Geo-Distribution of Actor-Based Services

January 24, 2017

Technical Report
MSR-TR-2017-3

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052

Important

This document is *work in progress*. Feel free to cite, but note that we will update the contents without warning (pages are timestamped at the bottom right), and that we are likely going to publish the content in some future venue, at which point we may update this paragraph.

Geo-Distribution of Actor-Based Services

Philip A. Bernstein
Sebastian Burckhardt
Alok Kumbhare
Jorgen Thelin
Microsoft Research
{philbe,sburckha,alokk,jthelin}@microsoft.com

Sergey Bykov
Muntasir Raihan Rahman
Microsoft
{sbykov,murahman}@microsoft.com

Natacha Crooks
University of Texas, Austin
ncrooks@cs.utexas.edu

Jose Faleiro
Yale University
jose.faleiro@yale.edu

Gabriel Kliot
Google
gkliot@google.com

Vivek Shah
University of Copenhagen
bonii@di.ku.dk

Adriana Szekeres
University of Washington
aaasz@cs.washington.edu

Abstract

Many service applications use actors as a programming model for the middle tier, to simplify synchronization, fault-tolerance, and scalability. However, efficient operation of such actors in multiple, geographically distant datacenters is challenging, due to the very high communication latency.

We present GEO, an open-source geo-distributed actor system that improves performance by caching actor states in one or more datacenters, yet guarantees the existence of a single latest version by virtue of a distributed cache coherence protocol. GEO supports both volatile and persistent actors, and supports updates with a choice of linearizable and eventual consistency. Our evaluation on several workloads shows substantial performance benefits, and confirms the advantage of supporting both replicated and single-instance coherence protocols as configuration choices. For example, replication can provide fast, always-available reads and updates globally, while batching of linearizable storage accesses at a single location can boost the throughput of an order processing workload by 7x.

1. Introduction

Actors have emerged as a useful abstraction for the middle tier of scalable service applications that run on virtualized cloud infrastructure in a datacenter [32, 33, 42]. In such systems, each actor is a single-threaded object with a user-defined meaning, identity, state, and operations. For example, actors can represent user profiles, articles, game sessions, devices, bank accounts, or chat rooms. Actors resemble miniature servers: they do not share memory, but communicate asynchronously, and can fail and recover independently. Actor systems scale horizontally by distributing the actor instances across a cluster of servers.

In a traditional bare-bones actor system, the developer remains responsible for the creation, placement, discovery, recovery, and load-balancing of actors. A newer generation of actor models [2, 32, 33], called *virtual actor models*, automate all of these aspects. The developer specifies only (1) a unique key for identifying each actor, and (2) how to save and load the actor state to/from external storage, if persistence is desired. As virtual actor systems can activate and deactivate actors based on use, they strongly resemble caches [26, 36, 38, 53] and provide similar performance benefits.

Geo-Distribution Challenge. Todays cloud platforms make it easy to operate a service in multiple datacenters, which can improve latency and availability for clients around the world. The virtual actor model is a promising candidate for architecting such services. It is not clear, however, *how to make it perform acceptably across continents*. Given the high communication latency (e.g., about 150ms round-trip between California and the Netherlands), a naive reuse of existing APIs and protocols that were designed for single datacenter clusters (with less than 2ms roundtrips between servers) has a poor chance of success.

Our experience suggests that to perform within a range that is appealing in practice, a geo-distributed virtual actor system must *exploit locality, if present*. For example, if an

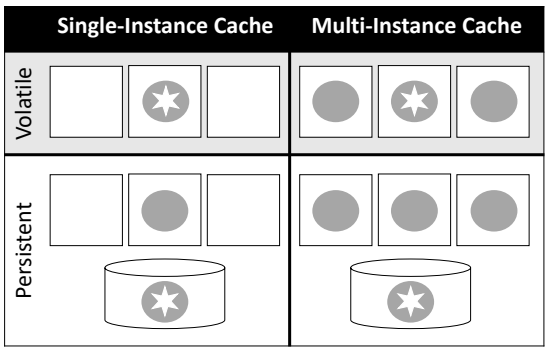

Figure 1. The four actor configuration options. Squares are clusters, Cylinders are the storage layer, circles are copies of the actor state, and the star marks the latest version (primary copy).

actor is accessed mostly from a single datacenter, those accesses should not incur any geo-remote calls. On the other hand, a solution should support *replication where necessary*. For example, if an actor is frequently accessed by multiple datacenters, accesses should utilize locally cached copies. Our system, called GEO, solves these requirements using new mechanisms and a new variant of the actor API.

GEOs implementation is structured hierarchically: a set of clusters is federated into a loosely connected multi-cluster. Each cluster maintains a local elastic actor directory that maps actors to servers, using existing mechanisms in virtual actor systems. To provide a simple, global view of the system and stay true to the virtual actor model, GEO automatically coordinates actor directories and actor states across all the clusters via several *distributed coherence protocols*. These protocols are non-trivial, as they must scale out, gracefully handle node failures, network partitions, and live configuration changes at the cluster and the multi-cluster level. They do not exhibit a single point of failure or contention.

GEO introduces a novel *versioned* actor-state API that gives the runtime more room for optimizations (such as replication and batching) when reading or updating actor state. Yet the application logic remains simple. The API offers fast local reads (of approximate state based on local cache) and fast local updates (via a background queue). Importantly, the use of these locally consistent operations is entirely optional: all actors support globally consistent, *linearizable* reads and writes, which are guaranteed to read or write the latest version.

1.1 Actor Configuration Options

To perform better across a wide range of requirements, GEO supports several configuration options (Fig.1). Each actor can be declared as either *volatile* (latest version resides in memory and may be lost when servers fail) or *persistent* (latest version resides in the storage layer). Furthermore, the caching policy for each actor can be declared as *single-instance* (state is cached in one cluster) or *multi-instance* (state is cached in every cluster). These choices can greatly affect performance. For example, caching multiple instances can reduce the access latency for actors without locality; but using a single instance can improve throughput for actors with locality, and for actors with a high update rate. We discuss these observations in the evaluation section.

1.2 Novelty and Relevance

Prior work on geo-distributed services has heavily focused on the challenge of providing geo-replicated storage [9, 13, 21, 23, 39, 45], usually using quorum-based algorithms. A distinguishing feature of our actor-based approach is that it separates geo-distribution from durability. Our protocols are not responsible for durability, because actors are either declared volatile (developers expressly forfeit durability) or persisted externally (developers want durability provided by a storage layer of their choice). Our protocols are not quorum-based, but use efficient primary-copy replication; they resemble cache coherence protocols used in multiprocessors. The storage layer (which often uses quorum-based algorithms internally) may be in a specific datacenter or itself be geo-distributed. Our system is largely agnostic of these details. Users can select any storage system, except that our current implementation assumes the storage layer supports strong consistency and conditional updates.

This separation of geo-distribution from durability is highly relevant for actor-based services:

1. Providing durability for volatile actors is wasteful. Volatile actors are pervasive in interactive or reactive applications, because the actor state is often a view of other state (e.g. other actors, or external state), and can thus be reconstructed or recomputed when lost. For example, if an object tracks current participants of a game and the current list of players is lost in a failure, it can quickly be reestablished, because each participant sends periodic heartbeats.
2. Developers want full control over where and how to store data. Often, there are important non-technical reasons for requiring that data be durably stored in a specific geographic location and/or a specific storage system and/or a specific format, such as: cost, legacy support, tax laws, data sovereignty, or security.
3. An un-bundling of functionality into independent components accelerates innovation, because it fosters independent competition for each aspect. This is clearly reflected in how cloud services are built these days, using a plethora of components, many of which are open-source.

1.3 Contributions

Our main contributions are the programming model, the system implementation, and the performance evaluation.

- GEOs **programming model** provides an effective separation of concerns between developing geo-distributed applications and the challenge of designing and implementing robust, distributed protocols for managing actor state. It is suitably abstract to allow plugging in and combining various such protocols. Developing such protocols is a subtle and complex task: hiding it beneath a simple API puts geo-distributed applications within the reach of mainstream developers.
- GEOs full-function **implementation** is open-source. A pre-release is available on GitHub citegeo and is being used in a commercial setting by an early adopter. It includes a new optimistic protocol for distributed datacenters to ensure that the cache contains at most one instance of an object worldwide. It also includes a new consistency protocol for synchronizing the state of persistent cache instances with each other and with storage, using batching to improve throughput.
- Our **evaluation** of GEO compares the performance of various consistency protocols and configuration options, showing their latency and throughput benefits.

The paper is organized as follows. We describe the programming model in §2, protocols to implement the model in §3, experimental results in §4, related work in §5, and the conclusion in §6.

2. Programming Model

We start by describing GEO from the viewpoint of a developer/operator who writes application code and operates the service. We define a *cluster* to be a set of servers, called *nodes*, connected by a high-speed net-work. Clusters are elastic and robust: nodes can be added or removed depending on load, and node failures are automatically detected and tolerated. A datacenter may contain multiple clusters, e.g., to group nodes into distinct failure zones that operate and fail independently.

Multi-Cluster Configuration. When deploying a cluster, the developer configures its cluster id, which must be unique. At any time (except when a configuration change is already underway), the operator can specify or modify the list of cluster ids that comprise the current multi-cluster.

Actor Declarations. Our actor model is based on virtual actors as used by the Orleans [3, 8, 33], Orbit [32], and Service Fabric Reliable Actors [42] frameworks. For each class of actors, the developer defines actor identity, actor interface, actor state, and code that implements the operations. The identity of the actor is determined by the combination of its class and a key, which is typically a string, integer, or GUID. The developer also declares which of the four configuration combinations in Fig. 1 to use. For persistent actors, the developer specifies how to save/restore the actor state to/from storage.

Activation and Deactivation. As in other virtual actor systems, actor instances are not explicitly created or deleted. Rather, they are automatically activated when used (i.e., when an operation specified in the actor interface is invoked), and deactivated when unused for some period of time. Single-instance actors are activated only in the cluster where they are first accessed, and multi-instance actors are activated in all clusters.

Note that the single-instance policy can exploit locality if all accesses are in the same datacenter, or if accesses by different datacenters are separated in time. For example, suppose a user Bob connects to a datacenter c, which causes Bobs profile p to be loaded from storage and cached in memory. Now Bob logs off and flies to another continent. Since he is off-line for a while, the cached instance of p in c is evicted. When Bob logs in to a local datacenter d at his destination, p is loaded into memory at d.

Actor State. The state of an actor can be read and written only from within its own operations (encapsulation). To do so, we support two alternative APIs with different tradeoffs:

1. The *basic state API* is specialized for single-instance actors. Actors using this API can read and update their state directly, but can execute only one operation at a time. The big advantage of the basic API is its simplicity. It is a perfect match for the volatile single-instance scenario (in Figure 1), since synchronous reads and writes on main memory run fast. However, it does not work for the multi-instance case, and it can suffer from performance problems in the persistent case when writing frequently and synchronously to storage.
2. The *versioned state API* is more involved, but also more powerful, and it is compatible with multi-instance configurations. It adds a level of indirection when reading and updating actor state, allows multiple reads and updates to proceed at the same time, and supports both local and global consistency. We describe it in more detail in the next subsection.

2.1 Versioned State API

The Versioned State API manages actor state indirectly, using *state objects* and *update objects*. For a state object `s` and update object `u`, the programmer must implement a deterministic method `s.apply(u)` that defines the effect of the update. For example, for an actor representing a counter that supports add and reset operations, we may define state and update objects as:

```
class CounterState {
  int count = 0;
  apply(Add x) { count += x.amount; }
  apply(Reset x) { count = 0; }
}
class Add { int amount; }
class Reset { }
```

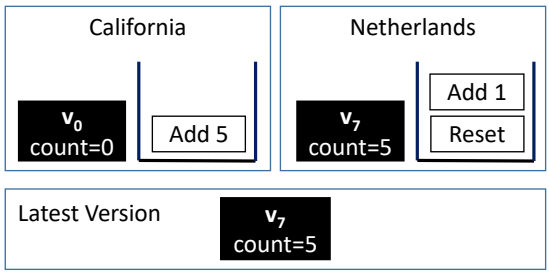

Figure 2. Sample snapshot of the internal state of an actor that uses the versioned state API and has two instances in different datacenters.

Conceptually, the consistency protocol applies updates one at a time to the latest version, thereby creating a sequence of *numbered global versions*. The initial state v_0 is defined by the default constructor of the state object. Every time an update is applied, the version number is increased by one. We visualize how the protocol manages states and updates in a local and global context as shown in Fig. 2 using state objects (black boxes) and update objects (white boxes) of the same types as in the counter example. There are two instances of the same actor, one in California, and one in the Netherlands. Each stores (1) a local copy of the last known version, and (2) a queue of unconfirmed updates (updates enter at the top and drain at the bottom). The bottom rectangle shows the latest version of the state.

Background Propagation. At all times, the consistency protocol runs in the background on each instance of an actor to propagate updates. It applies each queues updates to the latest version in order, interleaving them with updates from other queues, and it propagates the latest version to all instances. These tasks require communication. Thus, they may be slow or stall temporarily (for example, if intercontinental communication or storage are down). However, by design, such stalls do not impact the availability of an actor: it can always continue to be read and updated locally.

Where is the Latest Version? The location of the latest version depends on the configuration and protocol. For our current system, it is always located either in external storage (for persistent actors) or in memory (for volatile actors), as shown by the stars in Fig. 1. Importantly, regardless of the configuration, *the programmer can always rely on the existence of a latest version*, and can directly read and update it. This provides unified semantics for many consistency protocols without exposing configuration details such as the number of replicas and the nature of quorum configurations.

Local Operations. In many situations, it is acceptable to work with a somewhat stale actor state and to delay the confirmation of updates [9, 18, 47]. For example, a website may display a leaderboard, chat room discussion, or item inventory using a stale state, or an unconfirmed tentative state, instead of the latest version.

The Versioned API supports this in the form of *queued updates* and *cached reads*. They are local operations that complete quickly in memory, i.e., without waiting for any I/O. For updates, the programmer calls the function

```
void enqueue(Update u)
```

It appends the update to the local queue and then returns. To read the current state, the programmer can call

```
pair<State,int> read_confirmed()
```

It returns the locally cached version of the state, which is consistent but possibly stale, and its version number. For example, in Fig. 2, in California it returns version v_0 with count=0, which is stale. In the Netherlands it returns version v_7 with count=5, which is up-to-date. We offer a second local read variant:

```
State read_tentative()
```

It takes the cached version and superimposes the unconfirmed updates in the queue. For example, in Fig. 2, in California it returns a state with count=5 (add 5 to 0) and in the Netherlands, it returns a state with count=1 (reset 5 to 0, then add 1). A state returned by read_tentative does not have a version number because it is not part of the global sequence. There is no guarantee that it matches any past or future version.

Linearizable Operations. In some situations, we are willing to trade off latency for stronger consistency guarantees. For example, in the TPC-W benchmark [50], we guarantee to never oversell inventory, which requires coordination. To this end, GEO supports two synchronization primitives confirm_updates and refresh_now.

The synchronization primitive confirm_updates waits for the queue of the given instance to drain. It can be used to provide linearizable updates as follows, where await waits for the asynchronous operation that follows it to return:

```
linearizable_update(Update u) {
  enqueue(u);
  await confirm_updates();
}
```

The synchronization primitive refresh_now drains the queue like confirm_updates, but additionally, it also always fetches the latest version. It can be used to provide linearizable reads as follows:

```
linearizable_read() {
  await refresh_now();
  return read_confirmed();
}
```

Note that the synchronization placement is asymmetric: refresh_now precedes the read, while the call to confirm_updates follows the update. This ensures *linearizability* [16]: the operation appears to commit at a point of time after the function is called and before it returns.

Consistency Discussion. The Versioned API presented here is a variation of the global sequence protocol (GSP) operational consistency model [7, 25], applied on a per-actor basis. GSP uses an equivalent formulation based on totally-ordered broadcast, but assumes a single database rather than a set of independent actors, which limits scalability. GSP is itself a variation of the total-store order (TSO) consistency model for shared-memory multiprocessors. TSO has a different data abstraction level (read/write memory vs. read-/update application data) and all participants always read the latest version.

Our model preserves the local order of updates, and updates do not become visible to other instances until they are part of the latest version. Therefore, in the terminology of [5, 47, 48], the model supports causality, read-your-writes, monotonic reads, and consistent prefix of operations on the same object.

There are no ordering or atomicity guarantees about accesses to different actors, as each actor runs its protocol independently. This is important for horizontal scalability (which is the principal advantage of actor systems). Though it may complicate life for developers, it has not surfaced as a major issue. For one, ordering can be enforced by using linearizable operations (linearizability is compositional). Also, actors can often be made coarse-grained enough to cover desired invariants. For example, representing chat rooms rather than chat messages ensures causality of the chat content. Finally, applications can use actors to track workflows when coordinating updates across multiple actors (as in the order processing mini-benchmark in §4.5.1).

3. Implementation

GEO [15] is implemented in C# as extensions to Orleans, an open-source distributed actor framework available on GitHub [33]. GEO connects several elastic Orleans clusters over a wide-area network. The Orleans runtime uses consistent hashing to maintain a distributed, fault-tolerant directory that maps actor keys to instances [3]. It already handles configuration changes and node failures within a cluster, fixing the directory and re-activating failed instances where necessary. However, Orleans does not provide mechanisms for coordinating actor directories and actor state between clusters. To this end, we designed several distributed protocols.

- **Global Single Instance (GSI)** protocol for the single-instance caching policy. It coordinates actor directories between clusters to enforce mutual exclusion strictly (in pessimistic mode) or eventually (in optimistic mode).
- **Batching Compare-and-Swap (BCAS)** protocol for persistent actors. It implements the Versioned API on persistent storage that supports conditional updates.
- **Volatile Leader-Based (VLB)** protocol for volatile multi-instance actors. It implements the versioned API, storing the latest version in memory, at a fixed leader.

	Volatile	Persistent
Basic API	GSI	GSI (sync.)
Versioned API		
Single-Instance Policy	n/a	GSI + BCAS
Multi-Instance Policy	VLB	BCAS

Table 1. Protocol selection for a given API and policy.

These protocols reside at different system layers: the GSI protocol coordinates actor directories (it is an extension of Orleans directory protocol), while the BCAS and VLB protocols coordinate actor state (communicating among actor instances and with external storage).

Optimistic GSI and BCAS are *robust*: some actor instance is always available even if a remote cluster or storage is unreachable. This is important; datacenter failures are sufficiently common that large-scale web-sites routinely engineer for them [13, 29, 34, 35].

Live Multi-Cluster *Configuration Changes* are supported by all protocols, with some limiting assumptions: a configuration change must be processed by all nodes in all clusters before the next configuration change is injected. Also, the change may add or remove clusters, but not both at the same time.

GEO is open for experimentation, and allows plugging in different consistency protocols and variations beneath the same API. This can be helpful to devise custom protocols for specific settings (e.g. alternative forms of persistent storage, such as persistent streams). Also, it facilitates research on consistency protocols.

The protocol implementations match up with the chosen API and configuration as shown in Table 1. The n/a indicates an unsupported combination (not difficult to implement, but has no performance benefits).

3.1 GSI Protocol

At its heart, the global single-instance protocol is simple. When a cluster c receives an operation destined for some actor (identified by a key k), it checks its directory entry for k to see if an instance exists in c. If so, it forwards the operation to that instance. If not, it sends a request message to all other clusters to check whether they have an active instance. If it receives an affirmative response from a remote cluster c', it then forwards the request to c'. Else, it creates a new instance, registers it in its local directory, and processes the operation. But there are several problems with this sketch:

1. Contacting all clusters for every access to a remote actor instance is slow and wasteful.
2. When two clusters try to activate an instance at about the same time, their communication may interleave such that neither is aware of the other, and both end up activating a new instance.

3. If any of the remote clusters are slow to respond, or do not respond at all, the protocol is stuck and the actor is unavailable.

We solve these three problems as follows.

Cached Lookups. After determining that an instance exists in a remote cluster, we cache this information in the local directory. If the actor is accessed a second time, we forward the operation directly to the destination.

Race arbitration. A cluster in a requesting phase sets its directory state to *Requested*. Suppose clusters c and c' concurrently try to instantiate the same actor. When c responds to a request from c', if c detects that its directory state is *Requested*, then c knows it has an active request. It uses a global precedence order on clusters to determine which request should win (a more sophisticated solution like [10] is not necessary because races are rare and fairness is not an issue). If $c < c'$, then the remote request has precedence, so c changes its local protocol state from *Requested* to *Loser*. This effectively cancels the request originating from c. If $c > c'$ then the local request has precedence, so c replies *Fail*, which cancels the request originating from c'. A canceled request must start over.

Optimistic activation. If responses do not arrive timely, we allow a cluster to create an instance optimistically. We use a special directory state *Doubtful* to indicate that exclusive ownership has not been established. For all *Doubtful* directory entries, the runtime periodically retries the GSI request sequence. Thus, it can detect duplicate activations eventually, and deactivate one. Optimistic activation means that duplicate instances can exist temporarily, which may be observable by the application. It is an optional feature (programmers can choose pessimistic mode instead), but we found that it usually offers the right compromise between availability and consistency: for volatile actors, the actor state need not be durable, and eventual-single-instance is usually sufficient. For persistent actors, the latest version resides in storage anyway, not in memory, so having duplicate instances in memory temporarily is fine as well.

3.1.1 Protocol Definition

Each cluster c maintains a distributed directory that maps actor keys to directory states. For each actor k, the directory assumes one of the following states:

- [*Invalid*] there is no entry for actor k in the directory.
- [*Owned*, n] c has exclusive ownership of actor k, and a local instance on node n.
- [*Doubtful*, n] c has a local instance of k on node n but has not obtained exclusive ownership.
- [*Requested*] c does not yet have a local instance of k but is currently running the protocol to obtain ownership.
- [*Loser*] c does not have a local instance of k, and its current attempt to establish ownership is being canceled.
- [*Cached*, $c' : n$] c does not have a local instance of k, but believes there is one in a remote cluster c', on node n.

Request Sending. A node starts a GSI round by setting the local directory state to *Requested* and sending requests to all clusters.

Request Processing. A cluster c receiving a request from cluster c' replies based on its directory state:

- [*Invalid*] reply (*Pass*).
- [*Owned*, n] reply (*Fail*, $c : n$).
- [*Doubtful*, n] reply (*Pass*).
- [*Requested*] if $c < c'$, set directory state to *Loser* and reply (*Pass*), else reply (*Fail*).
- [*Cached*, n] reply (*Pass*).
- [*Loser*] reply (*Pass*).

Reply Processing. A cluster c processes responses as follows (first applicable rule):

- If directory state is [*Loser*], cancel and start over.
- If one of the responses is (*Fail*, $c' : n$), transition to [*Cached*, $c' : n$].
- If there is a (*Fail*) response, cancel and start over.
- If all responses are (*Pass*), create instance on local node n and transition to [*Owned*, n].
- If some responses are missing (even after waiting a bit, and resending the request), create an instance on local node n and transition to [*Doubtful*, n].

There can be races on directory state transitions: for example, the request processing may try to transition from *Requested* to Loser at the same time as the reply processing wants to transition from *Requested* to *Owned*. In our implementation, we ensure the atomicity of transitions by using a compare-and-swap operation when changing the directory state. In addition to the transitions defined above, (1) we periodically scan the directory for *Doubtful* entries and re-run the request round for each of them, and (2) if we detect that a [*Cached*, n] entry is stale (there is no instance at node n), we start a new request round.

3.1.2 Correctness

The protocol satisfies two correctness guarantees, which we prove in the appendix.

PROPOSITION 1. *The protocol ensures that for a given actor k at most one cluster can have a directory entry for k in the Owned state, even if messages are lost.*

PROPOSITION 2. *If no messages are lost, the protocol ensures that for a given actor k at most one cluster can have a directory entry for k in either the Owned or Doubtful state.*

3.1.3 Configuration Changes

In our framework, each node n locally stores the multi-cluster configuration C_n, which is a list of clusters specified by the administrator. During configuration changes, the administrator changes the C_n non-atomically. We handle this by adding the rule:

> A node n must reply (*Fail*) to a request it receives from a cluster that is not in C_n.

This is sufficient to maintain the guarantees stated in Propositions 2 and 1, provided that for any two different configurations associated with active requests in the system, one of them is always a superset of the other. This follows from the guarantees and restrictions on configuration changes in §2.

3.2 BCAS Protocol

The batching compare-and-swap protocol implements the versioned state API for persistent actors that are kept in storage that supports some form of conditional update, such as compare-and-swap (CAS). For our current implementation, we use ETags [52].

Local read and write operations (§2.1) can be serviced directly from the cached copy and the queue of unconfirmed updates. Those operations interleave with background tasks that write pending updates to storage, read the latest version from storage, notify other instances, and process notifications from other instances. All of these background tasks are performed by a single asynchronous worker loop that starts when there is work and keeps running in the background until there is none. Such a loop ensures there is at most one access to the primary storage pending at any time. This is important to ensure correct semantics and enables batching: while one storage access is underway, all other requests are queued. Since a single storage access can service all queued requests at once, we can mask storage throughput limitations.

When an instance successfully updates storage, it sends a notification to all other instances. This helps to reduce the staleness of caches.

3.2.1 Protocol Description

Each instance stores three variables:

- *confirmed* is a tuple [version, state] representing the last known version, initially [0, new State()].
- *pending* is a queue of unconfirmed updates, initially empty.
- *inbox* is a set of notification messages containing [version, state] tuples, initially empty.

In storage, we store a tuple [version, state] that represents the latest version.

Worker Loop. The worker repeats the following steps:

1. If some tuple in *inbox* has higher version than *confirmed*, then replace the latter with the former.
2. If we have not read from storage yet, or if there are synchronize_now requests and *pending* is empty, read the latest version from storage now and update *confirmed*.
3. If *pending* is not empty, then make a deep copy of *confirmed*, apply all the updates in *pending*, and then try to write the result back to storage conditionally.
   (a) On success (version matches): update *confirmed*. Remove written updates from *pending*. Broadcast *confirmed* as a notification message.
   (b) On failure (due to a version mismatch or any other reason): re-read the current version from storage, update *confirmed*, and restart step 3.

Idempotence. The above algorithm is incorrect if a storage update request fails after updating storage, because a retry will apply the update a second time. Our solution is to add a bit-vector to the data stored in storage, with one bit per cluster that flips each time that cluster writes. When rereading after a failed write, the state of this bit tells whether the previous write failed before or after updating storage.

3.3 VLB Protocol

The volatile leader-based protocol implements the versioned state API for volatile actors. It runs a loop similar to the BCAS protocol, except that (1) the primary state is stored at one of the instances, the designated leader, and not in storage, and (2) instead of updating the state using CAS, participants send a list of updates to the leader.

Currently, we use a simple statically-determined leader, either by a consistent hash or an explicit programmer specification. In the future, we may allow leader changes as in viewstamped replication [30] or the Raft protocol [31, 37].

Orleans provides fault-tolerance of instances within a cluster. If the node containing the leader or the leader directory entry fails, a new leader instance is created. In that case, the latest version is lost, which is acceptable since durability is not required for volatile actors. Still, we have a good chance of recovering the latest version: on startup, the leader can contact all instances and use the latest version found.

4. Evaluation

We now describe our experimental results. The goal is to reveal how configuration choices and API choices influence latency and throughput for varying workloads. In particular, we are interested in the relevance of the effects provided by the three protocols (single-instancing, batching, replication).

4.1 Experimental Setup

The experiments were run on Microsoft Azure in two datacenters, located in California and the Netherlands respectively (Fig. 3). In each datacenter, 30 **front-end** (FE) servers

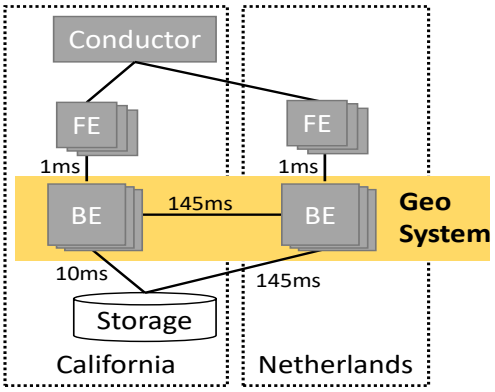

Figure 3. Setup and approximate round-trip times.

generate workload which is processed by 5 **back-end** (BE) servers that form an Orleans cluster. We vary the workload by varying the number of robots (simulated clients) that are evenly distributed over the FE from 400 to 60,000.

For the network, we use VNETs in Microsoft Azure connected by gateways. The inter-cluster round-trip (IRT) time is about 145ms. For storage, we use an Azure table storage account located in California. An access averages about 10ms from California, and about 145ms from the Netherlands.

FEs are 4-core 1.6 GHz processors with 7 GB of RAM. BEs and the conductor are 8-core 1.6 GHz pro-cessors with 14 GB of RAM. The number of front-ends is overprovisioned to ensure it is not the bottleneck.

4.1.1 Workloads

The **Byte-Array** micro-benchmark models a very simple workload where clients read and write passive actors. The actor state is a byte-array of 512B. There are two types of robots: reader/writer robots that read/update a byte sequence of 32B at a random offset. **TPC-W-mini** is a non-transactional variation of the TPC-W benchmark [50] explained in §4.5.1.

4.2 Latency

Our first series of experiments validate that our geo-distributed actor system can reduce access latencies by exploiting locality, for the byte-array workload.

We organize the results as shown in Fig. 4. The left and right half are separate sections that contain latency numbers for volatile and persistent actors, respectively. The two columns at the very left select the API and policy, which together determine the protocol (see Table1). The third column tells where the instance is cached, which matters for the single-instance policy.

4.2.1 Discussion of Volatile Section

The **first row** represents the single-instance protocol for a volatile actor cached in California. [Columns 1-2] the first access from California creates the single instance, which requires creating an Orleans actor after not finding it in the local directory (about 6ms) and running a round of the single-instance protocol (about 147ms). [Columns 2-3] repeated accesses from California hit the existing instance, and have standard Orleans actor access latency (2-3ms). [Columns 3-4] The first access from the Netherlands requires one round of the GSI protocol to detect the already existing instance in California, then another IRT to route the request to it. Ideally, this case should occur rarely. [Columns 5-6] Repeated accesses are routed directly to the instance in California, since its location has been cached, thus require only a single IRT.

The **second row** is symmetric to the first, with California and the Netherlands interchanged. The **third, fourth, sixth,** and **seventh rows** are blank because we do not currently support this combination of API and policy (easy to implement but has no benefits). The **fifth row** shows latency for linearizable operations with the VLB protocol, with the leader in California. As required by semantics, each operation incurs a round-trip to the leader (trivial from California, IRT from Netherlands). If the first access is a write; it requires two leader round-trips since our current implementation does not submit updates until after the first read. The **eighth row** shows latency for local operations (cached reads and queued writes) with the VLB protocol. These can complete without waiting for communication with a remote datacenter. Thus, latencies are roughly the same as Orleans actor creation (for the first access) and actor access latency (for repeated accesses).

4.2.2 Discussion of Persistent Section

The **first row** is largely the same as for the volatile case, except that all update operations require a storage update (+10ms to every second column). Additionally, the access that first creates the instance requires a storage read (+10ms to first two columns). The **second row** obeys the same logic as the first except that a storage roundtrip is 145ms, not 10ms (compared to volatile, +145ms to every second column, and +145ms to columns 5 and 6). The **third** and **fourth rows** represent the combination of GSI with linearizable operations. They are thus similar to the first and second row, but because they use linearizable, all reads go to storage, which can add up to another IRT. The **sixth and seventh rows** represent the combination of GSI with local operations. Thus they are very similar to the first two rows of the volatile section: latency is dominated by finding the instance, while the access itself is local to the instance. The **fifth row** represents the BCAS protocol using linearizable operations. It is similar to the volatile case, except that storage takes the role of the leader, at about the same latency in the Netherlands, but an extra 10-20ms in California. The **eighth row** again represents all-local operations, with latencies almost identical to the volatile case. Note that even if the very first access that creates an instance is a read, it does not have to wait for the

API	policy	inst. at	volatile actors: access from California: first read	first upd.	repeat read	repeat upd.	access from Netherlands: first read	first upd.	repeat read	repeat upd.	persistent actors (storage in California): access from California: first read	first upd.	repeat read	repeat upd.	access from Netherlands: first read	first upd.	repeat read	repeat upd.
Basic	single	Calif.	152.6	152.6	**2.2**	**2.1**	298.1	297.9	146.7	146.6	163.7	*173.2*	**2.1**	**13.3**	297.6	308.6	146.6	156.2
		Neth.	297.5	297.7	146.5	146.4	152.5	152.5	**2.2**	**2.2**	298.1	450.3	146.5	298.8	307.5	467.1	**2.2**	**154.1**
Versioned	single	Calif.									165.2	171.7	**9.4**	**12.1**	305.4	309.9	154.0	156.1
		Neth.									447.0	450.6	295.5	312.1	302.4	457.0	**150.9**	**154.4**
lin. ops	multi	both	6.4	6.3	**2.2**	**2.4**	157.4	306.6	**150.9**	**151.0**	15.2	25.7	**9.6**	**14.1**	156.2	312.2	**151.1**	**155.0**
Versioned	single	Calif.									152.9	152.9	**2.2**	**2.7**	298.2	298.5	146.6	147.1
		Neth.									298.2	298.2	146.5	147.0	152.9	153.2	**2.2**	**2.6**
local ops	multi	both	6.2	6.0	**2.1**	**2.3**	6.3	6.1	**2.2**	**2.4**	6.3	6.4	**2.2**	**2.6**	6.1	6.1	**2.2**	**2.6**

Figure 4. Median Access Latency in milliseconds. Cell color indicates the number of inter-cluster roundtrips (IRTs). Bold indicates the expected common case for the chosen policy (e.g. local hit for global single instance protocol).

first storage roundtrip because it can return version 0 (given by the default constructor).

4.2.3 Conclusions

Our results show that as expected, the caching layer can reduce latencies in many cases, when compared to accessing storage directly. By how much, and under what conditions, depends on the API as follows.

Single-Instance, Basic API. Both read and update latency (for volatile actors) and at least read latency (for persistent actors) are reduced to below 3ms if an actor is accessed repeatedly and at one datacenter only.

Versioned API, Linearizable Operations. Similar to the basic API in the volatile case. For the persistent case, there are no latency benefits since all operations have to access storage no matter what (by definition).

Versioned API, Local Operations. All repeated accesses at all datacenters for both volatile and persistent actors are reduced to below 3ms. All first accesses are reduced to less than 7ms. *The cost of durability and synchronization are effectively hidden.*

4.2.4 Additional Discussion

Each reported number is the median, estimated using a sample of 2000-4000 requests. We do not report the mean, because we found it an unsuitable statistic for this distribution (the long tail makes it difficult to estimate the mean with reasonable precision). In most cases, the 3rd quartile is only slightly higher than the median: less than an extra 10% for medians over 15ms, and less than an extra .3ms for medians below 3ms. But for medians between 6ms and 15ms, the 3rd quartile was significantly (20-60%) higher than the median.

Load. All latencies are for *very low load* (400 requests per second) over a period of 20s, including 5s-10s of warmup that is excluded. As the load increases, latencies increase steadily due to queueing delays. At that point, throughput is of more interest than latency, and we examine it in the next section.

4.3 Single-Actor Throughput

Our second series of experiments measure the throughput of a single actor under heavy load using the byte-array micro-benchmark. Since a well-tuned system avoids hot-spots, it is not a typical workload. Still, it offers useful insights into the behavior of our system.

4.4 Setup

To measure the peak throughput, our experiments run a series of increasing loads (by increasing the number of robots) for 20 seconds each. As the load increases, throughput increases steadily at first, then plateaus (as latency exceeds 1s, robots issue fewer requests per second). To keep the numbers meaningful and to obtain a measurable maximal throughput, we count a request towards throughput only if it returns within 1.5s. We observed a fair amount of fluctuation in peak throughput, some of which may be attributable to running on virtualized cloud infrastructure. Empirically, we can usually reproduce peak throughput within about 10%. We report all throughput numbers rounded to two significant digits.

4.4.1 Volatile Single-Actor Throughput

For the volatile case, we distinguish three configurations (Fig. 5). The *baseline* configuration places all the load on a single cluster containing the instance, while the *single* and *multi* configurations spread the load evenly over the two clusters. The *single* configuration caches a single actor instance (using the GSI protocol), while the *multi* configuration caches an instance in each cluster (using the VLB protocol). Peak throughputs for each configuration and protocol are shown in Fig. 6. We make the following observations.

For the *single* configuration, we achieve a peak throughput *within 15% of the single-datacenter baseline*. The throughput is lower because the higher latency of requests

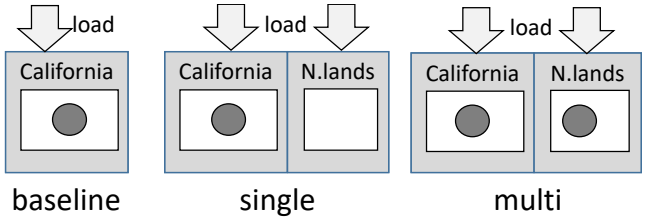

Figure 5. Volatile configurations.

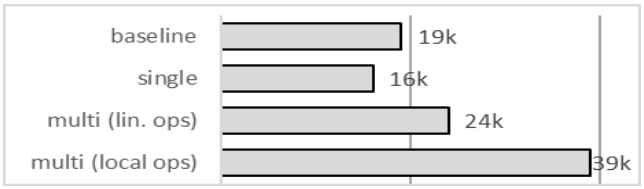

Figure 6. Volatile peak throughput.

from the Netherlands means more of them exceed the 1.5s cutoff.

Using the *multi* configuration consistently improves throughput compared to *single*:

- *Even linearizable operations perform about 50% better* (despite the strong consistency guarantee and the global serialization) because of the batching in VLB.
- Local operations have the best throughput, because reads can be served from the local cache, reducing latency and communication. We get about *double the throughput of the single-datacenter baseline*, which is as good as we can expect, considering that *multi* has exactly twice the servers of *baseline*.

4.4.2 Persistent Single-Actor Throughput

For the persistent case, we distinguish configurations {*close*, *far*, *multi*} which keep the latest version in external cloud storage as shown in Fig. 7. All place load evenly on both clusters. *close* and *far* use a single cached instance, which is close or far from the storage, respectively. *multi* uses one cached instance per cluster.

First, we examined throughput for the single-instance **Basic API**, shown in Fig. 8. We see that throughput heavily depends on the percentage of update operations.

For a workload of 100% update operations (top bar in both series), *the throughput is very low, about the reciprocal of the storage latency*. This is because with the Basic API, the actor instance cannot process any operations while an update to storage is in progress. If the workload contains only reads and no updates, throughput is o.k. because reads can be served quickly from the cache (bottom white bar in both series).

The **Versioned API** achieves much better throughput in the presence of updates because the BCAS protocol can overlap and batch read and update operations. Its peak

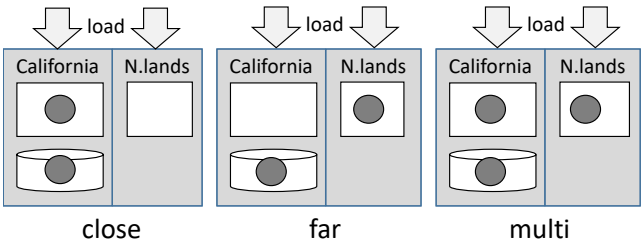

Figure 7. Persistent configurations.

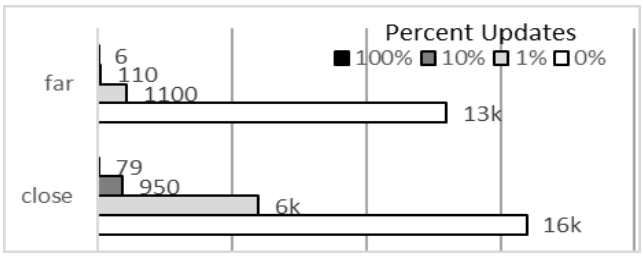

Figure 8. Persistent peak throughput, Basic API.

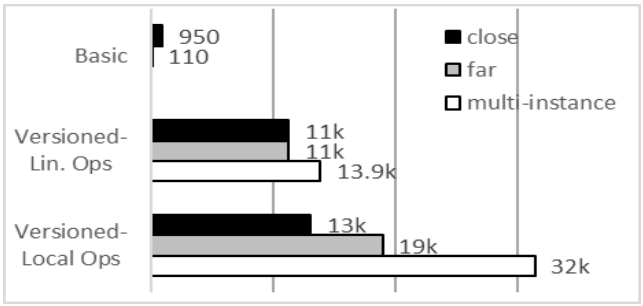

Figure 9. Persistent peak throughput, Versioned API.

throughput numbers for a 10% update rate are shown in Fig. 9.

Batching can improve peak throughput by two orders of magnitude even for linearizable operations: consider the single-instance in the *far* configuration. For the same configuration, the versioned API achieves 11k, compared to 110 for the Basic API, because a single storage access can serve many linearizable operations.

As expected, using local operations further improves the throughput (bottom series), because reads can be served at lower latency and with less communication. For single-instance, the improvement is roughly 15%-75% (*far* does better than *close* because the longer storage latency causes larger batch sizes, which saves work). For multi-instance, the performance is even better (32k). However, it does not quite reach the performance of the volatile series (39k).

4.5 Order Processing Throughput

We now study a slightly more realistic workload, which distributes over many small actors. Our results demonstrate

that the versioned API is very beneficial for the persistent case, because batching can mask storage performance limitations, and because it supports fine-grained consistency selection. Moreover, we discover that in a load-balanced system, single-instancing sometimes achieves better throughput than replication.

4.5.1 TPC-W-mini Benchmark

This benchmark models online order processing using workflows. It is inspired by the TPC-W benchmark, but makes simplifications to work around the lack of transactions and models only a subset of the transactions.

We use two classes of actors: (1) an *Item* actor represent a single item of inventory. It has a stock-quantity, a price, and a list of reservations (each consisting of a cart id and a multiplicity). (2) a *Cart* actor tracks the ordering progress of a customer. It contains workflow status and a list of items in the cart. There is one cart per robot.

Each robot goes through four workflow steps, corresponding to operations on the cart actors:

1. *create* — create a new workflow, starting with an empty user cart
2. *add items* — add a number of items to the cart, and validate their existence by calling exists on the item actors
3. *buy* — for each item in the cart, reserve the requested quantity by calling *reserve(cart-id, quantity)* on each item, and add up the returned price for all items
4. *buy-confirm* — finalize the purchase by calling *confirm(cart-id)* on each item.

Robots pause for a configurable thinking time between steps. They issue at most one new workflow every 4 seconds, which limits the request rate to an average of 1 request per second per robot.

The reservations in step *buy* are allowed to be optimistic (an item can be reserved without fully guaranteeing that the stock-quantity is sufficient); but in step *buy-confirm*, the reservation must be checked against the actual quantity available. If either of these steps fails, the workflow is aborted, and its reservations are undone.

Configuration Variations. Cart actors are always single-instance and volatile. For the item actors, we implemented two options (*basic*/*versioned*), using the basic and versioned API respectively, and we tried both options *persistent*/*volatile* for each. For the versioned case, we have an extra option (*lin*/*mixed*) where *lin* uses linearizable operations to read and update items (thus always working with the very latest inventory), while *mixed* uses local operations whenever possible without risking to oversell items, i.e. everywhere except in *confirm(cart-id)*.

Setup. The load and servers are evenly distributed over the two datacenters as in Fig. 3. We use the TPC-W scale setting of 1000 items, with one item per order and a thinking time of 100ms. Throughput is the number of workflow steps that complete in less than 1.5s, divided by the test duration (28s), rounded to two significant digits.

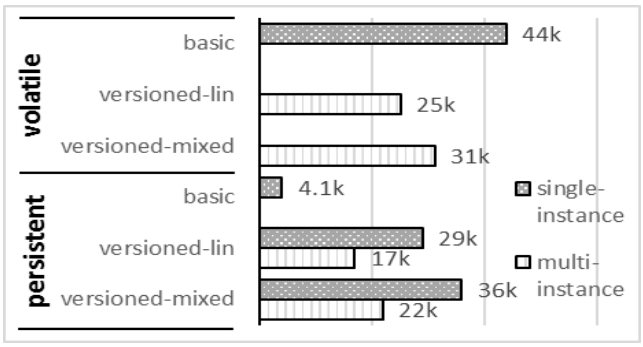

Figure 10.

4.5.2 Results

We show peak throughput results in Fig. 10, and make the following observations:

Single-instance Batching. For the persistent case, the basic API again performs poorly (4.1k) because it can process only one operation at a time. In comparison, with the Versioned API, *batching all reads and updates at a single instance improves throughput by a factor 7x*, to 29k. This is remarkable, especially considering that all operations remain linearizable with respect to external storage.

Mixed Consistency. Using strong consistency only where actually needed (i.e. during the confirm phase) provides an appreciable additional throughput improvement (about 24-30%). This confirms the benefit of an API that allows to adjust the consistency level not only per actor, but per individual operation.

To replicate or not to replicate. An interesting (and somewhat unexpected) result is that multi-instance exhibits lower throughput than single-instance. The reason is that coordinating the instances requires more overall work (notification messages, retries on conflicts) which reduces peak system throughput. This is in stark contrast to the results for single-actor throughput (Figs. 6,9), because for the latter, the extra work is performed by otherwise idle nodes. However, in a load balanced situation where all servers are highly utilized, extra work directly reduces global throughput.

5. Related Work

We do not know of prior work on geo-distributed actor systems, and more generally, of work that strongly separates the geo-distribution from durability. However, GEO's mechanisms touch on many aspects of distributed computing, which we summarize here: distributed objects, caching, geo-distributed transactions, multi-master replication, replicated data types, and consensus.

Distributed object systems from the 1980s and 90s are very similar to virtual actor systems, and share their ability to scale-out [2, 11, 14]. However, hardly any focused on stateful applications, on geo-distribution, or consider situations where the object state is persisted externally. One exception is Thor [22], but it used a highly-customized object-server with server-attached storage and thus lacked the flexibility of GEO. Of course, at the time the now common and cheaply available separation of compute and storage in the cloud did not exist.

Group communication (e.g., Isis [4] and Horus [51]) and distributed shared memory systems (e.g., Interweave [46]) offer protocols for coherent replication. However, they do not offer the abstraction level of our virtual actor API, with its choice of volatile, externally persisted, single-instance, and multi-instance configurations, and with optimized coherence protocols for each case; nor do they provide a state API that offers an easy choice between eventual consistency and linearizability, and permits reasoning in terms of latest, confirmed, and tentative versions.

Actors that are persisted can be thought of as an object-oriented cache. Two popular cache managers for datacenters are Redis and memcached. In Redis, writes are processed by a primary cache server and can be replicated asynchronously to read-only replicas [38]. In memcached, cache entries are key-value pairs, which are spread over multiple servers using a distributed hash table [26]. Neither system is object-oriented and neither offers built-in support for geo-distribution.

There has been a steady stream of papers in recent years on systems to support transactions over geo-distributed storage, using classical ACID semantics [1, 13], weaker isolation for better performance [21, 24, 45], and optimized geo-distributed commit protocols [19, 27, 28]. Unlike GEO, they do not provide an actor model with user-defined operations, nor do they separate geo-distribution from durability.

The same distinctions apply to multi-master replication, which has a rich literature of algorithms and programming models [17, 40, 47, 48]. Most of them focus on conflict detection, typically using vector clocks. Vector clocks are not needed by the BCAS and VLB protocols since they serialize updates at a primary copy.

There are many approaches that, like GEO, distinguish between fast, possibly-inconsistent operations vs. slow consistent operations [12, 14, 18, 21, 41]. Bayou [49] was among the first and is especially similar to GEO. In Bayou, updates are procedures, though they are in a stylized format having a conflict check and merge function. In both systems, an object/actor state is its sequence of committed updates, followed by the tentative ones. Unlike GEO, Bayou makes tentative updates visible before they commit, and their ordering may change until they commit. Like GEO, Bayous implementation uses a primary copy to determine which updates have committed.

Some geo-replication mechanisms avoid agreement on a global update sequence to improve performance and availability [6, 21, 43, 44]. However, these systems can be difficult to use for developers, because they do not guarantee the existence of a latest version, and do not support arbitrary user-defined operations on objects or actors. Rather, each abstract data type requires a custom distributed algorithm as in CRDTs [6, 43, 44], or commutativity annotations on operations as in RedBlue consistency [21]. Also, linearizable operations are usually not supported [21].

Some key-value stores offer some of GEO's functionality, but only for their custom storage and not for objects. For example, PNUTS [12] supports linearizable operations on a primary copy of each record, which can migrate between datacenters. Its operations offer consistency-speed tradeoffs.

Cassandra offers eventually consistent, highly-scalable geo-distributed storage using quorum consensus for both reads and writes [9]. Updates are timestamped writes or deletes and are applied to all replicas. They replace the current state unconditionally. Thus, clients cannot atomically read-modify-write object state as in GEO or Bayou.

Cassandra programmers can configure how many replicas to maintain where, and indicate how many replicas to read or update on any given operation, which lets them control tradeoffs similar to our configuration choices. However, it is difficult to extract high-level semantic consistency guarantees (as in our versioned state API) from these low-level numeric parameters; there is no concept of a primary copy or global version numbers, or support for linearizable operations.

As a workaround, recent versions of Cassandra offer lightweight transactions which run a multi-phase Paxos [20] protocol. By contrast, consensus in GEO is either fast or delegated: For volatile objects, GEO runs a simple single-phase leader-based consensus. The leader is determined by a mutual exclusion protocol (GSI) for the single-instance case, or statically for the multi-instance case (VLB). For persistent objects, the BCAS protocol sequences and batches updates at one or more instances, and when committing the batches, delegates the final consensus to the storage layer.

6. Conclusion

This paper introduced GEO, an actor system for implementing geo-distributed services in the cloud. Its virtual actor model separates geo-distribution from durability, and supports both volatile and externally persisted actors. It can exploit locality where present (single-instance configuration), and supports replication where necessary (multi-instance configuration). The complexity of protocols to support these options is hidden underneath GEOs simple linearizable API, which puts geo-distributed applications within the reach of mainstream developers. GEOs implementation includes three such protocols for distributed coherence that tolerate failures and configuration changes. Our evaluation of latency

and throughput demonstrates that the model permits some very effective optimizations at the protocol layer, such as replication (for reducing global latency) and batching (for hiding performance limitations of the storage layer).

Availability. The GEO project is open-source and a pre-release is publicly available on GitHub [19]. It is already being used in a commercial setting by an early adopter, a company operating a service with clusters in several continents.

Future Work. We would like to investigate more protocols and protocol variations, for example to support storage systems with different synchronization primitives. This may require factoring out duplicate functionality in each layer. A design framework could be developed to help choose among such storage systems for a given workload. Another interesting challenge is the design and implementation of a mechanism for geo-distributed transactions on actors. Finally, one could develop adaptive protocols that switch between configurations automatically.

References

[1] J. Baker, C. Bond, J. C. Corbett, J. Furman, A. Khorlin, J. Larson, J.-M. Leon, Y. Li, A. Lloyd, and V. Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *Conference on Innovative Data System Research (CIDR)*, pages 223–234, 2011.

[2] H. E. Bal, M. F. Kaashoek, and A. S. Tanenbaum. Orca: a language for parallel programming of distributed systems. *IEEE Transactions on Software Engineering*, 18(3):190–205, Mar 1992.

[3] P. Bernstein, S. Bykov, A. Geller, G. Kliot, and J. Thelin. Orleans: Distributed virtual actors for programmability and scalability. Technical Report MSR-TR-2014-41, Microsoft Research, March 2014.

[4] K. P. Birman and R. V. Renesse. *Reliable Distributed Computing with the ISIS Toolkit*. IEEE Computer Society Press, 1993.

[5] S. Burckhardt. Principles of eventual consistency. *Found. Trends Program. Lang.*, 1(1-2):1–150, Oct. 2014.

[6] S. Burckhardt, A. Gotsman, H. Yang, and M. Zawirski. Replicated data types: Specification, verification, optimality. In *Principles of Programming Languages (POPL)*, 2014.

[7] S. Burckhardt, D. Leijen, J. Protzenko, and M. Fähndrich. Global sequence protocol: A robust abstraction for replicated shared state. In *European Conference on Object-Oriented Programming (ECOOP)*, 2015.

[8] S. Bykov, A. Geller, G. Kliot, J. Larus, R. Pandya, and J. Thelin. Orleans: Cloud computing for everyone. In *Symposium on Cloud Computing (SoCC)*, pages 16:1–16:14, 2011.

[9] The Apache Cassandra Project. `http://cassandra.apache.org`.

[10] K. M. Chandy and J. Misra. The drinking philosophers problem. *ACM Trans. Program. Lang. Syst.*, 6(4):632–646, Oct. 1984.

[11] J. Chase, F. Amador, E. Lazowska, H. Levy, and R. Littlefield. The amber system: Parallel programming on a network of multiprocessors. In *Symposium on Operating Systems Principles (SOSP)*, pages 147–158, 1989.

[12] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. Pnuts: Yahoo!'s hosted data serving platform. *Proc. VLDB Endow.*, 1(2):1277–1288, Aug. 2008.

[13] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. 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, and D. Woodford. Spanner: Google's globally distributed database. *ACM Trans. Comput. Syst.*, 31(3):8:1–8:22, Aug. 2013.

[14] A. Fekete, D. Gupta, V. Luchangco, N. Lynch, and A. Shvartsman. Eventually-serializable data services. In *Principles of Distributed Computing (PODC)*, pages 300–309, 1996.

[15] GEO system prototype. Available as a branch forked from the Orleans github project, at `https://github.com/sebastianburckhardt/orleans/tree/geo-samples`, 2016.

[16] M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM TOPLAS*, 12, 1990.

[17] B. Kemme, R. Jiménez-Peris, and M. Patiño-Martínez. *Database Replication*. Synthesis lectures on data management. Morgan & Claypool Publishers, 2010.

[18] T. Kraska, M. Hentschel, G. Alonso, and D. Kossmann. Consistency rationing in the cloud: Pay only when it matters. *Proc. VLDB Endow.*, 2(1):253–264, Aug. 2009.

[19] T. Kraska, G. Pang, M. J. Franklin, S. Madden, and A. Fekete. Mdcc: Multi-data center consistency. In *European Conference on Computer Systems (EuroSys)*, pages 113–126, 2013.

[20] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16:133–169, 1998.

[21] C. Li, D. Porto, A. Clement, R. Rodrigues, N. Preguiça, and J. Gehrke. Making geo-replicated systems fast if possible, consistent when necessary. In *Operating Systems Design and Implementation (OSDI)*, 2012.

[22] B. Liskov, M. Castro, L. Shrira, and A. Adya. Providing persistent objects in distributed systems. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 230–257, 1999.

[23] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Don't settle for eventual: scalable causal consistency for wide-area storage with COPS. In *Symposium on Operating Systems Principles (SOSP)*, 2011.

[24] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Stronger Semantics for Low-Latency Geo-Replicated Storage. In *Networked Systems Design and Implementation (NSDI)*, 2013.

[25] H. Melgratti and C. Roldán. A formal analysis of the global sequence protocol. In *COORDINATION 2016*, pages 175–191. Springer, 2016.

[26] Memcached. Available under BSD 3-clause license. https://github.com/memcached/memcached, 2016.

[27] F. Nawab, D. Agrawal, and A. E. Abbadi. Message futures: Fast commitment of transactions in multi-datacenter environments. In *Conference on Innovative Data System Research (CIDR)*, 2013.

[28] F. Nawab, V. Arora, D. Agrawal, and A. El Abbadi. Minimizing commit latency of transactions in geo-replicated data stores. In *International Conference on Management of Data (SIGMOD)*, pages 1279–1294, 2015.

[29] The netflix simian army. http://techblog.netflix.com/2011/07/netflix-simian-army.html, Sept. 2011.

[30] B. M. Oki and B. H. Liskov. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *Principles of Distributed Computing (PODC)*, pages 8–17, 1988.

[31] D. Ongaro and J. Ousterhout. In search of an understandable consensus algorithm. In *USENIX ATC'14: USENIX Annual Technical Conference*, pages 305–320, 2014.

[32] Orbit - virtual actors for the jvm. BSD 3-clause license. https://github.com/orbit/orbit, 2016.

[33] MIT license. https://github.com/dotnet/orleans, 2016.

[34] The year in downtime: The top 10 outages of 2013. http://www.datacenterknowledge.com/archives/2013/12/16/year-downtime-top-10-outages-2013/.

[35] Ponemon institute: 2013 study on data center outages. http://www.emersonnetworkpower.com/documentation/en-us/brands/liebert/documents/white%20papers/2013_emerson_data_center_outages_sl-24679.pdf.

[36] R. Power and J. Li. Piccolo: Building fast, distributed programs with partitioned tables. In *Operating Systems Design and Implementation (OSDI)*, pages 293–306. USENIX Association, 2010.

[37] The raft consensus algorithm. https://raft.github.io/, 2016.

[38] Redis. http://redis.io/documentation/, 2016.

[39] J. B. Rothnie and N. Goodman. A survey of research and development in distributed database management. In *International Conference on Very Large Data Bases (VLDB)*, pages 48–62, 1977.

[40] Y. Saito and M. Shapiro. Optimistic replication. *ACM Computing Surveys*, 37:42–81, 2005.

[41] M. Serafini, D. Dobre, M. Majuntke, P. Bokor, and N. Suri. Eventually linearizable shared objects. In *Principles of Distributed Computing (PODC)*, pages 95–104, 2010.

[42] Service fabric reliable actors. Available for the Windows Azure platform, see https://azure.microsoft.com/en-us/documentation/articles/service-fabric-reliable-actors-get-started/, 2016.

[43] M. Shapiro, N. Preguica, C. Baquero, and M. Zawirski. A comprehensive study of convergent and commutative replicated data types. Technical Report Rapport de recherche 7506, INRIA, 2011.

[44] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. Conflict-free replicated data types. In *13th Int. Symp. on Stabilization, Safety, and Security of Distributed Systems (SSS)*, Grenoble, France, Oct. 2011.

[45] Y. Sovran, R. Power, M. K. Aguilera, and J. Li. Transactional storage for geo-replicated systems. In *Symposium on Operating Systems Principles (SOSP)*, 2011.

[46] C. Tang, D. Chen, S. Dwarkadas, and M. L. Scott. Efficient distributed shared state for heterogeneous machine architectures. In *International Conference on Distributed Computing Systems (ICDCS)*, pages 560–, 2003.

[47] D. Terry. Replicated data consistency explained through baseball. *Commun. ACM*, 56(12):82–89, Dec. 2013.

[48] D. B. Terry. *Replicated Data Management for Mobile Computing*. Synthesis Lectures on Mobile and Pervasive Computing. Morgan & Claypool, May 2008.

[49] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *Symposium on Operating Systems Principles (SOSP)*, 1995.

[50] Tpc-w. http://www.tpc.org/tpcw/tpc-w_wh.pdf.

[51] R. van Renesse, K. P. Birman, and S. Maffeis. Horus: A flexible group communication system. *Commun. ACM*, 39(4):76–83, Apr. 1996.

[52] W3C. http://www.w3.org/1999/04/Editing/.

[53] Windows azure cache. http://www.windowsazure.com/en-us/documentation/services/cache, 2016.

A. GSI Correctness Proof

The goal of the GSI protocol, as defined in §3.1.1, is to disallow two instantiations of the same actor. We prove this in two failure models. In the first model, we assume that messages can be lost. Thus, if a cluster c does not receive a message that it expects from a sender d within a timeout period, then c must assume that d is simply slow or unable to communicate with c (but may be able to communicate with other clusters). In the second model, there are no communication failures.

A.1 With lost messages

PROPOSITION 1. *The protocol ensures that for a given actor k at most one cluster can have a directory entry for k in the Owned state, even if messages are lost.*

PROOF. Suppose two clusters, c and d, have such a directory entry. To arrive in that state, each of them must have executed the GSI request sequence at some point and moved to the *Owned* state in the final step, when processing the responses. We want to show that this is impossible, which proves the proposition.

We do not know exactly how the steps of the two competing requests interleaved; however, we can reason our way through several distinct cases and eventually derive a contradiction, which proves the proposition. First, consider the following table which labels the steps of the protocol in each cluster:

Cluster c	Cluster d
c_1. Send Request	d_1. Send Request
c_2. Wait for Replies	d_2. Wait for Replies
c_3. Process all replies and update state to *Owned*	d_3. Process all replies and update state to *Owned*

We will now show that this table is not consistent with any ordering of the events, via a case analysis. The four cases are based on when and whether d received the request sent by step c_1.

1. Suppose d received the request from c_1 before d_1. There are three sub-cases, depending on when and whether c received ds request from d_1.
   - Suppose c received ds request from d_1 after c_3. Then c replied $(\textit{Fail}, c)$ to d, and d does not move to state *Owned* in step d_3 (because it either saw this response, or no response at all), which contradicts our assumption.
   - Suppose c received ds request from d_1 before c_3. This must have happened after c_1 (because we assumed that c_1 happened before d_1). Therefore, c was in state [*Requested*]. If $c < d$, then c updated its state to [*Loser*]. Therefore, when c processed replies to its request in c_3, it would not set its state to *Owned*, contradicting our assumption. If $c > d$, then it replied (*Fail*) to d, in which case d in step d_3 would not set its state to *Owned* (because it would either see that response or no response at all), contradicting our assumption.
   - Suppose c did not receive ds request from d_1 at all. Then it would not move to state *Owned* in c_3, contradicting our assumption.
2. Suppose d received the request from c_1 after d_1 but before d_3. Thus, d was in state [*Requested*] when it received that request. There are two sub-cases:
   - If $c < d$, then d replied to c with (*Fail*), in which case c does not move to state *Owned* in c_3 (because it either saw this response, or no response at all), contradicting our assumption.
   - If $c > d$, then d replied (*Pass*) and set its state to [*Loser*]. Therefore, d does not move to state *Owned* in step d_3, contradicting our assumption.
3. Suppose d received the request from c_1 after d_3. Since d was in the *Owned* state at that point it must have replied (*Fail*) to c's request. Therefore, c does not move to state *Owned* in step c_3 (because it either saw this response, or no response at all), contradicting our assumption.
4. Finally, suppose d did not receive the request from c_1. Then it cannot have replied, and c does not move to state *Owned* in step c_3, contradicting our assumption.

This concludes the proof of Proposition 1. □

A.2 Without lost messages

If communication failures do not occur, the protocol makes the stronger guarantee that at most one cluster is in Owned or Doubtful state, which means at most one cluster has an instance active.

PROPOSITION 2. *If no messages are lost, the protocol ensures that for a given actor k at most one cluster can have a directory entry for k in either the Owned or Doubtful state.*

PROOF. with no lost messages, the reply processing never moves a directory entry to the *Doubtful* state. Thus, the claim follows directly from Proposition 1. □