwidth is limited. However, as the bandwidth exceeds 10Gb/sec, Active-Memory starts to outperform the other protocols. Increasing the available bandwidth does not result in any increase in the throughput of the other schemes, which is due to the important fact that these schemes are CPU-bound, thereby not exploiting high bandwidth of the modern networks. We performed the same experiment for the workload with multi-partition transactions (which we used for the experiment in Section 6.3), and observed similar results, except that H-Store was massively outperformed by Calvin, even when the bandwidth was limited. The measured network footprint for each multi-partition transaction is shown in the last column of Table 3. The plot for the distributed transactions is omitted due to space constraints.

Table 3: The network traffic per transaction in each replication protocol in our unified platform.

	Single-partition Transactions	**Multi-partition Transactions**
H-Store	∼ 0.3KB	∼ 1.5KB
Calvin	∼ 0.5KB	∼ 0.6KB
Log shipping	∼ 2.5KB	∼ 3.7KB
Active-Memory	∼ 4.5KB	∼ 6KB

In short, this experiment reveals that in the conventional Ethernet networks where the bandwidth was scarce, adding processing redundancy to avoid the network communication would be extremely beneficial. However, in the modern networks with their abundant bandwidth, the bottleneck has shifted in the direction of CPU; the replication protocols that do not leverage CPU-efficient RDMA operations and high bandwidth of these networks are not optimal anymore to provide LAN-based replication.

6.5 Replication Factor

The previous three experiments fixed the replication factor to 2-safety. In this experiment, we examine the effect of different

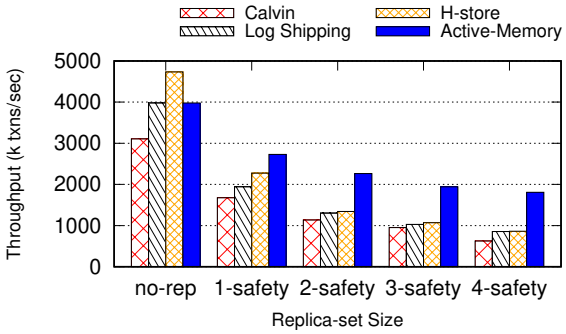


Figure 12: Replication Factor – Impact of replication factor.

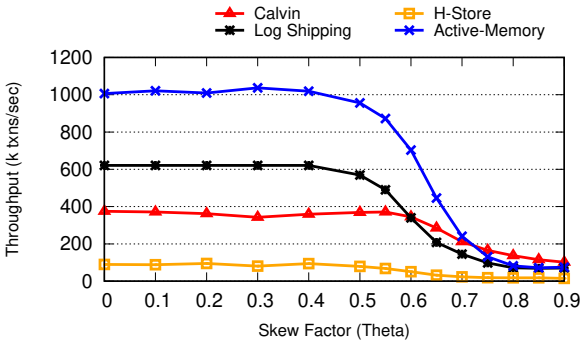


Figure 13: Contention – The throughput on 5 machines with varying skew factor. Records are chosen according to a Zipfian distribution from the entire cluster with different contention levels.

replication factors on throughput. Here, all transactions are single-partition, each reading and modifying 10 records.

The results in Figure 12 show the throughput of the four protocols as we varied the replication factor. With no replication (i.e. 0-safety), H-Store achieves the highest throughput, which is in agreement with the results of the scalability experiment. Active-Memory and log shipping exhibit the same performance as they use the same 2PL-based execution. The sequencing and scheduling overhead of Calvin accounts for its lower throughput.

Increasing the replication factor entails more redundant processing for H-Store and Calvin. For example, the throughput of H-Store for 1-safety replication is almost half of its throughput for no-replication, since each transaction is now executed twice. The same applies to Calvin. In addition, since the transaction modifies all 10 records that it reads, log replay in log shipping incurs almost the same cost as redoing the transaction. As the replication factor increases, the gap between Active-Memory and the other schemes widens since higher replication factors translate to more redundancy for them. On the other hand, a higher replication factor only increases the number of RDMA write operations that Active-Memory needs to send to its replicas, which does not impact the throughput proportionally, since modern NICs can handle up to tens or hundreds of million RDMA operations per second without putting much overhead to the CPU.

6.6 Impact of Contention

The goal of this experiment is to measure the impact of data skew on the throughput of different replication schemes. In YCSB, the skew is controlled using the Zipfian constant *theta*. When theta is 0, the records are uniformly selected from the entire cluster, and when it is 0.9, the accesses are highly skewed, resulting in a small set of contended records. Since each transactions touch 10 records,

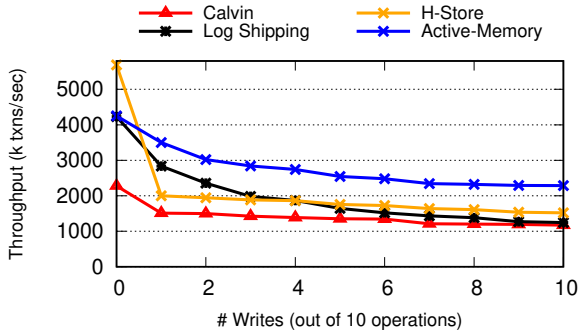


Figure 14: Read-Write Ratio – the throughput of different protocols on 5 machines with single-partition transaction, with varying number of write operations per transaction.

choosing a record randomly from the entire cluster makes all transactions multi-partition.

Figure 13 shows the measured throughput with different skew factors. For the values of theta up to 0.4, the performances of all the protocols remain unaffected. As we saw before in Section 6.3, H-Store performs poorly when the workload is dominated with distributed transactions, while Active-Memory maintains its relative better performance compared to the other replication schemes. As contention reaches 0.8, both Active-Memory and log shipping encounter high abort rates due to lock conflicts, and therefore their performances degrade significantly. The throughput of Calvin drops as well since the locks for each transaction have to be held during the network communication with the other servers. However, due to its deterministic lock scheduling and the fact that a transaction's working set is known prior to execution, Calvin can tolerate high contention better than all the other schemes. In particular, at high contention, the performance pf Calvin is up to 70% better than the second-best scheme, which is Active-Memory.

Note that Active-Memory is a replication protocol, and does not specify how concurrency should be managed. Currently, our implementation is based on 2PL for executing transactions. Moreover, in our protocol, a transaction is executed immediately upon arriving in the system with no sophisticated scheduling. By implementing Active-Memory on top of a different execution scheme than simple 2PL, which possibly employs a more sophisticated transaction re-ordering (such as [42] or [28]), one can expect that it will circumvent such cases and become more resilient to high data skew.

6.7 Read-Write Ratio

To analyze the effect of transactions with different read-write ratio on throughput, we varied the number of updates in transactions (out of 10 operations). Figure 14 shows the result for this experiment. H-Store has the highest throughput for read-only transactions (i.e. no write operations), as such a transaction is executed only in one node without the need to be replicated. Calvin, on the other hand, has the overhead of sequencing transactions and scheduling locks, even though that we applied the optimization that read-only transactions are only executed in the first replica, and not in the others. Log shipping and Active-Memory perform similarly for read-only transactions due to their identical execution scheme when there is no replication involved.

However, as transactions start to have more write operations, the log would contain more records and thus more work for the replicas. Also, even with 1 write operation, the optimizations to H-Store and Calvin is no longer possible, which explains their drop. With 4 write operations, log shipping delivers the same performance as

the deterministic approach of H-store, and with 10 writes, the same performance as Calvin. The throughput of both deterministic protocols, on the other hand, do not degrade significantly with more writes, as all operations are still handled locally without much extra redundancy compared to fewer writes. For Active-Memory, more writes in transactions indicate more undo log entries to replicate and more in-place updates to send, which explains the decrease in its throughput. However, it retains its better performance compared to the other schemes for all numbers of write count, and the more write there is, the wider the gap between Active-Memory's primary-backup replication and log shipping.

7. RELATED WORK

In this section, we discuss related works on high availability in conventional networks as well as RDMA-based OLTP systems.

7.1 Replication in Conventional Networks

In shared-nothing databases, the primary copy eager replication, also known as active-passive, is implemented through log shipping, and has been the most widely used technique to provide fault tolerance and high availability in strongly consistent databases [40] (such as Oracle TimesTen [19] and Microsoft SQL Server Always On [26]). In conventional networks, the coordination between replicas in log shipping, however, incurs significant cost, motivating much research effort in this area. Qin et al., identified two problems with using log shipping in high performance DBMSs [33]. First, logs can be so large in size that the network bandwidth becomes the bottleneck. second, sequentially replaying the log at the backups can make them constantly fall behind the primary replica. Eris [22] is a recent transactional system that proposes a network co-design which moves the task of ordering transactions from the replication protocol to the datacenter network, and eliminates the coordination overhead for a specific class of transactions, namely independent transactions (the same class that H-Store [16] and Granola [6] also optimize for). RAMCloud [30, 31] is a distributed key-value store that keeps all the data in the collective DRAM of the cluster, and only supports single-key operations. RAMCloud also takes a primary-backup approach to replication. However, unlike our work, RAMCloud does not assume non-volatile memory, and only keeps one copy of each object in memory. Instead, the durability is guaranteed by storing the changes to remote disks using a distributed log. Therefore, a failure of a node may suspend the operation of the system for the records on that node until the data is recovered from the backup disks.

RemusDB is another active-passive system that proposes to push replication functionality outside of the databases to the virtualization layer itself [25]. In general, even though that many of the optimizations for removing coordination between replicas in the aforementioned research projects indeed improve the performance in specific cases or environments, but the main problem, which is overhead of replaying logs, still persists.

For deterministic active-active replication, two well-known examples, namely H-Store [16] and Calvin [37], were studied in detail in Section 2.2. In short, they both rely on a deterministic execution and replication paradigm to ascertain that all replicas go through the same serial history of transactions. This way, replicas never diverge from each other without having to coordinate during execution. H-Store is mainly optimized for single-partition transactions, as it also eschews record locking due to its single-thread-per-partition serial execution. On the other hand, Calvin is more efficient for multi-partition transactions or workloads with high data contention, as it employs a novel distributed sequencer and deterministic locking scheme. Calvin requires that the read-set and write-set of transactions are known apriori, while H-Store requires to know all the transactions upfront.

As we already saw in Section 2, both active-passive and active-active introduce computation overhead, each in a different way. Active-passive involves exchanging multiple rounds of messages, and replaying the logs. Active-active results in processing the same transaction redundantly multiple times. Active-Memory uses RDMA Write to forgo both of these overheads.

7.2 RDMA-based OLTP Systems and Replication

RDMA, combined with the emerging NVM technologies as the main permanent storage, have been changing the landscape of distributed data store designs, especially in the area of distributed transaction processing. In addition to proposals to use RDMA for state machine replication through Paxos [38, 32], an array of key-value stores (e.g. [27, 14, 21]) and full transactional databases (e.g. [9, 41, 42, 15, 29]) have been proposed, each with a different way to leverage RDMA for executing transactions. Among these systems, some provide high availability using different variants of RDMA and implement active-passive replication [9, 36, 43, 15]. For example, FaRM [9] implements log shipping using one-sided RDMA Write operations to transfer logs to the non-volatile memory of the backups. To reduce transaction latency and enable group-replication, the coordinator in the FaRM protocol considers a transaction replicated once the RDMA Writes are sent to backups. Tailwind [36] is an RDMA-based log replication protocol built on top of RAMCloud, that proposes a mechanism to detect incomplete RDMA operations if a failure happens to the primary while it is replicating its log. In any case, no matter how the logs are shipped (using one-sided or two-sided RDMA), they have to be processed by the backups synchronously or asynchronously to provide high availability, which as we have already discussed in previous sections, imposes processing overhead on backups. Active-Memory is the first fault-tolerant replication scheme that fully leverages RDMA Write to consistently modify the state of the backup nodes in a failure atomic way.

8. CONCLUSIONS

In this paper, we presented Active-Memory, an RDMA-based mechanism to provide high availability using strongly consistent primary-backup replication. First, we identified that existing active-passive and active-active replication protocols were optimized for reducing network traffic at the cost of increased computation overhead. While such a design decision makes sense for conventional networks, it is no longer the best design choice for new-generation networks that offers orders of magnitude higher bandwidth and lower latency. Therefore, the main objective of Active-Memory is to minimize the CPU processing overhead by relying on the new features and properties of RDMA-enabled networks. Active-Memory achieves this goal by 1) using a novel RDMA-compatible undo logging mechanism, and 2) updating data records in the replicas directly using one-sided RDMA write operations. Evaluation shows that Active-Memory can achieve 2× performance improvement compared to the second-best protocol that we evaluated.

9. ACKNOWLEDGEMENT

This research is funded in part by the NSF CAREER Award IIS-1453171, Air Force YIP AWARD FA9550-15-1-0144, and supported by Google, Intel, and Microsoft as part of the MIT Data Systems and AI Lab (DSAIL).

10. REFERENCES

[1] D. Barak. Tips and tricks to optimize your rdma code. https://www.rdmamojo.com/2013/06/08/tips-and-tricks-to-optimize-your-rdma-code/, 2013. [Accessed: 2019-01-11].

[2] P. A. Bernstein and N. Goodman. Concurrency control in distributed database systems. *ACM Computing Surveys (CSUR)*, 13(2):185–221, 1981.

[3] C. Binnig, A. Crotty, A. Galakatos, T. Kraska, and E. Zamanian. The end of slow networks: it's time for a redesign. *PVLDB*, 9(7):528–539, 2016.

[4] E. Cecchet, G. Candea, and A. Ailamaki. Middleware-based database replication: the gaps between theory and practice. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 739–752. ACM, 2008.

[5] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154. ACM, 2010.

[6] J. A. Cowling and B. Liskov. Granola: Low-overhead distributed transaction coordination. In *USENIX Annual Technical Conference*, volume 12, 2012.

[7] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: amazon's highly available key-value store. In *ACM SIGOPS operating systems review*, volume 41, pages 205–220. ACM, 2007.

[8] A. Dragojević, D. Narayanan, O. Hodson, and M. Castro. Farm: fast remote memory. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, pages 401–414. USENIX Association, 2014.

[9] A. Dragojević, D. Narayanan, E. B. Nightingale, M. Renzelmann, A. Shamis, A. Badam, and M. Castro. No compromises: distributed transactions with consistency, availability, and performance. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 54–70. ACM, 2015.

[10] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The dangers of replication and a solution. *ACM SIGMOD Record*, 25(2):173–182, 1996.

[11] X. Hu, M. Ogleari, J. Zhao, S. Li, A. Basak, and Y. Xie. Persistence parallelism optimization: A holistic approach from memory bus to rdma network. In *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018.

[12] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *USENIX annual technical conference*, volume 8. Boston, MA, USA, 2010.

[13] InfiniBand Trade Association. Infiniband roadmap. https://www.infinibandta.org/infiniband-roadmap/. [Accessed: 2019-05-02].

[14] A. Kalia, M. Kaminsky, and D. G. Andersen. Using rdma efficiently for key-value services. *ACM SIGCOMM Computer Communication Review*, 44(4):295–306, 2015.

[15] A. Kalia, M. Kaminsky, and D. G. Andersen. Fasst: Fast, scalable and simple distributed transactions with two-sided (rdma) datagram rpcs. In *OSDI*, volume 16, pages 185–201, 2016.

[16] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. Zdonik, E. P. Jones, S. Madden, M. Stonebraker, Y. Zhang, et al. H-store: a high-performance, distributed main memory transaction processing system. *PVLDB*, 1(2):1496–1499, 2008.

[17] B. Kemme and G. Alonso. Database replication: a tale of research across communities. *PVLDB*, 3(1-2):5–12, 2010.

[18] J. Kim, K. Salem, K. Daudjee, A. Aboulnaga, and X. Pan. Database high availability using shadow systems. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, pages 209–221. ACM, 2015.

[19] T. Lahiri, M.-A. Neimat, and S. Folkman. Oracle timesten: An in-memory database for enterprise applications. *IEEE Data Eng. Bull.*, 36(2):6–13, 2013.

[20] A. Lakshman and P. Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.

[21] B. Li, Z. Ruan, W. Xiao, Y. Lu, Y. Xiong, A. Putnam, E. Chen, and L. Zhang. Kv-direct: High-performance in-memory key-value store with programmable nic. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 137–152. ACM, 2017.

[22] J. Li, E. Michael, and D. R. Ports. Eris: Coordination-free consistent transactions using in-network concurrency control. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 104–120. ACM, 2017.

[23] P. MacArthur and R. D. Russell. A performance study to guide rdma programming decisions. In *High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICESS), 2012 IEEE 14th International Conference on*, pages 778–785. IEEE, 2012.

[24] H. Mahmoud, F. Nawab, A. Pucher, D. Agrawal, and A. El Abbadi. Low-latency multi-datacenter databases using replicated commit. *PVLDB*, 6(9):661–672, 2013.

[25] U. F. Minhas, S. Rajagopalan, B. Cully, A. Aboulnaga, K. Salem, and A. Warfield. Remusdb: Transparent high availability for database systems. *The VLDB Journal*, 22(1):29–45, Feb. 2013.

[26] R. Mistry and S. Misner. *Introducing Microsoft SQL Server 2014*. Microsoft Press, 2014.

[27] C. Mitchell, Y. Geng, and J. Li. Using one-sided {RDMA} reads to build a fast, cpu-efficient key-value store. In *Presented as part of the 2013 {USENIX} Annual Technical Conference ({USENIX}{ATC} 13)*, pages 103–114, 2013.

[28] S. Mu, Y. Cui, Y. Zhang, W. Lloyd, and J. Li. Extracting more concurrency from distributed transactions. In *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*, pages 479–494, 2014.

[29] S. Novakovic, Y. Shan, A. Kolli, M. Cui, Y. Zhang, H. Eran, L. Liss, M. Wei, D. Tsafrir, and M. Aguilera. Storm: a fast transactional dataplane for remote data structures. *arXiv preprint arXiv:1902.02411*, 2019.

[30] D. Ongaro, S. M. Rumble, R. Stutsman, J. Ousterhout, and M. Rosenblum. Fast crash recovery in ramcloud. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 29–41. ACM, 2011.

[31] J. Ousterhout, A. Gopalan, A. Gupta, A. Kejriwal, C. Lee, B. Montazeri, D. Ongaro, S. J. Park, H. Qin, M. Rosenblum, et al. The ramcloud storage system. *ACM Transactions on Computer Systems (TOCS)*, 33(3):7, 2015.

[32] M. Poke and T. Hoefler. Dare: High-performance state machine replication on rdma networks. In *Proceedings of the*

24th International Symposium on High-Performance Parallel and Distributed Computing, pages 107–118. ACM, 2015.

[33] D. Qin, A. D. Brown, and A. Goel. Scalable replay-based replication for fast databases. *PVLDB*, 10(13):2025–2036, 2017.

[34] M. Stonebraker, S. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland. The end of an architectural era:(it's time for a complete rewrite). In *PVLDB*, pages 1150–1160, 2007.

[35] M. Stonebraker and A. Weisberg. The voltdb main memory dbms. *IEEE Data Eng. Bull.*, 36(2):21–27, 2013.

[36] Y. Taleb, R. Stutsman, G. Antoniu, and T. Cortes. Tailwind: fast and atomic rdma-based replication. In *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*, pages 851–863, 2018.

[37] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, and D. J. Abadi. Calvin: fast distributed transactions for partitioned database systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 1–12. ACM, 2012.

[38] C. Wang, J. Jiang, X. Chen, N. Yi, and H. Cui. Apus: Fast and scalable paxos on rdma. In *Proceedings of the 2017 Symposium on Cloud Computing*, pages 94–107. ACM, 2017.

[39] T. Wang, R. Johnson, and I. Pandis. Query fresh: Log shipping on steroids. *PVLDB*, 11(4):406–419, 2017.

[40] M. Wiesmann, F. Pedone, A. Schiper, B. Kemme, and G. Alonso. Understanding replication in databases and distributed systems. In *Proceedings 20th IEEE International Conference on Distributed Computing Systems*, pages 464–474. IEEE, 2000.

[41] E. Zamanian, C. Binnig, T. Harris, and T. Kraska. The end of a myth: Distributed transactions can scale. *PVLDB*, 10(6):685–696, 2017.

[42] E. Zamanian, J. Shun, C. Binnig, and T. Kraska. Chiller: Contention-centric transaction execution and data partitioning for fast networks. *arXiv preprint arXiv:1811.12204*, 2018.

[43] Y. Zhang, J. Yang, A. Memaripour, and S. Swanson. Mojim: A reliable and highly-available non-volatile memory system. In *ACM SIGARCH Computer Architecture News*, volume 43, pages 3–18. ACM, 2015.