Transactions for Distributed Actors in the Cloud

Tamer Eldeeb
Columbia University
tamer.eldeeb@columbia.edu

Philip A. Bernstein
Microsoft Research
philbe@microsoft.com

***Abstract*— Many cloud-service applications have a middle tier organized as micro-services or actors. Such applications have small objects that are spread over many servers and communicate via message passing. Transactions in such an application are necessarily distributed. However, distributed transactions usually perform poorly in this environment, primarily because locks must be held until after the forced-writes of two-phase commit, which are slow in cloud storage systems. We present a new transaction protocol that avoids this blocking by releasing all of a transaction's locks during phase one of two-phase commit, and by tracking commit dependencies to implement cascading abort. While a transaction T runs phase one, later conflicting transactions batch their updates. After T is prepared, the delayed batch can prepare, enabling a distributed form of group commit. We describe how to implement our protocol in an object-oriented runtime such as JVM or .NET. The performance measurements of our implementation in the Orleans actor framework show throughput up to 20x that of two-phase locking and two-phase commit.**

Keywords—database system, transaction, two-phase locking

I. Introduction

Many cloud services have a 3-tier architecture with a stateless front-end, a stateful middle-tier that implements business logic, and a storage layer. The stateful middle-tier is needed due to its heavy CPU and memory requirements, which make it uneconomical to embed as stored procedures in the storage layer. Today, the middle-tier is frequently organized as a set of micro-services. This is driven in part by the popularity of container technology like Docker [26], which is offered by most major cloud providers [5][29][32][39]. An application built as micro-services is composed of small, independent services that are versioned, deployed, upgraded and scaled separately. Services communicate via well-defined APIs (usually REST) and do not have access to shared data.

A similar model is actors, which are also popular in building middle-tier cloud applications, especially interactive ones such as games, social networks, Internet of Things, and telemetry [6][10]. Such applications are made of many actors, which are objects that do not share memory and interact via asynchronous messages. Actors are an extreme case of very small micro-services. In what follows, we use the more familiar, generic term "object" unless talking about a specific actor system.

It is often necessary to perform an operation that spans multiple objects with strong consistency and fault tolerance guarantees. Since objects are isolated from each other, multi-actor operations require a transaction mechanism [14]. Since cloud services are distributed across many servers for scalability and availability, most transactions that access multiple objects will access multiple servers. Such transactions must be distributed. The well-known challenges of scaling distributed transactions, e.g. in [31][45], has led many cloud systems to offer limited transaction support (e.g., within a shard or with weak isolation) or no support at all. This puts the burden on developers to use ad-hoc methods to obtain cross-object consistency, which is hard to do well.

To understand the scalability challenge, consider the most popular distributed transaction protocol, two-phase locking (2PL) for isolation with two-phase commit (2PC) for atomicity. To ensure isolation, a transaction holds locks until it finishes executing. Each object in its readset can release read-locks when it receives a prepare-request in phase-one of 2PC. However, each object in its writeset must hold its write locks until it receives a commit request in phase-two of 2PC. This technique, called **strict 2PL**, ensures that a transaction that aborts before phase-two can undo its writes easily, without cascading aborts.

Two-phase commit does two synchronous writes to storage before phase-two, one to prepare and one to commit. Thus, a data item participating in a transaction needs to be locked and inaccessible for two round-trips to storage. This greatly limits throughput on write-hot data, which is a bottleneck in most transaction applications. For example, a typical cloud storage system has high latency due to networking, disk, and replication overhead. In our runs, a write to cloud storage takes on average ~20 milliseconds (ms) within a single datacenter. Thus, a transaction holds locks for ~40 ms. This limits throughput to 25 transactions/second (tps) on write-hot data, even though distributed transaction execution typically takes a few milliseconds. Low-latency SSD-based cloud storage is faster [4][28], but it still incurs double-digit millisecond 2PC latencies, plus higher cost.

This problem of lock-holding time also applies to system that use optimistic concurrency control (OCC). Like strict 2PL, an OCC validator needs to set write locks on a transaction *T*'s writeset before validating *T* and hold those locks until *T* is committed, to ensure *T* can be aborted without cascading aborts.

In this paper we present a novel distributed transaction system that avoids these problems by allowing a transaction to release locks early, during phase-one of the 2PC protocol. After a transaction *T* finishes executing, it will not acquire more locks, so holding locks after this point has no value from a 2PL perspective. But if *T* releases write locks before it commits, subsequent transactions can read "dirty" data that will be invalid if *T* aborts. To avoid this inconsistency, a transaction keeps track of its dependencies on uncommitted transactions. A central validator service, the Transaction Manager, makes sure that *T* commits only after all of *T*'s dependent transactions commit. If one of *T*'s dependent transactions aborts, then *T* will abort too.

After T releases its write lock on object O, suppose another transaction T_1 updates O, terminates, and runs phase-one of 2PC. T_1 can unlock O, but must wait for T to finish writing O to storage before it can write O to storage. This handshake ensures that T_1's storage write is applied after T's storage write. Since O is unlocked, another transaction T_2 can update O, terminate, run phase-one of 2PC, and unlock O. Thus, while T is writing to storage, a sequence of later transactions can update O, terminate, start phase-one of 2PC, and unlock O. After T's storage write finishes, all of those later transactions can write their updates to O's storage as a batch, in one round-trip.

This batching of updates greatly increases transaction throughput on write-hot data. It is analogous to group commit, a popular technique for increasing transaction throughput in centralized systems. It also improves transaction latency. In a system that holds locks through 2PC, each transaction in a batch would have to wait for two synchronous writes by *every* transaction that precedes it in the batch, instead of just the one transaction that precedes the batch. Our performance measurements show throughput up to 20x that of strict 2PL/2PC, which we believe is sufficient for most real world applications.

We describe a general implementation architecture for our protocol in an object-oriented runtime such as JVM or .NET. It uses a centralized transaction manager to validate dependencies and to assign a timestamp to each transaction. Centralizing the transaction manager simplifies recovery. Timestamp assignment enables snapshot reads of multiversion data, to avoid conflicts between read-only queries and update transactions.

We also describe our implementation of that architecture in Orleans [10], a popular platform for building distributed applications using the actor model. In Orleans, actors are location transparent and can move between servers dynamically. This presents interesting challenges for keeping track of transaction execution and data access, challenges that apply to other actor platforms too. Finally, we present results and analysis of experiments measuring the performance of our implementation and comparing it to that of strict 2PL/2PC.

In summary, our contributions are as follows:

- We introduce a new optimization of 2PL/2PC for middle-tier applications that enables high transaction throughput despite high storage latency. It applies to cloud storage or any system where storage latency is much higher than transaction execution time.
- We describe a generic implementation architecture for the new mechanism (Sections II and III).
- We describe an implementation in a commercial programming framework, Orleans (Section IV). This is the first implementation we know of that supports cascading aborts.
- We present an experimental study that substantiates our performance claims (Section V).

Section VI covers related work. Section VII is the conclusion.

II. OVERVIEW

Conceptually there are two main entities in the system: **Application Servers** (or simply, **servers**) where transactions execute, and the Transaction Manager (TM), which is responsible for validating a transaction's dependencies (see Figure 1). A transaction executes as a set of communicating objects. Stateful objects persist their state in cloud storage. The TM has a log, which also is in cloud storage.

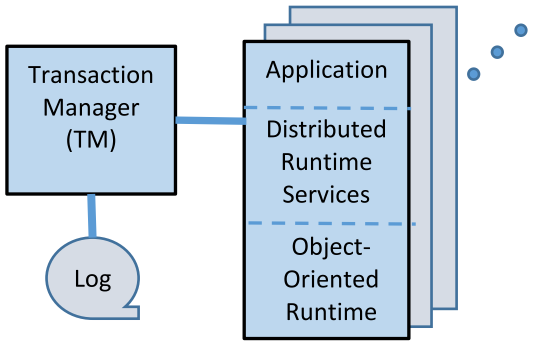


Figure 1 System Architecture

A transaction T starts by a client request to one of the servers to start a new transaction. That server acts as T's **coordinator**. As T executes, it acquires locks on objects it accesses at the coordinator and other servers, and it records all dependencies. After T finishes executing, the coordinator starts the prepare-phase of 2PC by asking each object O that T accessed to prepare.

When O receives a Prepare request, it releases T's lock. If T read but did not update O, then O replies "Prepared" to T's coordinator. Otherwise, O has to write T's update to storage. If O has another in-progress storage-write W, it waits for W to complete before persisting T's update (and any other updates to O while W was in-progress). After persisting T's update, O replies Prepared to T's coordinator. After receiving a reply of Prepared from all objects that T accessed, T's coordinator sends a description of T and T's dependencies to the TM for validation.

To process T's validation request, the TM waits until T's dependencies have committed. Then it assigns a (logical) timestamp to T, writes a commit record for T to its log, and notifies T's coordinator. Finally, T's coordinator replies to the client's request that T committed.

This design has several desirable characteristics that were mentioned in Section I. First, write locks are no longer held during the lengthy phases of 2PC that involve network round-trips and writes to storage. Second, since all commit decisions are made in one place (i.e., the TM), as long as the TM is alive it is easy for a server to learn the fate of a transaction, for example, when it times out waiting for a decision or while recovering from a failure. To cope with TM failure, a warm- or hot-standby TM can be used. Finally, the TM can provide a safe timestamp to use for reading consistent snapshots, which allows read-only transactions to execute without locks, without the need for validation, and without 2PC.

This design has three potential drawbacks: cascading aborts are possible, the centralized TM is a scalability bottleneck, and single-server transactions require validation. First, regarding cascading aborts, they will happen only due to server failures, e.g., a hardware or operating system failure, which are relatively rare. To see why, consider the other possible failures:

- When a transaction releases its locks, it has already finished executing, so transaction program failures and deadlocks are no longer possible.
- A controlled failure, such as a rolling upgrade, can be handled gracefully by waiting for all of a server's transactions to finish before restarting the server.
- If the TM fails, a watchdog detects that fact and recreates the TM, connecting it to the existing TM log. Since each transaction knows its dependencies, it can re-send its validation request to the new TM.

When a server *S* fails, there is significant delay in cluster reconfiguration, since other servers need to wait long enough to be sure that *S* failed and is not simply slow. Then they must work around *S'*s failure until *S* recovers. This typically takes a minute or so. Thus, independent of the existence of transactions, application execution will be disrupted. Cascading aborts will add to the period of unavailability, but the effect will be incremental, not a fundamentally new effect to be coped with.

Second, regarding the TM as a potential bottleneck, we observe that the TM is a simple RPC server that is not involved during most of the transaction execution. By carefully controlling communications to the TM, as we describe in Section III.C, our TM implementation can process hundreds of thousands of transactions per second; we substantiate that with measurements in Section V.D. Percolator [40], Tango [8], and other systems have RPC servers similar in purpose to ours that achieve very high throughput as well. Therefore, the TM will not be a bottleneck for the vast majority of applications.

The third issue is transactions that update just one object ordinarily do not need to execute 2PC. However, in our design, if a transaction *T* updates an object whose latest version is not yet committed, then the TM must validate *T* before *T* commits. This affects latency, but not throughput due to batching

Like any transaction system (2PL or not), our design works best if transaction execution time is short, since it reduces the conflict rate. Ideally, a transaction *T* accesses objects that are already in memory before *T* starts. This avoids *T* waiting for objects to be loaded from cloud storage while it holds locks. If the data set is too large to fit in memory, the developer can plan data accesses to achieve high throughput for hot objects. One way is to pre-fetch objects before executing the transaction. Another is to arrange object accesses so that hot objects are the last to be accessed. This is beneficial because it minimizes the execution time during which an access to the hot object causes a transactions conflict. Other techniques are in [14], Chapter 6.

III. RUNTIME ARCHITECTURE

Our proposed implementation architecture assumes that applications are written in an object-oriented language that uses an object-oriented runtime library, such as the .NET Framework or Java Virtual Machine, and executes in a cluster of servers. We further assume the existence of a set of runtime services that run on each server in the cluster (collectively referred to as "the runtime") that provide distributed systems functionality (e.g. directory services, RPC, and failure detection and recovery). To support transactions in this environment, we augment the runtime with the following abstractions: transactional object, transaction manager, transactional RPC, transaction context, and transaction agent.

A. *Transactional Object*

A ***transactional object*** is an object whose state is protected by ACID transactions. It has methods to read and write its state and to ***prepare***, ***commit*** and ***abort***, which are required for the 2PC protocol. In effect, a transactional object acts as a mini-database. It is the unit of access, meaning that a transaction that reads or writes any part of the object's state is considered to have read or written its entire state. Examples of a transactional object could be a micro-service in a micro-services platform, or an actor in an actor-based system.

A transactional object is ***multi-versioned***, meaning that it can hold many versions of its state. This enables it to undo updates due to cascading aborts and to support snapshot queries [9], which we use for read-only transactions. Each version has a version id, e.g., the identity of the transaction that wrote it.

We assume that transactions are implicitly bracketed by method tags, as in .NET's COM+ and J2EE. Each method, *M*, of an object can have one of the following tags:

RequiresNew – Every call to *M* starts a new transaction, *T*, and completes *T* on exit.

Required – If *M'*s caller is executing a transaction, *T*, then *M* becomes part of *T*. If not, then *M* starts a new transaction, *T'*, and completes *T'* on exit.

NotSupported – *M* never executes within a transaction, even if its caller is executing a transaction.

When *M* is invoked, the runtime uses *M's* tag to determine whether to start a transaction or propagate the caller's transaction to *M*. To start a transaction, the runtime sends a message to the ***Transaction Manager (TM),*** which dispenses a transaction id. If a method executing in object O_1 is running in transaction *T* and O_1 invokes a "Required" method in another object O_2, then O_2 should execute in *T*. This is called ***transactional RPC***. Methods can also be tagged as **ReadOnly**.

B. *Transaction Context*

Each RPC made by a transactional object carries a hidden parameter called the ***transaction context.*** The transaction context includes the identity of the caller's transaction, *T*, and of the objects and versions accessed by *T*. The runtime creates it when *T* starts. It contains:

- *T'*s id
- The ***identity*** and ***version*** of objects that *T* read (i.e., *T'*s ***readset***)
- The identity and version of the objects that *T* wrote (i.e., *T'*s ***writeset***)
- *T'*s ***dependencies*** (i.e. the transactions that must commit before *T* can commit)

The transaction context is passed back and forth between the object that started *T* and other objects *T* accesses. The transaction context supports a Union method, which accepts another transaction context with the same transaction id and unions its readset, writeset and dependencies with its own. After an RPC is completed, the callee returns an updated transaction context which the caller unions with its own.

C. Transaction Agent

The ***transaction agent*** (***TA***) is a service that runs on every server in the cluster. It provides interfaces to start a new transaction, commit a transaction, and query the status of a transaction. Transactional objects do not communicate with the TM directly, but use the TA to provide TM functionality.

To start a new transaction, *T*, the runtime calls the TA with a few parameters (e.g., timeout), which the TA forwards to the TM. The TM starts *T* and returns a transaction id.

To commit, *T* calls its TA, passing its transaction context, *C*. Then the TA runs our modified 2PC protocol. During the prepare-phase, the TA calls Prepare on each transactional object *O* participating in *T*, which it finds in *C*. Prepare unlocks *O* and, if *T* wrote *O*, persists *O'*s updated state. If all of *T'*s Prepare's succeed, the TA forwards *C* to the TM to validate *T'*s dependencies and commit *T*. As in standard 2PC, if the prepare-phase fails or the TM aborts *T*, the TA calls Abort on all participants. However, the TA does not call Commit on the participants if the TM replies with success. This is lazily done by the TM during checkpointing, which is described in Section III.G.

The TA controls all communications with the TM. This is critical for *system* scalability. The system is limited by the rate at which one server can process messages. To enable the system to scale to a higher transaction rate than the maximum message rate, the TA groups many TM requests into a smaller number of messages. It sends the Start and Commit calls that it receives to the TM in batches. At any point in time the TA has two TM messages outstanding: one for transaction start requests and one for transaction commit requests. All requests received after the message is sent are queued until a response is received and then are batched in the next message. This prevents the TAs from overwhelming the TM. It also allows the TM to limit the rate of messages it receives from TA's by increasing latencies as its load increases. The TA can apply throttling if its queues grow beyond a threshold.

D. Concurrency Control

We propose using object-granularity locking for concurrency control. When the runtime propagates a transaction, *T*, to a transactional object, *O*, it locks *O* on behalf of *T*. While *T* holds the lock, only method calls that are part of *T* may execute. Other calls are queued. Any deadlock avoidance scheme can be used. We propose using Wait-Die [42]. We assume the runtime guarantees that there is only one active instance of *O* at any point in time.

Since locks are not persisted, if a server hosting a transactional object *O* fails, a transaction *T* holding the lock on *O* will lose the lock and *T'*s in-flight updates of *O*. If *O* recovers before *T* starts 2PC, then *T* must avoid being fooled into committing and thereby breaking atomicity. The prepare-phase does this by checking whether *T* still holds its locks; if not, it aborts *T* (which might cause a cascading abort). It is also possible for *T* to lose a lock due to a failure and then re-acquire the lock by accessing *O* again before it finishes, perhaps via a different call-path. In this case, *T'*s context will refer to two versions of *O*. To handle this, if the union operation on *T'*s transaction context has seen more than one version of any object, it will abort *T*.

E. Managing Object State

As mentioned in Section III.A, a transactional object acts as a mini-database, with methods to read and write state and Prepare, Commit and Abort its transaction.

For each transactional object *O*, the runtime maintains two persistent logs: the ***Active versions log*** and ***Stable versions log***. The active versions log is an ordered list of writes by transactions that have started the prepare-phase but have not yet told *O* whether they have committed. Each *entry* is a pair <*S*, *T*> where *S* is the value of the state written by transaction *T*. The stable version log is an ordered list of the writes that are known to be committed. Each entry is a triple < *S*, *T*, LSN> where *S* and *T* are the same as before, and LSN is the log sequence number assigned by the TM to *T*. The stable version log may have more than one version to enable snapshot reads at an older timestamp.

Each transactional object, *O*, also has an in-memory ***working version*** that can be read or written by the transaction holding the lock on *O*. This working version eventually moves to the active versions log during the prepare-phase.

The basic methods of transactional objects are shown in Figure 2**.** To service a read on object *O* by transaction *T*, the runtime invokes Recover(), which asks the TA for the status of transactions in *O'*s active versions log and removes all that are known to have aborted. This avoids creating a dependency that is sure to cause *T* to abort. If this is *T'*s first access of *O*, the runtime creates a working version for *O*, which is a copy of the latest write, and returns that working version for every subsequent access. It also updates *T'*s readset and dependencies in its transaction context. If *T* is a read-only transaction, it does snapshot reads, which are described in Section III.I. The write procedure just overwrites the working version with a new one. To simplify the description, we assume a transaction does at most one write to an object. In reality a transaction can do many writes, and the transaction context keeps track of the write count.

As a mini-database, a transactional object also participates in 2PC. Recall that the coordinator TA calls Prepare on every transactional object *O* that participates in a transaction, *T*. This method is responsible for ensuring that no failures happened since *T* accessed *O*, and that *T'*s writes to *O* (if any) are persisted so that *T* can commit even if failures happen later.

Note that the object lock is released before the write is persisted. The write is queued on the ***persist_queue***, which is *drained* by a background worker. It performs the write to persistent storage by appending the version to the active versions log, after which it notifies the caller of Prepare. The log-append is asynchronous, returning a promise. The worker writes all the versions in the queue in one batch operation, which is essential to enabling high write-throughput per object.

Our design is unconventional, using a log-per-object, instead of a log-per-database. It makes sense with cloud storage because storage writes scale out and because cloud storage systems do not currently offer a log abstraction. If write-bandwidth per-server is a bottleneck, a per-database log could be beneficial by enabling batched writes over different objects; our design could exploit it as in [8]. However, our design can obtain the same benefit without a shared log by batching writes to objects in the same shard into one write, which is supported by most cloud storage systems. We suspect a per-database log would do no better than that, a possible topic for future work.

```
Write(S)
TC := Get Transaction Context
working_version = S
Add O to TC.write_set
IF active_versions_log is not empty:
    V = active_versions_log.last.T
    Add V to TC.dependencies
END IF
```

```
Read
TC := Get Transaction Context
Recover()
IF TC.transaction_id does not have working version:
    State = DeepCopy (get_latest_from_logs())
    Add (O, V) to TC.read_set
    IF active_versions_log is not empty:
        Add V to TC.dependencies
    END IF
    working_version = State
END IF
RETURN working_version
```

```
Recover
FOR each entry E in active_versions_log:
    IF TransactionAgent.IsAborted(E.Version):
        Abort(E.Version)
        BREAK
    END IF
END FOR
```

```
Abort(transaction_id)
IF active_versions_log contains transaction_id:
    Remove entry and all subsequent entries from log
END IF
```

```
Prepare(transaction_id, read_version, write)
lock_holder = get_current_lock_holder()
IF lock_holder != transaction_id:
    RETURN false
END IF
IF read_version != null and
    read_version != get_latest_version():
    RETURN false
END IF
Release_lock()
IF !write:
    RETURN true
END IF
Task<bool>  promise
        = active_version_log.append(working_version)
RETURN promise
```

Figure 2 Generic methods of transactional objects

F. Transaction Manager

There is one (active) TM per application deployment, which is responsible for starting, validating, and committing transactions. It is also responsible for ***aborting*** transactions that do not finish executing within a timeout period (even if the transaction is not explicitly aborted), to handle coordinator failures among others. The TM must be ***fault tolerant*** so it can perform its duties even in the face of its own failure.

The TM maintains a log of transaction state in persistent storage. It commits a transaction *T* by adding a log entry for *T*. It uses the well-known presumed-abort optimization where the absence of a commit record in the log implies *T* aborted and thus does not need to log aborts [40]. It also maintains an in-memory version of the log called the ***Transaction Table***. When a TM starts up, it rebuilds the transaction table from the log. To avoid logging each transaction start, it logs a high-water mark and then assigns transaction ids up to that water mark in-memory.

To commit a transaction *T*, the TM receives *T*'s context object. It queues *T* until all of *T*'s dependencies have committed. If any of *T*'s dependencies abort, the TM aborts *T*, too; transactions that depend on *T* will abort when they ask the TM to commit. After all dependencies are satisfied, the TM puts *T* on the ***group commit queue***, to be picked up by a service running in the TM called the ***group commit worker*** and durably written to the log. After *T* is written to the log, the TM replies to the coordinator TA with success. For the purpose of evaluating dependencies, the TM treats *T* as committed once it is added to the group commit queue, without waiting for the log write to complete. This is safe because when a transaction T' that depends on *T* commits, it will be appended later in the log. So if a TM or log failure prevents *T* from committing, T' will not commit either. This is a variant of a well-known optimization described in [24] [34].

For each transaction *T* in the group commit queue, the group commit worker appends a record containing *T*'s transaction id and the identity of objects in its writeset to the durable log in queue order, and assigns *T*'s log sequence number (LSN).

The unavailability of the TM will block the progress of all active transactions and prevent starting new ones. Thus, it is very important that the TM is made highly available, e.g., by enabling it to fail over to an active replica with little downtime after failure or upgrade. This can be done using standard replicated service designs [17][19], which we do not describe here.

G. Checkpointing

As the system runs, the TM and runtime continually append entries to the TM's log and the object's active version log. We need a process of checkpointing to trim these logs and remove old entries in order to prevent the logs from growing arbitrarily large. To do this, the TM puts all committed transactions on the ***checkpoint queue*** in LSN order to be picked up by a service in the TM called the ***checkpoint worker***.

This worker calls the Commit method on every object O in a transaction T's writeset to notify O that T committed and informs O of T's LSN. Then O adds T to its stable version log and removes T from its active version log. It also removes older versions from T's stable version log if the log has grown too large or if the older versions will not be used for snapshot reads again. (See Section III.I for more detail.) Since the TM has limited computational resources and is a potential bottleneck, these calls are offloaded to the application servers and batched for efficiency. After every object in T's writeset acknowledges having added T to its stable version log, the TM trims its own log by removing T's entry.

H. *Garbage Collection*

To prevent the Transaction Table from growing too large and causing the TM to run out of memory, the TM needs to identify which transaction entries are no longer needed and can thus be removed from the table.

The TM needs a Transaction Table entry for a transaction T for as long as T is active. Using presumed-abort, it presumes T aborted if it does not exist in the table; so the TM can remove T's entry immediately after T aborts. If T commits, the TM needs to keep T's entry until all transactions that depend on T have completed. To determine how long to keep T's entry, after the TM checkpoints T, it records the largest transaction id t_{max} of all active transactions. No transaction with id greater than t_{max} will depend on T, because after T is checkpointed, all objects in T's writeset know that T committed. Therefore, when transaction t_{max} completes, the TM can remove the entry for T.

I. *Snapshot Read*

The TM's log totally orders the transactions, so each LSN defines the point-in-time of a consistent snapshot. We use LSNs as timestamps. To read a consistent snapshot at time t, for each object O in a transaction T's readset, T reads the version in O's stable version log that has the largest LSN $\leq t$.

The TM checkpoints transactions in LSN order. Thus, after it checkpoints a transaction whose LSN is t, all entries with LSN $\leq t$ are already in the stable transaction logs. This makes t a safe timestamp to use for a snapshot read. Whenever the checkpoint worker successfully completes a batch, it sends the LSN of the last transaction the batch, t_{safe}, to the TAs in every message, so the TAs can use it for read-only transactions.

A snapshot read with timestamp t fails if the object O has no version with LSN $\leq t$ in its stable version log. This can happen if O trimmed its log and removed old versions. To make this less likely, during checkpointing each object checks its TA's value of t_{safe} to decide whether to remove an old version from its log.

IV. ORLEANS IMPLEMENTATION

This section describes our implementation of the architecture of Section III in Orleans, called **Thorp.**

A. *Introduction to Orleans*

Orleans is a programming framework that extends the .NET Framework to simplify the development of scalable and fault-tolerant distributed applications. We briefly introduce Orleans here, just enough to understand how we added transactions to it.

In Orleans, objects are called ***grains*** and have two important properties**.** First, grains are actors, so that they cannot share state and can communicate only via asynchronous method calls. Second, each grain has a location-transparent identity, which is the only way to reference it. These two properties enable the Orleans runtime to place each grain on any server. Typically, it distributes grains randomly across servers of a deployment, to minimize the chance that any server is a bottleneck.

If a grain G is not currently running when one of its methods is invoked, the Orleans runtime chooses a server on which to activate G, executes the G's constructor on that server, and then performs the method call. It retains a reference to G in its distributed grain directory so that future invocations can be directed to it. If a grain is idle for too long, the Orleans runtime calls the grain's destructor and releases the grain's resources. This model of activate-on-demand is very similar to the demand-paging model of virtual memory. For this reason, Orleans calls it the Virtual Actor Model.

Orleans offers a simple declarative model of grain persistence, where a grain class identifies its member variables that should be persisted. The Orleans runtime maps those variables to persistent storage, which the user specifies via a ***StorageProvider*** attribute. It uses that mapping to populate a grain state when it is activated and to write it back out when it is deactivated. It also allows a grain to save its state at any time, e.g., just before returning from a method call that modifies its state. Developers can ignore this declarative persistence model and write custom code to map object state to storage.

B. *Programming Model*

To ensure the programming model for transactions is natural to Orleans users, transactions are opt-in. They have no impact on programmers unless they use them. Transactional grains (i.e. transactional objects) are identical to regular grains except that accesses to their state occur within a transaction. For practical reasons, a second goal was to minimize modifications to the Orleans runtime. Therefore, much of the implementation is done in base classes that transactional grains use via inheritance.

1) *State*

For each grain class, the developer defines a class that represents the grain state to be protected by transactions. Each is an arbitrary C# class, expressed similarly to declarative persistence. See Figure 3.

```
public class BankAccountState
{
    long Balance { get; set; }
    string Currency { get; set; }
}
```

Figure 3 State definition

2) *Transactional Grain*

For a grain to be declared transactional, its interface has to extend ***ITransactionalGrain***; and the grain implementation has to extend ***TransactionalGrain***<**T**>, where T is the class that defines its state. (See Figure 4.) This is analogous to a non-

Grain Interface

```
public interface IBankAccountGrain :
ITransactionalGrain
{
    Task<bool> Withdraw(long amount);
    Task Deposit(long amount);
}
```

Grain Implementation

```
[StorageProvider(
     ProviderName = "AzureTableStore")]
public class BankAccountGrain :

TransactionalGrain<BankAccountState>,
        IBankAccountGrain
{
  public Task<bool> Withdraw(long amount)
  {
    if (this.State.Balance >= amount)
    {
      this.State.Balance -= amount;
      return Task.FromResult<bool>(true);
    }
    return Task.FromResult<bool>(false);
  }
  public Task Deposit(long amount)
  {
    this.State.Balance += amount;
    return TaskDone.Done;
    }
}
```

Figure 4 Transactional grain

```
public class ATMGrain : Grain, IATMGrain
{
   [Transaction("RequiresNew")]
   public Task<bool> Transfer(
   IBankAccountGrain from,
   IBankAccountGrain to,
   long amount)
   {
   bool can = await from.Withdraw(amount);
   if (can)
       await to.Deposit(amount);
    return can;
   }
}
```

Figure 5 Using transactions in Thorp

transactional grain, whose interface has to extend ***IGrain*** and whose implementation has to extend ***Grain.*** The programmer declares the persistent store for the grain state using the ***StorageProvider*** attribute, just like in declarative persistence. To access the state, the grain uses a C# property field called ***State*** that is defined in the base ***TransactionalGrain***<**`T`**>.

An attribute on each grain method is used to describe where a transaction starts and how grain method calls compose within a transaction. There are three possible values of the attribute, which were described in Section III.A: Requires New, Required, and Not Supported. For example, in the class definition in Figure 5, the ***Transfer*** method will always start a new transaction when called. Since BankAccountGrain is a transactional grain, its methods have the "***Required***" transaction attribute by default. This means the calls to ***Withdraw*** and ***Deposit*** will join the transaction started by ***Transfer***. Note that ATMGrain can start transactions even though it is not a transactional grain itself.

C. Implementation

1) Transactional Grain

Much of the runtime functionality in Section III is implemented via inheriting from TransactionalGrain. The class TransactionalGrain<**`T`**> implements Prepare, Commit, Abort, and Recover, as described in Sections IIIIII.A, III.C, and III.E. Write and Read are implemented as the setter and getter of the State property. Every transactional grain class inherits from TransactionalGrain<**`T`**> and hence picks up its behavior.

Read and Write operate on the object's state, which is of type **`T`**. They map state to storage using Orleans' declarative persistence model, described in Section IV.A. Orleans serialization features are used to generate a deep copy of the state of arbitrary type **`T`** as required by the Write procedure.

2) Runtime extensions

The Orleans Runtime is a set of subsystems that run on each server of an Orleans cluster. It is responsible for turning grain method calls into messages, locating the server that has the grain instance (called an **activation**), forwarding the message to the activation, scheduling the messages to guarantee single-threaded access and finally running the grain code to execute the message and send back the result of the method call. We added the Transaction Agent (TA) as a subsystem to the runtime.

We extend the runtime to provide transaction support. The runtime recognizes the transaction attribute. It calls the TA on the local server to start a transaction and get a transaction context. It passes that context along with calls to methods that have the "***Required***" attribute. Since a transaction can make concurrent calls to grains, it takes the ***union*** of contexts that are returned. Finally, when it is time for a transaction to commit, the context is passed to the TA. The runtime is also modified to enforce the behavior described in Section IV.

The runtime scheduler locks the activation while a method is executing until the method finishes. We modify this behavior for transactional grains, so that the lock is kept even after the method execution is complete, until explicitly released (or due to timeout, to handle failures). While the lock is held, only method calls that are part of the transaction holding the lock (and read-only methods) are allowed to execute; the rest are queued. Wait-Die is used to prevent deadlocks.

3) Transaction Manager

The TM is implemented in C# as a stand-alone multi-threaded service. It uses the Orleans client library to communicate with the Orleans cluster and invoke grain methods. To offload work to the servers during the checkpointing process we described in Section III.G, the TM does not invoke methods directly on TransactionalGrain. Instead, it relies on grains called **CheckpointHelper** to call the

Commit method on each individual TransactionalGrain, which happens on the Orleans servers.

V. EXPERIMENTS

We present an experimental study of Thorp, our Orleans transactions implementation. First, we measure the performance of the TM to verify its scalability. We then compare the performance of transactions to non-transactional persistent Orleans grains to measure the transactions overhead. Finally, we compare Thorp to a traditional strict 2PL/2PC implementation. In this section, when we refer to 2PL, we mean *strict* 2PL.

A. Experimental Setup and Workload

In all experiments we deployed Orleans as a Microsoft Azure cloud service. The deployment has 3 roles:

Orleans Silo: Orleans servers are called **silos**. Each one is configured as a Large Azure virtual machine (VM), which has 4 cores and 7GB RAM. The number of instances varies depending on the experiment.

Orleans TM: This is configured as an Extra-Large Azure VM (8 cores, 14GB RAM). There is always one TM instance.

Workload Generator: This is where we run clients that generate the experiment's workload. It is configured as an Extra-Large Azure VM; the number of instances varies depending on the experiment.

We use Azure Table Storage [38] as durable storage for all state. Each entity (i.e. row) in an Azure Table has a key and a schema-less dictionary as the value. We distribute the grain writes across multiple storage accounts to circumvent Azure's limits for a single storage account.

Each reported result is computed by running the experiment 3 times for 2 minutes and averaging the results. For readability of graphs, we omit standard deviations, which in all cases are under 10%.

B. Summary of Results

The key findings of our experimental evaluation are:

- Thorp's TM can handle hundreds of silos and over 100K transactions per second.
- Read-only transactions have very low overhead, whereas read-write transactions are expensive.
- For a single write-hot grain, the throughput of Thorp is over 20x the throughput of 2PL/2PC.
- Thorp's throughput far outperforms 2PL/2PC for workloads that have write-hot grains. For workloads where load is evenly distributed, Thorp's throughput is similar to 2PL/2PC but it suffers from higher latencies due to batching.

C. TM Transaction Processing

In this experiment, we measure the throughput of the TM's transaction processing pipeline. The workload generator calls the TM to get a transaction id, then creates a synthetic transaction and submits it to the TM to be committed. Synthetic transactions are created in groups, where each transaction takes a dependency on all transactions in the same group with smaller ids. We vary the number of transactions in each group, as well as the order in which the transactions within the group are submitted to the TM for commit, to measure their effects on the throughput. Note that in this experiment the workload generator is running in the same process as the TM, so there is no networking overhead.

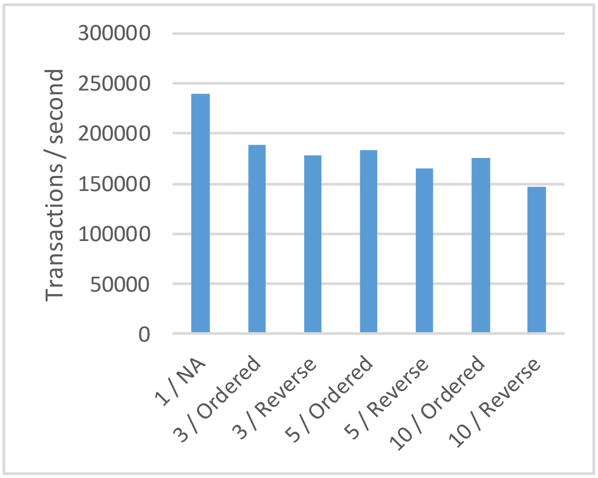


Figure 6 TM Throughput

The results are shown in Figure 6. Each bar is labelled *n*/*r* where *n* is the size of the group and *r* is the order in which transactions are submitted within group. As expected, the throughput declines with an increase in the number of dependencies and when dependencies are satisfied out of order. The effect of reversing the order is larger than increasing the group size. Throughput drops significantly going from group size 1 to group size 3, because transactions that have no dependencies bypass a significant portion of the processing.

Profiling revealed that insertion of new records in the transaction table becomes a bottleneck as the table grows larger. This can be mitigated by parallelizing the sequential garbage collection algorithm so it can remove old records faster as well as by partitioning the table, but we leave that for future work.

D. TM Scalability

Our goal is to measure the TM's end-to-end performance, including networking and TM processing of requests. In this experiment, the TAs run on the workload generator machines. This allows us to run multiple TAs on each VM and thus be able to run with more TAs than if we ran one per silo. Each TA starts transactions and submits them to the TM for validation and commit at a fixed rate (1000 per second). Each transaction has a single grain in its readset and writesets.

As shown in Figure 7, throughput scales linearly with each agent added until it reaches ~110k, after which it remains steady with increasing latency and variance. At that rate we are not limited by CPU utilization or the bottlenecks identified in Section V.A. By profiling, we discovered a bottleneck in the RPC server that causes the processing and deserialization of incoming messages to be single-threaded. We believe that TA to TM messaging can be greatly improved over the current implementation (which we picked for simplicity not efficiency) to reduce overhead and obtain much higher numbers if needed.

E. *Transaction Overhead*

We measure transaction overhead by comparing the performance of transactional grains vs. non-transactional persistent grains. The setup for this experiment has one Orleans silo, where we activate 200 transactional grains, each with 1KB of state. The workload generator has multiple workers. Each worker repeatedly selects a grain id uniformly at random and issues one RPC to that grain, which could either be a read-only operation or a write. We vary the distribution of read-only and write operations for each run as shown in Figure 8. In these and other throughput measurements, the number of workers is chosen high enough to keep the silo CPU utilization above 90%. In latency measurements, the number of workers is fixed at 20.

Read-only transactions have a relatively small effect on throughput, and virtually none on latency. This is expected, since the only additional work done by read-only transactions compared to ordinary reads is a deep copy of the grain state. By contrast, write operations (even non-transactional ones) are much costlier. Write transactions incur 2 extra RPCs, 2 writes to storage, a deep copy, and maintenance of multiple versions. In addition, the effect of batching commit requests to the TM significantly impacts latency. Therefore, a developer should use transactions only when they are really needed, and not just mindlessly do every operation within a transaction.

F. *Single-Grain Micro-benchmark*

Hot data is a worst case for transaction performance and often arises in practice. In this experiment we evaluate Thorp's performance for a single hot grain and compare it to 2PL/2PC and non-transactional persistent grains. The setup is similar to that of Section V.E, except we only activate one grain whose state size is 100 bytes. We start by measuring read-only performance. The results are in Figure 9. As expected, the overhead of deep-copying grain state affects Thorp's Read throughput. Read-2PL/2PC's performance suffers because it has to do an extra RPC to release the locks, though it does not have to run 2PC.

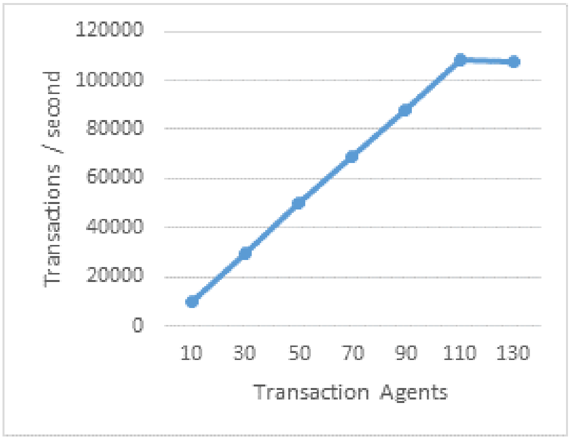


Figure 7 TM Throughput w.r.t # Transaction Agents

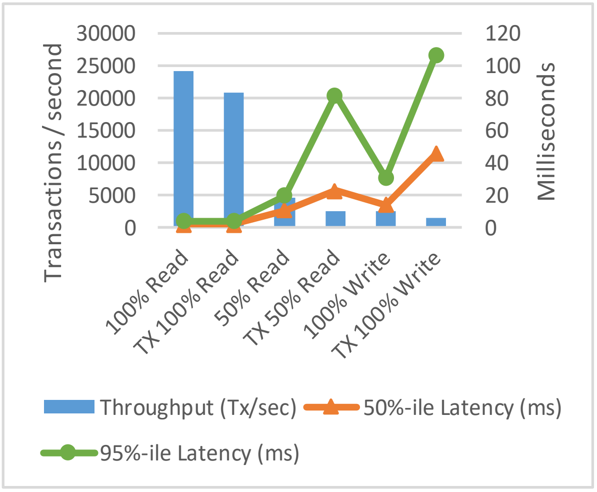


Figure 8 Performance of a single Orleans silo

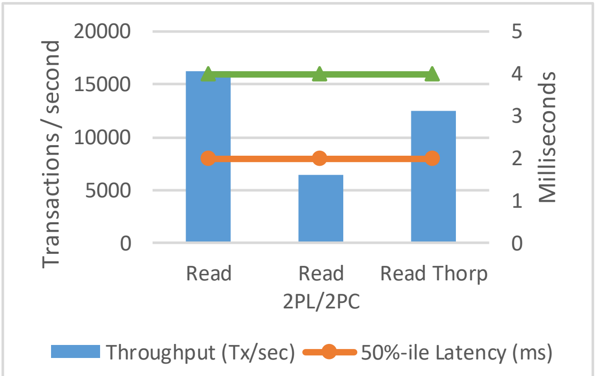


Figure 9 Grain read throughput

Since the grain is locked during the two storage writes of 2PC, we can analytically model the write throughput of 2PL/2PC using the following formula:

1 / [2 * IO_Delay]

where IO_Delay is the latency of writing to storage. If execution time is negligible compared to IO_Delay, then for an average IO_Delay of 10ms we expect 2PL/2PC throughput to be ~50 transactions per second.

In Thorp the write throughput is not directly tied to storage latency. However, there is a limit on the size of an Azure Table entity (64 KB) which limits how many versions a grain can hold onto before they are checkpointed and removed. We define **Max Log Size** as the maximum number of versions the grain can hold in its log before hitting this limit. We define the **Checkpoint Rate** of the grain as the number of checkpoints it performs per second. Then, analytically we can model the write throughput of Thorp using the formula:

Checkpoint Rate * Max Log Size

Figure 10 plots the results of running single-grain write workloads. Thorp's write throughput beats 2PL/2PC by more than 20x. Moreover, it is higher than the plain persistent grain, because the latter is locked while the state is persisted to storage.

The throughput of 2PL/2PC follows the analytical model closely, but not for Thorp. At first glance, one would expect a max log size of 640, since the max entity size is 64KB and the grain state is 100 bytes. Given the observed checkpoint rate of 16, the model predicts a throughput of approximately 10,000 transactions per second. But we observe much less. One reason is that a single silo can perform about 1400 transactions per second, as we saw in Section V.E. While performing all operations on a single grain is more efficient due to increased batching of writes to storage, the execution is single-threaded and the CPU core is saturated at the observed throughput. Another reason is that the state that is written to the log is more than 100 bytes per version, due to versioning and serialization overhead, so the max log size is significantly less than 640.

This experiment highlights Thorp's ability to better handle what [1] calls "linchpin objects" which is one of the main challenges to adapting strongly consistent transactions at scale.

G. *Secondary Index*

The goal of this experiment is to use a realistic workload to study how load distribution affects the performance of Thorp compared to 2PL/2PC. We find database benchmarks to be unrealistic for a middle-tier actor system, since they are designed to run as stored procedures and stress database functionality. Instead, we chose a commonly requested feature in Orleans, namely, the ability to retrieve grains by the value of a property other than the primary key, which requires maintaining a secondary index. The consistency requirements differ from those found in classical database systems in that it is usually sufficient for indexes to be causally and eventually consistent [12]. That is, the index can only be updated after the grain update commits, and the index update should execute shortly after the grain update; some delay is acceptable as long as the index update is not lost or postponed indefinitely.

The workload in this experiment centers around secondary index updates. In addition to being a real-world Orleans scenario, it is also representative of multi-step workflows. Workflow is a popular pattern in middle-tier applications, where transactions are needed to link successive steps reliably.

There are three types of transactional grains in this workload, **ValueGrain**, **IndexGrain** and **IndexUpdateWorkflowGrain**. Each ValueGrain has a State that consists of two string properties, *p* and *p_indexed*. The IndexGrains collectively comprise a secondary index on the p_indexed value of ValueGrains; each IndexGrain's state is a list of references to ValueGrains whose p_indexed value hashes to a key of IndexGrain. There are five types of transactions in this workload:

1- **ValueGrain property read**: Read the value of property p of a single ValueGrain.

2- **ValueGrain property update**: Update the value of property p of a single ValueGrain.

3- **ValueGrain indexed property update**: Update the value of property p_indexed of a single ValueGrain *vg*. Send a message with the pair <vg, vg.p_indexed> to an IndexUpdateWorkflowGrain to process the index updates.

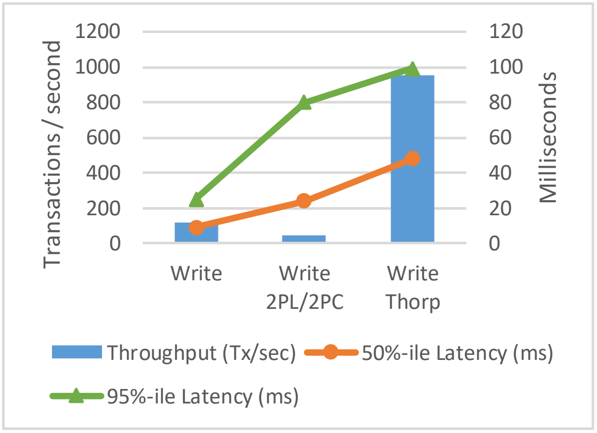
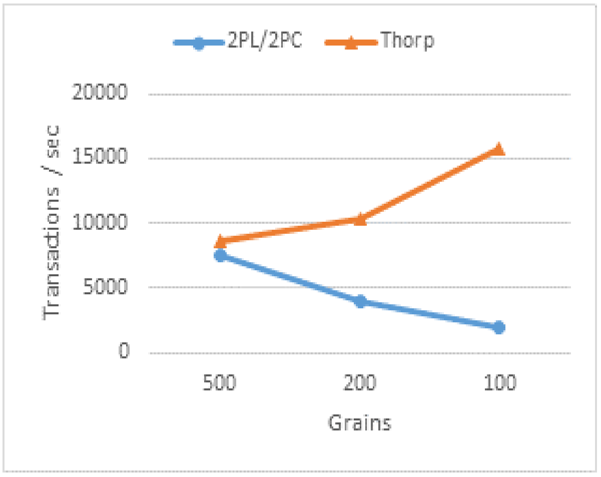


Figure 10 Grain write throughput



	2PL/2PC	Thorp
50%-ile Latency(ms)	27	51
95%-ile Latency(ms)	85	112

Figure 11 Secondary index workload results

4- **IndexGrain removal**: Read the value of p_indexed of a ValueGrain. If the value changed, remove the grain's reference from the old IndexGrain.

5- **IndexGrain insertion**: Read the value of p_indexed of a ValueGrain. and insert a reference into the appropriate IndexGrain.

After transaction 3 commits, the IndexUpdateWorkflowGrain is responsible for executing transactions 4 and 5. Note that the ability to read the ValueGrain's state and updating the index atomically means that transactions 4 and 5 are idempotent and the messages do not have to be processed in order.

This experiment uses 20 Orleans silos. Like Section V.E, the workload generator has multiple workers, each repeatedly selecting a grain id uniformly at random and issuing one RPC to the ValueGrain with that id, which performs one of the ValueGrain transactions with distribution 50% read, 30% update and 20% indexed update. Since Orleans load-balances grains randomly, only 5% of ValueGrains are co-located with their old or new IndexGrain.

We vary the number of grains in each experiment and measure the throughput (total transactions per second) and latency of index update operations (i.e. transactions 4 and 5). The results are in Figure 11. We measured the latency to perform the index update operations without transactions or persistence; the average latency is ~2 ms.

Once again we see that the latency of Thorp is higher than 2PL/2PC due to the effect of batching. The throughput for both implementations is similar when the load is distributed evenly across many grains, with Thorp having the advantage due to faster reads. However, when there is a small number of write-hot grains, 2PL/2PC's throughput drops significantly as it becomes limited by the high storage latency, while Thorp's throughput actually *improves* due to increased batching efficiency. With 100 hot grains, Thorp performs ~7x better than 2PL/2PC.

VI. RELATED WORK

Commit dependencies: The notion of commit dependency was introduced in the ACTA framework [21]. We know of two prior works that use the concept. In SL**,** if a transaction T_1 updates x and a later transaction T_2 reads x, then T_2 speculates by having two incarnations, T_{21} that reads T_1's before-image of x and T_{22} that reads its after-image [42]. T_{21} and T_{22} both take a commit dependency on T_1. If T_1 commits, T_{22} is retained, else T_{21} is retained. The simulation study in [42] of a distributed DBMS shows that SL gets better throughput than 2PL by overlapping speculative executions of T_2 with T_1, at the cost of more CPU load. By contrast, in Thorp, T_2 only takes a dependency on T_1 after T_1 terminates. Hence, it has no more CPU load, and its performance benefit derives from batching storage updates. Unlike SL, Thorp is implemented—not simulated—in commercial middle-tier programming framework, not a DBMS.

Microsoft's Hekaton uses a more limited form of commit dependency [25]. It allows T_2 to take a commit dependency on T_1 if T_2 started after T_1 finished execution and entered the validation phase but has not yet committed. Thus, it benefits from overlapping T_2's execution with T_1's validation. Unlike Thorp, Hekaton is a DBMS, not a programming framework, and is not distributed.

Actor Frameworks: An earlier design of Orleans had a transaction mechanism based on multi-master replication [18]. It only provided snapshot isolation, not serializability. It was dropped before Orleans was released because it performed poorly and users found it too complex [10]. Akka is a Java-based actor framework that has transactions, but only on a single machine [2]**.** Orleans is compared to Akka in [10].

Weak consistency models: Although the need for distributed transactions was a given 15 years ago, since then there has been a trend toward using alternative models that provide weaker consistency but better performance or partition tolerance. Some systems offer transactions only within a shard [7][11][19]. Some offer weaker consistency, either with transactions [33][35] [37] or without [20][23][36]. Some avoid transactions altogether and use a workflow model instead [27][46] [47]. By contrast, Thorp offers full ACID transactions.

Distributed Transactions: Work on distributed transactions started in the 1970's [3][15]. Transactional middleware of the 1980's offered distributed transactions using a centralized TM [14], which was standardized by X/Open [44]. Multi-version concurrency control also began appearing in that period [13]**.** Recent examples of distributed ACID transaction systems include Calvin, Deuteronomy, and Spanner.

Spanner is Google's geo-replicated database system [22]. Like Thorp, it uses multi-versioning to support lock-free read-only transactions and 2PL/2PC for read-write transactions. However, unlike Thorp it holds write locks until after a transaction commits, which limits throughput as we discussed earlier. Moreover, it uses the TrueTime API, enabled by GPS and atomic clocks, to assist in timestamp assignment. This makes it hard to adopt in a system like Thorp that is intended to run in public cloud environments that lack that support.

Calvin provides scalable distributed transactions by having data servers agree on the ordering of transactions before executing them [45]. Its design depends on deterministically executing a transaction T once its position in the ordering is determined. That requires knowing the data items accessed by T beforehand, which is highly problematic for middle-tier micro-services, since their behavior is dynamic and unpredictable.

Deuteronomy is a distributed transaction system comprised of two components [34]: a Transaction Component (TC) that is responsible for concurrency control and recovery, and one or more Data Components (DC) that provide data storage and access. Although this architecture is targeted at database systems rather than middle-tier services, its componentization bears some similarities to Thorp's. However, unlike our TM, the TC is heavily involved in the execution of every transaction.

VII. CONCLUSION

We presented an architecture and implementation of distributed transactions for cloud-based middle-tier applications composed of micro-services. In a cloud environment, storage latency limits performance, because each transaction requires two round-trips to storage and during that time no other transactions can update its readset or writeset. We showed how to avoid this performance limitation by allowing a transaction to release its write locks during phase-one of two-phase commit, tracking commit dependencies, and ensuring it does not commit until the transactions it depends on have committed. We implemented our architecture in Orleans, a middle-tier actor framework, and showed its throughput is much higher than that of the classical approach. We expect to release Thorp as open source before ICDE 2017.

There is much that can be done to extend this work. On the research side, one could experiment with variations of our technique to identify further optimizations. For example, one could try multi-version optimistic concurrency control, so transactions can read or overwrite data that was last written by a still active transaction. This should increase the maximum throughput when the transaction conflict rate is low. On the practical side, it would be beneficial to add a hot standby TM, for faster failover. Scalability could be improved by parallelizing garbage collection of the transaction table and speeding up the RPC server (Sections V.C and V.D).

It would be beneficial to avoid deep copying the entire object state when a small update is made to a big structure, e.g., a dictionary. One way is to implement a custom transactional variation of the data structure that can log and undo incremental updates. To avoid altering the API, another way is to auto-generate wrappers that transparently handle log and undo operations.

Acknowledgments

We thank Sergey Bykov, Gabi Kliot, and Jorgen Thelin for design suggestions and for helping us with Orleans. We also thank Doug Terry for his help.

References

[1] Ajoux, P., N. Bronson, S. Kumar, W. Lloyd and K. Veeraraghavan, "Challenges to Adopting Stronger Consistency at Scale," HotOS 2015.

[2] Akka documentation, http://akka.io/docs/

[3] Alsberg, P., J. D. Day: A Principle for Resilient Sharing of Distributed Resources. ICSE 1976: 562-570.

[4] Amazon: "DynamoDB," https://aws.amazon.com/dynamodb/.

[5] Amazon: "Amazon Elastic Container Service," https://aws.amazon.com/ecs/.

[6] Armstrong, J.: "Erlang," CACM 53, 9: 68-75 (2010).

[7] Baker, J. C. Bond, J.C. Corbett, JJ Furman, A. Khorlin, J. Larson, J-M. Leon, Y. Li, A. Lloyd, V. Yushprakh: "Megastore: Providing Scalable, Highly Available Storage for Interactive Services". CIDR 2011: 223–234.

[8] Balakrishnan, M., D. Malkhi, T. Wobber, M. Wu, V. Prabhakaran, M. Wei, J.D. Davis, S. Rao, T. Zou, A. Zuck: "Tango: distributed data structures over a shared log." SOSP 2013: 325-340

[9] Berenson, H., P. Bernstein, J. Gray, J. Melton, E. O'Neil and P. O'Neil, "A critique of ANSI SQL isolation levels.," SIGMOD 1995.

[10] Bernstein, P.A., S. Bykov, A. Geller, G. Kliot, J. Thelin, "Orleans: Distributed Virtual Actors for Programmability and Scalability," MSR-TR-2014-14, http://research.microsoft.com/apps/pubs?id=210931.

[11] Bernstein, P., I. Cseri, N. Dani, N. Ellis, A. Kalhan, G. Kakivaya, D.B. Lomet, R. Manne, L. Novik, T. Talius: "Adapting Microsoft SQL server for cloud computing." ICDE 2011: 1255-1263.

[12] Bernstein, P.A., M. Dashti, T. Kiefer, D. Maier: "Indexing in an Actor-Oriented Database." CIDR 2017, to appear. Draft available on request.

[13] Bernstein, P.A., N. Goodman: "Multiversion Concurrency Control - Theory and Algorithms." TODS 8(4): 465-483 (1983).

[14] Bernstein, P.A., V. Hadzilacos, N. Goodman: "*Concurrency Control and Recovery in Database Systems*." Addison-Wesley, 1987.

[15] Bernstein, P. A., E. Newcomer: "Principles of Transaction Processing", *Morgan Kaufmann*, 2nd ed., 2009.

[16] Bernstein, P.A., D.W. Shipman, J.B. Rothnie Jr.: "Concurrency Control in a System for Distributed Databases." TODS 5(1): 18-51 (1980)

[17] Burrows, M.: "The Chubby lock service for loosely-coupled distributed systems." OSDI 2006: 335-350.

[18] Bykov, S., A. Geller, G. Kliot, J. Larus, R. Pandya, and J. Thelin: "Orleans: Cloud Computing for Everyone." SOCC 2011, 16:1-16:14

[19] Calder, B., J. Wang, A. Ogus, N. Nilakantan, A. Skjolsvold, S. McKelvie, Y. Xu, S. Srivastav, J. Wu, H. Simitci, J. Haridas, C. Uddaraju, H. Khatri, A. Edwards, V. Bedekar, S. Mainali, R. Abbasi, A. Agarwal, . M. Fahim ul Haq, . M. Ikram ul Haq, D. Bhardwaj, S. Dayanand, A. Adusumilli, M. McNett, S. Sankaran, K. Manivannan, L. Rigas: "Windows Azure Storage: a highly available cloud storage service with strong consistency." SOSP 2011: 143-157.

[20] Chang, F., J. Dean, S. Ghemawat, W.C. Hsieh, D.A. Wallach, M. Burrows, T. Chandra, A. Fikes, R.E. Gruber, "Bigtable: A Distributed Storage System for Structured Data", ACM TOCS 26(2): 4:1-4:26 (2008).

[21] Chrysanthis, P.K., K. Ramamritham, "ACTA: a framework for specifying and reasoning about transaction structure and behavior," SIGMOD 1990: 194-203.

[22] Corbett, J.C., J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A/ Gubarev, C. Heiser, P. Hochschild, W.C. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, D. Woodford: "Spanner: Google's Globally-Distributed Database." ACM Trans. Comput. Syst. 31(3): 8 (2013).

[23] DeCandia, G., D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, W. Vogels: "Dynamo: amazon's highly available key-value store." SOSP 2007: 205-220.

[24] DeWitt, D.J., R. H. Katz, F. Olken, L. D. Shapiro, M. R. Stonebraker, D. Wood: "Implementation Techniques for Main Memory Database Systems." SIGMOD 1984: 1-8.

[25] Diaconu, C., C. Freedman, E. Ismert, P. Larson, P. Mittal, R. Stonecipher, N. Verma, M. Zwilling: "Hekaton: SQL Server's Memory-Optimized OLTP Engine." SIGMOD 2013

[26] Docker. [Online]. Available: http://www.docker.com/.

[27] Garcia-Molina, H., K. Salem: "Sagas." SIGMOD 1987: 249-259.

[28] Google: "Google Cloud BigTable," https://cloud.google.com/bigtable.

[29] Google: "Google Container Engine," https://cloud.google.com/container-engine/.

[30] Gray, J., A. Reuter: "Transaction Processing: Concepts and Techniques." Morgan Kaufmann 1993.

[31] Helland, P.: "Life beyond Distributed Transactions: an Apostate's Opinion." CIDR 2007.

[32] Kubernetes. http://kubernetes.io/.

[33] Lakshman, A., P. Malik: "Cassandra – A Decentralized Structured Storage System." Operating Systems Review 44(2):35-40 (2010).

[34] Levandoski, J.J., D. Lomet, M.F. Mokbel, K. K. Zhao, "Deuteronomy: Transaction Support for Cloud Data," CIDR 2011.

[35] Li, C., D. Porto, A. Clement, J. Gehrke, N. Preguic, R. Rodrigues: "Making Geo-Replicated Systems Fast if Possible, Consistent when Necessary." OSDI 2012: 265-278.

[36] Lloyd, W., M.J. Freedman, M. Kaminsky, D.G. Andersen: "Don't settle for eventual: scalable causal consistency for wide-area storage with COPS." SOSP 2011: 401-416.

[37] Lloyd, W., M.J. Freedman, M. Kaminsky, D.G. Andersen: "Stronger Semantics for Low-Latency Geo-Replicated Storage." NSDI 2013: 313-328.

[38] Microsoft: "Microsoft Azure Storage," https://azure.microsoft.com/en-us/services/storage/.

[39] Microsoft: "Azure Service Fabric," https://azure.microsoft.com/en-us/services/service-fabric/.

[40] Mohan, C., B.G. Lindsay, R, Obermarck: "Transaction Management in the R* Distributed DBMS." TODS 11(4): 378-396 (1986).

[41] Peng, D., F. Dabek: "Large-scale Incremental Processing Using Distributed Transactions and Notifications." OSDI 2010: 351-264

[42] Reddy, P.K., M. Kitsuregawa: "Speculative Locking Protocols to Improve Performance for Distributed Database System." IEEE TKDE. 16(2): 154-169 (2004)

[43] Stearns, R.E., P.M. Lewis II, D.J. Rosenkrantz. "Concurrency Controls for Database Systems." IEEE FOCS 1976: 19-32.

[44] The Open Group: Distributed Transaction Processing: Reference Model, Version 3. https://www2.opengroup.org/ogsys/catalog/G504

[45] Thomson, A., T. Diamond, S-C Weng, K. Ren, P. Shao, D.J. Abadi: "Calvin: fast distributed transactions for partitioned database systems." SIGMOD 2012: 1-12.

[46] Xie, C., C. Su, M. Kapritsos, Y. Wang, N. Yaghmazadeh, L. Alvisi, P. Mahajan: "Salt: Combining ACID and BASE in a Distributed Database." OSDI 2014: 495-509.

[47] Zhang, Y., R. Power, S. Zhou, Y. Sovran, M.K. Aguilera, J. Li: "Transaction chains: achieving serializability with low latency in geo-distributed storage systems." SOSP 2013: 276–291.