Geo-Distribution of Actor-Based Services

PHILIP A. BERNSTEIN, Microsoft Research, USA
SEBASTIAN BURCKHARDT, Microsoft Research, USA
SERGEY BYKOV, Microsoft, USA
NATACHA CROOKS, University of Texas, Austin, USA
JOSE M. FALEIRO, Yale University, USA
GABRIEL KLIOT, Google, USA
ALOK KUMBHARE, Microsoft Research, USA
MUNTASIR RAIHAN RAHMAN, Microsoft, USA
VIVEK SHAH, University of Copenhagen, Denmark
ADRIANA SZEKERES, University of Washington, USA
JORGEN THELIN, Microsoft Research, USA

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. Caching and replication are essential to hide latency and exploit locality; but it is not a priori clear how to combine these techniques with the actor programming model.

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's programming model 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.

CCS Concepts: • **Computer systems organization** → **Cloud computing**; *Dependable and fault-tolerant systems and networks*; • **Software and its engineering** → **Message oriented middleware**; *Organizing principles for web applications*; • **Computing methodologies** → *Distributed computing methodologies*; • **Theory of computation** → Distributed algorithms;

Additional Key Words and Phrases: Virtual Actors, Cloud Services, Geo-Distribution, Consistency

ACM Reference Format:
Philip A. Bernstein, Sebastian Burckhardt, Sergey Bykov, Natacha Crooks, Jose M. Faleiro, Gabriel Kliot, Alok Kumbhare, Muntasir Raihan Rahman, Vivek Shah, Adriana Szekeres, and Jorgen Thelin. 2017. Geo-Distribution of Actor-Based Services. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 107 (October 2017), 26 pages.
https://doi.org/10.1145/3133931

2475-1421/2017/10-ART107
https://doi.org/10.1145/3133931

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 [Akka 2016; Orbit 2016; Orleans 2016; SF Reliable Actors 2016]. 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 line of actor frameworks [Orbit 2016; Orleans 2016; SF Reliable Actors 2016] have adopted the *virtual actor model* [Bernstein et al. 2014; Bykov et al. 2011] which automates all of these aspects.

For each class of actors, the developer specifies only (1) a unique key for identifying instances, (2) the actor state, and (3) the supported operations. An operation can read and update the actor's state, and can call other actors' operations (see Fig. 1 for an example). Actors cannot share state: operation arguments and return values are deeply copied. Note that actor operations are not just one-way messages as in the pure actor model, but can return a value or throw an exception.

The class and key are sufficient to identify a virtual actor instance and call its operations. Actor instances are automatically activated when first used (i.e., when an operation is invoked), and deactivated when unused for some period of time. This is a significant difference to the standard actor model, where actors are created explicitly by the application, and are garbage-collected once unreachable.

For virtual actors, programmers can specify how to save and load the actor state to/from external storage, if persistence is desired. Thus, the runtime can distribute, activate and deactivate actors based on use, effectively resembling a distributed cache [Memcached 2016; Power and Li 2010; Redis 2016; Windows Azure Cache 2016].

Geo-Distribution Challenge. Today's 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.* Although round-trip latency between servers is under 2ms within a datacenter, it can be two orders of magnitude higher between distant datacenters, e.g., about 150ms between California and the Netherlands. Thus, a naive reuse of single-datacenter application architectures, programming models, and communication protocols 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 actor is accessed mostly from a single datacenter, those accesses should not incur any geo-remote calls. Moreover, a solution should support *replication where appropriate.* 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 variation of the virtual actor programming model.

Geo's 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, and gracefully handle node failures, network partitions, and

```
actor ChatRoom[guid: string]
{
  state messages: List⟨string⟩;
  op Read(): List⟨string⟩ { return messages; }
  op Post(userid: string, msg: string): boolean
  {
    if (!await User[userid].MayPost()) { // calls User actor
      return false;
    } else {
      messages.Add(msg);
      return true;
    }
  }
}
```

```
actor User[userid: string]
{
  state banned;

  // called by Post operation
  op MayPost(): boolean {
    return !banned;
  }

  op Ban()
  {
    banned = true;
  }
}
```

Fig. 1. Basic pseudo-code illustration of the virtual actor programming model, on a *chat service* example. The actor classes on the left and right represent chat rooms and users, respectively. Each actor class defines a key (first line), state (third line), and operations. The Post operation on the ChatRoom actor first calls the User actor for the user identified by userid (wherever it may be located) to check if this user may post a message.

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 optional: all actors support globally consistent, *linearizable* reads and writes, which are guaranteed to read or write the latest version.

1.1 Actor Configuration in Geo

Clusters and Multi-Clusters. We define a *cluster* to be a set of servers, called *nodes*, connected by a high-speed network. Clusters are elastic and robust: while the service is operating, nodes can be added or removed, and node failures are automatically detected and tolerated. Similarly, several clusters can be joined into a multi-cluster; and clusters can be added or removed from the multi-cluster. A single datacenter may contain multiple clusters, e.g., to group nodes into distinct failure zones that operate and fail independently.

Configuration Options. To perform better across a wide range of requirements, Geo supports several configuration options for actors in a multi-cluster as shown in Fig. 2. 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 node of one cluster) or *multi-instance* (state is cached in one node of 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.

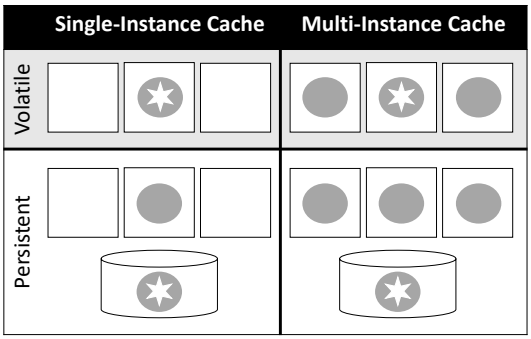

Fig. 2. The four actor configuration options in a multi-cluster. Squares are clusters, Cylinders are the storage layer, circles are copies of the actor state, and the star marks the latest version (primary copy).

Activation on First Access. In Geo, single-instance actors are activated only in the cluster where they are first accessed, while multi-instance actors are activated in all clusters when first accessed.

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 Bob's 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.

1.2 Novelty and Relevance

Prior work on geo-distributed services has heavily focused on the challenge of providing geo-replicated storage [Cassandra 2016; Corbett et al. 2013; Li et al. 2012; Lloyd et al. 2011; Rothnie and Goodman 1977; Sovran et al. 2011], usually using quorum-based algorithms. A distinguishing feature of our actor-based approach is that it **separates geo-distribution from durability**, which is highly relevant for actor-based services:

(1) Providing durability for volatile actors is meaningless and 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. Also, actors may be used for synchronization purposes, rather than to reliably store data.

(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.

Internally, the storage system typically uses quorum-based replication, possibly within a single datacenter, or possibly geo-distributed. Geo is agnostic to such details. The only requirement at this time is that the storage supports strongly consistent reads and conditional updates; future implementations may relax this further.

1.3 Contributions

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

- Geo's **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.
- Geo's **implementation** is open-source. It is part of the Orleans distribution [Orleans 2016], and is 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

Geo extends the C# Orleans virtual actor framework [Bernstein et al. 2014; Bykov et al. 2011; Orleans 2016] by extending its APIs to support the multi-cluster configuration options in Fig. 2. These options are offered under two APIs:

The **basic state API** (§2.1) is simply the already-existing Orleans actor API. It supports the single-instance configuration for volatile and persistent actors (left half of Fig. 2). It is simple to use and efficient for volatile grains, since synchronous reads and writes on main memory run fast. However, it does not support multi-instance configurations, and it can suffer from performance problems on write-hot persistent grains.

The **versioned state API** (§2.2) is a bit more involved, but supports all four configurations in Fig. 2 under a single API and consistency model, and can batch updates on write-hot grains. For each read or update operation, the developer can prioritize consistency (which guarantees linearizability) or speed (which guarantees an immediate response), as proposed and formalized by the GSP (global sequence protocol) consistency model [Burckhardt et al. 2015; Gotsman and Burckhardt 2017].

2.1 Basic State API in Orleans

Fig. 3 shows an example of how to define Orleans actors for the example from Fig.1.

Grain interfaces are defined as C# interfaces (Fig. 3a). These define the publicly visible operations of a grain, and select the key type by deriving from a corresponding framework interface. Here, IGrainWithStringKey and IGrainWithGuidKey select string and Guid as the key type for the user and chat-room grains, respectively.

Volatile grains are defined by C# objects that derive from the framework class Grain and the grain interface (Fig. 3b). Grain state and grain operations simply map to object state and object methods. Note that the result of the Post method, which is of type Task⟨bool⟩, is effectively a promise, automatically constructed by the C# compiler as a continuation following the await keyword. It returns true or false depending on the outcome of the conditional.

```
public interface IUser : IGrainWithStringKey
{
    Task⟨bool⟩ MayPost();
}
public interface IChatRoom : IGrainWithGuidKey
{
    Task⟨bool⟩ Post(string user, string msg);
    Task⟨List⟨string⟩⟩ Read();
}
```

```
public class ChatRoom : Orleans.Grain, IChatRoom
{
    private List⟨string⟩ Messages;
    public async Task⟨List⟨string⟩⟩ Read() { return Messages; }
    public async Task⟨bool⟩ Post(string user, string msg) {
        if (await GrainFactory.GetGrain⟨IUser⟩(user).MayPost()) // call grain and await result
        {
            Messages.Add(msg);
            return true;
        } else {
            return false;
        }
    }
}
```

```
[Serializable]
public class ChatState // defines the chatroom state
{
    public List<string> Messages = new List<string>();
}

[StorageProvider(ProviderName = "whatStorageToUse")]
public class ChatRoom : Orleans.Grain⟨ChatState⟩, IChatRoom
{
    public async Task⟨List⟨string⟩⟩ Read() { return State.Messages; }
    public async Task⟨bool⟩ Post(string user, string msg)
    {
        if (await GrainFactory.GetGrain⟨IUser⟩(user).MayPost()) // call grain and await result
        {
            State.Messages.Add(msg);
            await WriteStateAsync(); // persist state, await I/O completion (storage ack)
            return true;
        } else {
            return false;
        }
    }
}
```

Fig. 3. Example that illustrates the existing Orleans C# API for defining volatile and persistent actors, called *grains*. **(a)** top: common interface definitions. **(b)** middle: a volatile chat-room grain implementation. **(c)** bottom: a persistent chat-room grain implementation.

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

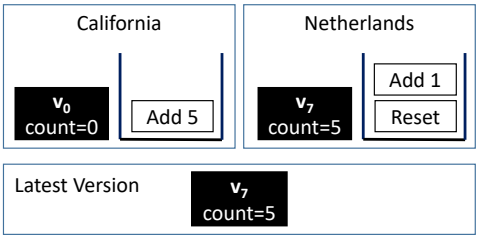

Fig. 4. (a) **left**: example definition of state and update objects. (b) **right**: sample snapshot of the internal state of an actor that uses the versioned state API and has two instances in different datacenters.

Persistent grains are defined as shown in Fig. 3c. The state is separated out into a serializable class ChatState. The ChatRoom grain then derives from the framework class Grain⟨ChatState⟩ passing ChatState as a type parameter. The grain state is accessible to grain operations via the State property (implemented by the base class); when modifying the state, operations call WriteStateAsync (implemented by the base class) to write the changes to storage. How to persist the state can be configured by selecting a *storage provider* via a *class attribute*, a general C# mechanism for annotating meta data that uses a bracket syntax. Here, the attribute [StorageProvider(ProviderName = "whatStorageToUse")] that immediately precedes the ChatRoom class definition means that a storage provider with name "whatStorageToUse" should be used to persist the ChatRoom grains. A storage provider with the specified name must be configured on startup. Application developers can write their own storage providers or use already-existing providers for common cloud storage systems.

2.2 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 shown in Fig. 4a.

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. 4b 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.

read/write operations	
read_tentative	() → State
read_confirmed	() → (State, int)
enqueue	Update → ()
synchronization primitives	
confirm_updates	() → task
refresh_now	() → task

Fig. 5. Signature of the Versioned State API.

Background Propagation. At all times, the consistency protocol runs in the background on each instance of an actor to propagate updates. It applies each queue's 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. 2. 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. It is often acceptable to work with a somewhat stale actor state and to delay the confirmation of updates [Cassandra 2016; Kraska et al. 2009; Terry 2013]. 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 the configuration in Fig. 4b, 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. 4b, 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 [2005] benchmark, 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 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* [Herlihy and Wing 1990]: 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 (Fig. 5) is a variation of the global sequence protocol (GSP) operational consistency model [Burckhardt et al. 2015; Gotsman and Burckhardt 2017; Melgratti and Roldán 2016], 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 [Terry 2008, 2013] as formalized by [Burckhardt 2014], 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, the principal advantage of actor systems. Though it may complicate life for developers, they have not raised it 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 as actors ensures causality of the chat content. Also, in cases where an application has to coordinate updates to multiple actors, such as in the order-processing mini-benchmark in §4.4.1, it can implement a "workflow actor" to track which updates have been applied.

Linearizability vs. Sequential Consistency. If the user asks for linearizability, we guarantee true linearizability (including the real-time aspect — updates are applied to the persistent storage in real time in between call and return). It is not cheap, but sometimes it is the right thing (our experiments in §4 show exactly how expensive it is). Offering true linearizability as an option, where needed, is convenient and practical. We do not think that Sequential Consistency (SC) is a desirable consistency choice to offer, as it is neither strong enough to be undisputable and easy to explain (clients can easily communicate out of band and thus observe real time), nor weak enough to provide any significant performance benefits (unlike causally consistent GSP).

2.3 Versioned State API in Orleans

Fig. 6 shows a brief example of how to use the Versioned State API in Orleans to express the pseudo-code example from Fig. 1, at the time of writing. For the latest API syntax, a more thorough discussion of features, and additional examples, see the online documentation [Orleans 2016].

The classes that define state and update objects are the class ChatState (line 2) and the class MessagePostedEvent (line 9), respectively. The state class defines an Apply method (line 5) to specify the effect of the update on the state. Both classes are marked with the Serializable attribute (lines 1, 8) since the runtime needs to serialize/deserialize these objects to send them in messages and persist them in storage.

```
1  [Serializable]
2  public class ChatState
3  {
4      public List⟨string⟩ Messages = new List⟨string⟩();
5      public void Apply(MessagePostedEvent e) { Messages.Add(e.Content); }
6  }
7
8  [Serializable]
9  public class MessagePostedEvent()
10 {
11     public string Content;
12 }
13
14 [GlobalSingleInstance] // or [OneInstancePerCluster] for multi−instance
15 [LogConsistencyProvider(ProviderName = "whatProtocolToUse")]
16 [StorageProvider(ProviderName = "whatStorageToUse")]
17 public class ChatGrain : JournaledGrain⟨VersionedChatState⟩, IChatRoom
18 {
19     public async Task⟨List⟨string⟩⟩ Read()
20     {
21         return TentativeState.Messages;
22     }
23     public async Task⟨bool⟩ Post(string user, string msg)
24     {
25         var usergrain = GrainFactory.GetGrain⟨IUser⟩(user);
26         var allowed = await usergrain.MayPost();
27         if (allowed) {
28             EnqueueEvent(new MessagePostedEvent() { Content = msg });
29             return true;
30         } else {
31             return false;
32         }
33     }
34 }
```

Fig. 6. Example that illustrates how to use the versioned-state API from within the C# Orleans framework. Both the actor state and the actor updates are defined as classes (top). The grain implementation class (bottom) inherits from JournaledGrain which provides the versioned state API as described in §2.2.

The grain implementation is defined by the class ChatGrain (line 17). It derives from the framework class JournaledGrain, which implements the versioned API (Fig. 5) with some minor differences:

- the function read_tentative is exposed as a property TentativeState (used on line 21).
- the function enqueue is named EnqueueEvent (used on line 28).

When the grain operation Post (line 23) is called on a ChatGrain instance, the code first constructs a reference to the user grain for the user who is posting (line 25). It then calls the MayPost method on that grain and waits for the result (line 26). If the user is allowed to post, a new MessagePostedEvent object is constructed and enqueued (line 28). As written, the Post operation always returns immediately, without waiting for any I/O operations to complete. For stronger consistency guarantees, one could use linearizable operations as described in §2.2.

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

API		Configuration		Protocols Used
Basic	(Fig. 3b)	volatile	single-instance	GSI
Basic	(Fig. 3c)	persistent	single-instance	GSI (sync.)
Versioned	(Fig. 6)	volatile	single-instance	n/a
Versioned	(Fig. 6)	persistent	single-instance	GSI + BCAS
Versioned	(Fig. 6)	volatile	multi-instance	VLB
Versioned	(Fig. 6)	persistent	multi-instance	BCAS

Configuration. All configuration options (single-instance vs. multi-instance, volatile vs. persistent, choice of storage provider) are specified by class attributes (lines 14–16). It is possible to tweak them without altering the application semantics or changing the application code, which simplifies performance tuning.

3 IMPLEMENTATION

Geo is implemented in C# as extensions to Orleans, an open-source distributed actor framework available on GitHub [Orleans 2016]. 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 [Bernstein et al. 2014]. It already handles configuration changes and node failures within a cluster, fixing the directory and re-activating failed instances where necessary. However, prior to Geo, Orleans did 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, either 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.

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 [Corbett et al. 2013; Netflix 2011; Outages 2013; Ponemon 2013].

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 a currently unsupported combination. It is not difficult to implement, but has no performance benefits.

3.1 GSI Protocol Description

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 [Chandy and Misra 1984] 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.

3.2 GSI 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.

3.3 GSI Protocol Correctness

The goal of the GSI protocol 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 account for the possibility that *d* is simply slow or unable to communicate with *c*, but may yet be able to communicate with other clusters. In the second model, there are no communication failures.

Proposition 3.1. *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 d's request from d_1.
   - Suppose c received d's request from d_1 after c_3. Then c replied (*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 d's 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 d's 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 3.1. □

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 3.2. *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 3.1. □

3.3.1 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 3.2 and 3.1, 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.

3.4 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 [E-Tags 2016].

Local read and write operations (§2.2) 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.

3.4.1 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.

3.5 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 [Oki and Liskov 1988] or the Raft protocol [Ongaro and Ousterhout 2014; Raft 2016].

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 (Fig. 7). In each datacenter, 30 **front-end** (FE) servers 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.

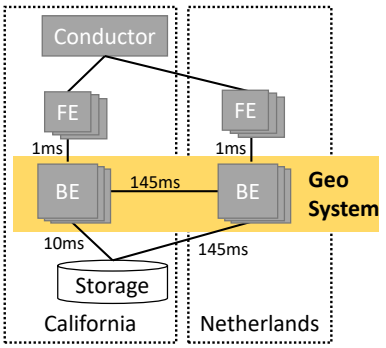

Fig. 7. Setup and approximate round-trip times.

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.

FE's are 4-core 1.6 GHz processors with 7 GB of RAM. BE's and the conductor are 8-core 1.6 GHz processors 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 [2005] benchmark explained in §4.4.1.

4.2 Latency

Our first series of experiments validates 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. 8. The left and right half are separate sections that contain latency numbers for volatile and persistent actors, respectively. The two columns at 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**

Fig. 8. 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).

far 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, and 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 cannot 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 dataÂňcenter. 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 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 measures 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.

4.3.1 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.

4.3.2 Volatile Single-Actor Throughput. For the volatile case, we distinguish three configurations (Fig. 9a). 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*

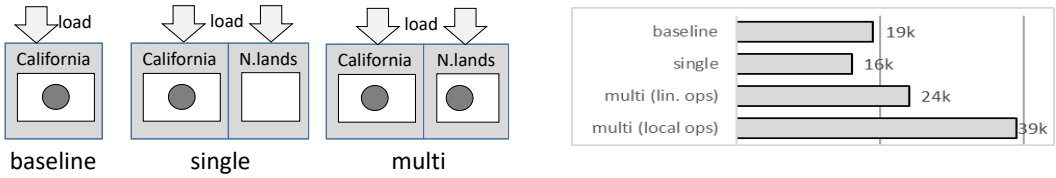

Fig. 9. Volatile single-actor throughput experiments. (a) **left**: configurations. (b) **right**: peak throughput measured.

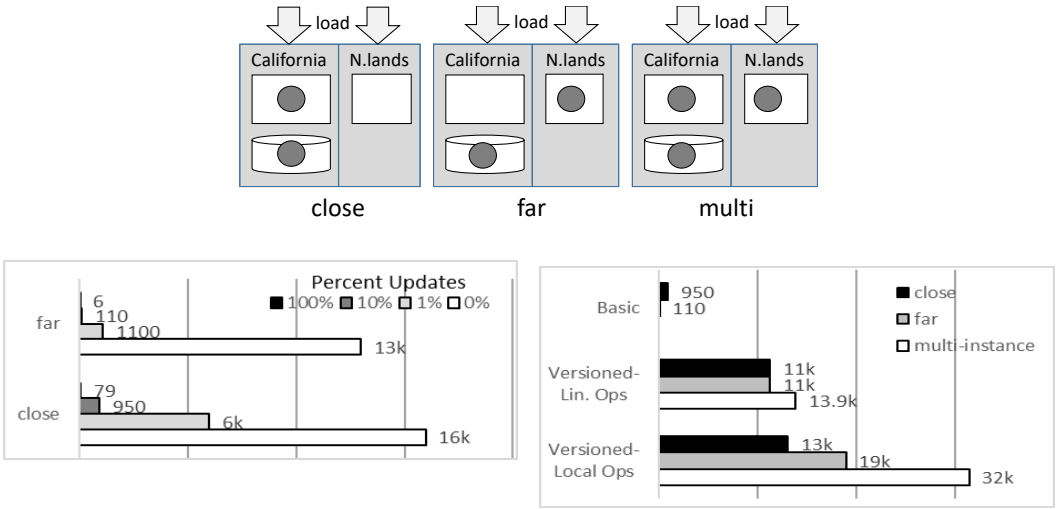

Fig. 10. Persistent single-actor throughput experiments. (a) **top**: configurations. (b) **bottom left**: basic API peak throughput. (c) **bottom right**: versioned API peak throughput.

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. 9b. 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 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*.

4.3.3 Persistent Single-Actor Throughput. For the persistent case, we distinguish configurations {*close*, *far*, *multi*} which keep the latest version in external cloud storage (Fig. 10a). 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. 10b. 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 good 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 throughput numbers for a 10% update rate are shown in Fig. 10c.

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).

4.4 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.

4.4.1 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 represents 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:

(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.

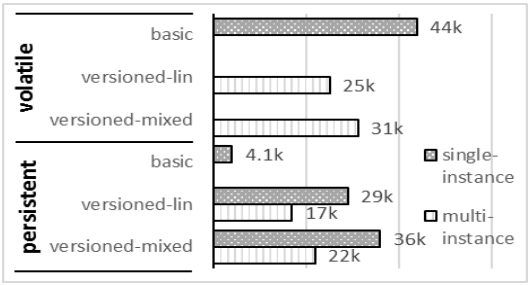

Fig. 11. Order processing throughput results.

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. 7. 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.

4.4.2 Results. We show peak throughput results in Fig. 11, 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. 9b, 10c), 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 Objects and Middleware. Distributed object systems from the 1980's and 90's are very similar to virtual actor systems, and share their ability to scale-out [Bal et al. 1992; Chase et al. 1989; Fekete et al. 1996]. However, hardly any focused on stateful applications, on geo-distribution, or consider situations where the object state is persisted externally. One exception is Thor [Liskov et al. 1999], 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 [Birman and Renesse 1993] and Horus [van Renesse et al. 1996]) and distributed shared memory systems (e.g., Interweave [Tang et al. 2003]) 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 [Redis 2016]. In memcached, cache entries are key-value pairs, which are spread over multiple servers using a distributed hash table [Memcached 2016]. Neither system is object-oriented and neither offers built-in support for geo-distribution.

Geo-Distributed Transactions. There has been a steady stream of papers in recent years on systems to support transactions over geo-distributed storage, using classical ACID semantics [Baker et al. 2011; Corbett et al. 2013], weaker isolation for better performance [Li et al. 2012; Lloyd et al. 2013; Sovran et al. 2011], and optimized geo-distributed commit protocols [Kraska et al. 2013; Nawab et al. 2013, 2015]. 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 [Kemme et al. 2010; Saito and Shapiro 2005; Terry 2008, 2013]. 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.

Weak and Strong Consistency. There are many approaches that, like Geo, distinguish between fast, possibly-inconsistent operations vs. slow consistent operations [Cooper et al. 2008; Fekete et al. 1996; Kraska et al. 2009; Li et al. 2012; Serafini et al. 2010]. Bayou [Terry et al. 1995] 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, Bayou's 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 [Burckhardt et al. 2014; Li et al. 2012; Shapiro et al. 2011a,b]. 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 [Burckhardt et al. 2014; Shapiro et al. 2011a,b], or commutativity annotations on operations as in RedBlue consistency [Li et al. 2012]. Also, linearizable operations are usually not supported [Li et al. 2012].

Geo-Replicated Storage. Some key-value stores offer some of Geo's functionality, but only for their custom storage and not for objects. Bigtable [Chang et al. 2008] and HBase [HBase 2016]

offer read and write access to a single copy, which is replicated for fault tolerance but not directly accessible to applications. Users can control data placement, whose location is controlled via a master directory. By contrast, Geo uses the GSI protocol to ensure all replicas agree on an object's location.

Dynamo [Decandia et al. 2007], and its derivatives Cassandra [Cassandra 2016] and Riak [Riak 2016], offer eventually consistent, highly-scalable geo-distributed storage using quorum consensus for both reads and writes. In Cassandra, updates replace the current state unconditionally. Thus, clients cannot atomically read-modify-write object state as in Geo or Bayou. Riak offers last-writer-wins too. It also enables linearizability via an atomic compare-and-swap operation using Multi-Paxos, but only within one datacenter. Alternatively, it can store all concurrent updates and leave conflict resolution to the user, e.g., using CRDTs. PNUTS [Cooper et al. 2008], another key-value store, supports linearizable operations on a primary copy of each record, which can migrate between datacenters. Its operations offer consistency-speed tradeoffs.

Cassandra programmers can configure how many replicas to maintain where, and 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 protocol [Lamport 1998]. 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.

A significant conceptual distinction is that in Geo, durability is provided by external storage - not by Geo itself. Geo is just a layer between the application program and the storage. This has some specific benefits when running a geo-distributed application, such as (1) for volatile actors, programmers do not have to pay the hefty price of geo-replication (waiting for a global quorum response), and (2) programmers can use Geo in conjunction with standard off-the-shelf cloud storage services, and within any pre-existing policies for data location and replication.

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 Geo's simple linearizable API, which puts geo-distributed applications within the reach of mainstream developers. Geo's 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 has been integrated into the Orleans project on GitHub [Orleans 2016]. 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

Akka 2016. Akka - Actors for the JVM. Apache 2 License, https://github.com/akka/akka. (2016).

Jason Baker, Chris Bond, James C. Corbett, JJ Furman, Andrey Khorlin, James Larson, Jean-Michel Leon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. 2011. Megastore: Providing Scalable, Highly Available Storage for Interactive Services. In *Conference on Innovative Data system Research (CIDR)*. 223–234. http://www.cidrdb.org/cidr2011/Papers/CIDR11_Paper32.pdf

H. E. Bal, M. F. Kaashoek, and A. S. Tanenbaum. 1992. Orca: a language for parallel programming of distributed systems. *IEEE Transactions on Software Engineering* 18, 3 (Mar 1992), 190–205. DOI: http://dx.doi.org/10.1109/32.126768

Philip A. Bernstein, Sergey Bykov, Alan Geller, Gabriel Kliot, and Jorgen Thelin. 2014. *Orleans: Distributed Virtual Actors for Programmability and Scalability.* Technical Report MSR-TR-2014-41. Microsoft Research.

Kenneth P. Birman and Robbert Van Renesse. 1993. *Reliable Distributed Computing with the ISIS Toolkit.* IEEE Computer Society Press, Los Alamitos, CA, USA.

Sebastian Burckhardt. 2014. Principles of Eventual Consistency. *Found. Trends Program. Lang.* 1, 1-2 (Oct. 2014), 1–150.

Sebastian Burckhardt, Alexey Gotsman, Hongseok Yang, and Marek Zawirski. 2014. Replicated Data Types: Specification, Verification, Optimality. In *Principles of Programming Languages (POPL)*. 271–284.

Sebastian Burckhardt, Daan Leijen, Jonathan Protzenko, and Manuel Fähndrich. 2015. Global Sequence Protocol: A Robust Abstraction for Replicated Shared State. In *European Conference on Object-Oriented Programming (ECOOP)*. 568–590.

Sergey Bykov, Alan Geller, Gabriel Kliot, James Larus, Ravi Pandya, and Jorgen Thelin. 2011. Orleans: Cloud Computing for Everyone. In *ACM Symposium on Cloud Computing (SOCC '11)*. Article 16, 14 pages.

Cassandra 2016. The Apache Cassandra Project. http://cassandra.apache.org. (2016).

K. M. Chandy and J. Misra. 1984. The Drinking Philosophers Problem. *ACM Trans. Program. Lang. Syst.* 6, 4 (Oct. 1984), 632–646. DOI: http://dx.doi.org/10.1145/1780.1804

Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. 2008. Bigtable: A Distributed Storage System for Structured Data. *ACM Trans. Comput. Syst.* 26, 2, Article 4 (June 2008), 26 pages. DOI: http://dx.doi.org/10.1145/1365815.1365816

J. Chase, F. Amador, E. Lazowska, H. Levy, and R. Littlefield. 1989. The Amber System: Parallel Programming on a Network of Multiprocessors. In *ACM Symposium on Operating Systems Principles (SOSP)*. ACM, New York, NY, USA, 147–158. DOI: http://dx.doi.org/10.1145/74850.74865

Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. 2008. PNUTS: Yahoo!'s Hosted Data Serving Platform. *Proc. VLDB Endow.* 1, 2 (Aug. 2008), 1277–1288. DOI: http://dx.doi.org/10.14778/1454159.1454167

James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. 2013. Spanner: Google's Globally Distributed Database. *ACM Trans. Comput. Syst.* 31, 3, Article 8 (Aug. 2013), 22 pages. DOI: http://dx.doi.org/10.1145/2491245

G. Decandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. 2007. Dynamo: amazon's highly available key-value store. In *Symposium on Operating Systems Principles*. 205–220. DOI: http://dx.doi.org/10.1145/1294261.1294281

E-Tags 2016. Editing the Web - Detecting the Lost Update Problem Using Unreserved Checkout. http://www.w3.org/1999/04/Editing/. (2016).

Alan Fekete, David Gupta, Victor Luchangco, Nancy Lynch, and Alex Shvartsman. 1996. Eventually-serializable Data Services. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing (PODC '96)*. ACM, New York, NY, USA, 300–309. DOI: http://dx.doi.org/10.1145/248052.248113

Alexey Gotsman and Sebastian Burckhardt. 2017. Consistency Models with Global Operation Sequencing and their Composition. to appear. In *Conference on Distributed Computing (DISC)*.

HBase 2016. Apache HBase. http://hbase.apache.org/index.html. (2016).

M. Herlihy and J. Wing. 1990. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.* 12, 3 (1990), 463–492. DOI: http://dx.doi.org/10.1145/78969.78972

Bettina Kemme, Ricardo Jiménez-Peris, and Marta Patiño Martínez. 2010. *Database Replication.* Morgan & Claypool Publishers.

Tim Kraska, Martin Hentschel, Gustavo Alonso, and Donald Kossmann. 2009. Consistency Rationing in the Cloud: Pay Only when It Matters. *Proc. VLDB Endow.* 2, 1 (Aug. 2009), 253–264. DOI: http://dx.doi.org/10.14778/1687627.1687657

Tim Kraska, Gene Pang, Michael J. Franklin, Samuel Madden, and Alan Fekete. 2013. MDCC: Multi-data Center Consistency. In *Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys '13)*. ACM, New York, NY, USA, 113–126. DOI: http://dx.doi.org/10.1145/2465351.2465363

Leslie Lamport. 1998. The Part-time Parliament. *ACM Trans. Comput. Syst.* 16, 2 (May 1998), 133–169. DOI: http://dx.doi.org/10.1145/279227.279229

Cheng Li, Daniel Porto, Allen Clement, Johannes Gehrke, Nuno Preguiça, and Rodrigo Rodrigues. 2012. Making Geo-replicated Systems Fast As Possible, Consistent when Necessary. In *Operating Systems Design and Implementation (OSDI)*. USENIX Association, Berkeley, CA, USA, 265–278. http://dl.acm.org/citation.cfm?id=2387880.2387906

Barbara Liskov, Miguel Castro, Liuba Shrira, and Atul Adya. 1999. Providing Persistent Objects in Distributed Systems. In *European Conference on Object-Oriented Programming (ECOOP)*. Springer-Verlag, London, UK, UK, 230–257. http://dl.acm.org/citation.cfm?id=646156.679840

Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. 2011. Don't Settle for Eventual: Scalable Causal Consistency for Wide-area Storage with COPS. In *ACM Symposium on Operating Systems Principles (SOSP)*. ACM, New York, NY, USA, 401–416. DOI: http://dx.doi.org/10.1145/2043556.2043593

Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. 2013. Stronger Semantics for Low-latency Geo-replicated Storage. In *Networked Systems Design and Implementation (NSDI)*. USENIX Association, Berkeley, CA, USA, 313–328. http://dl.acm.org/citation.cfm?id=2482626.2482657

Hernán Melgratti and Christian Roldán. 2016. *A Formal Analysis of the Global Sequence Protocol.* Springer, 175–191.

Memcached 2016. Available under BSD 3-clause license. https://github.com/memcached/memcached. (2016).

Faisal Nawab, Divyakant Agrawal, and Amr El Abbadi. 2013. Message Futures: Fast Commitment of Transactions in Multi-datacenter Environments. In *Conference on Innovative Data system Research (CIDR)*.

Faisal Nawab, Vaibhav Arora, Divyakant Agrawal, and Amr El Abbadi. 2015. Minimizing Commit Latency of Transactions in Geo-Replicated Data Stores. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD '15)*. ACM, New York, NY, USA, 1279–1294. DOI: http://dx.doi.org/10.1145/2723372.2723729

Netflix 2011. The netflix simian army. http://techblog.netflix.com/. (2011).

Brian M. Oki and Barbara H. Liskov. 1988. Viewstamped Replication: A New Primary Copy Method to Support Highly-Available Distributed Systems. In *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing (PODC '88)*. ACM, New York, NY, USA, 8–17. DOI: http://dx.doi.org/10.1145/62546.62549

Diego Ongaro and John Ousterhout. 2014. In Search of an Understandable Consensus Algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference (USENIX ATC'14)*. USENIX Association, Berkeley, CA, USA, 305–320. http://dl.acm.org/citation.cfm?id=2643634.2643666

Orbit 2016. Orbit - Virtual Actors for the JVM. BSD 3-clause license. https://github.com/orbit/orbit. (2016).

Orleans 2016. Orleans - Distributed Virtual Actor Model for .NET. MIT license. https://github.com/dotnet/orleans. (2016).

Outages 2013. The Year in Downtime: The Top 10 Outages of 2013. http://www.datacenterknowledge.com/archives/2013/12/16/year-downtime-top-10-outages-2013/. (2013).

Ponemon 2013. Ponemon Institute: 2013 Study on Data Center Outages. http://www.emersonnetworkpower.com/documentation/en-us/brands/liebert/documents/%20white%20papers/2013_emerson_data_center_outages_sl-24679.pdf. (2013).

Russell Power and Jinyang Li. 2010. Piccolo: Building Fast, Distributed Programs with Partitioned Tables. In *Operating Systems Design and Implementation (OSDI)*. 293–306.

Raft 2016. The raft consensus algorithm. https://raft.github.io/. (2016).

Redis 2016. http://redis.io/documentation/. (2016).

Riak 2016. Riak KV. http://docs.basho.com/riak/kv/. (2016).

James B. Rothnie and Nathan Goodman. 1977. A Survey of Research and Development in Distributed Database Management. In *International Conference on Very Large Databases (VLDB)*. 48–62.

Yasushi Saito and Marc Shapiro. 2005. Optimistic Replication. *ACM Comput. Surv.* 37, 1 (March 2005), 42–81. DOI: http://dx.doi.org/10.1145/1057977.1057980

Marco Serafini, Dan Dobre, Matthias Majuntke, Péter Bokor, and Neeraj Suri. 2010. Eventually Linearizable Shared Objects. In *Proceedings of the 29th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC '10)*. ACM, New York, NY, USA, 95–104. DOI: http://dx.doi.org/10.1145/1835698.1835723

SF Reliable Actors 2016. 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).

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

Mark Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011b. Conflict-free Replicated Data Types. In *13th Int. Symp. on Stabilization, Safety, and Security of Distributed Systems (SSS)*. Grenoble, France.

Yair Sovran, Russell Power, Marcos K. Aguilera, and Jinyang Li. 2011. Transactional Storage for Geo-replicated Systems. In *ACM Symposium on Operating Systems Principles (SOSP)*. ACM, New York, NY, USA, 385–400. DOI: http://dx.doi.org/10.1145/2043556.2043592

Chunqiang Tang, DeQing Chen, Sandhya Dwarkadas, and Michael L. Scott. 2003. Efficient Distributed Shared State for Heterogeneous Machine Architectures. In *Proceedings of the 23rd International Conference on Distributed Computing Systems (ICDCS '03)*. IEEE Computer Society, Washington, DC, USA, 560–. http://dl.acm.org/citation.cfm?id=850929.851916

D. Terry, M. Theimer, K. Petersen, A. Demers, M. Spreitzer, and C. Hauser. 1995. Managing update conflicts in Bayou, a weakly connected replicated storage system. *SIGOPS Oper. Syst. Rev.* 29 (December 1995), 172–182. Issue 5. DOI: http://dx.doi.org/10.1145/224057.224070

Douglas B. Terry. 2008. *Replicated Data Management for Mobile Computing*. Morgan & Claypool Publisher.

Douglas B. Terry. 2013. Replicated Data Consistency Explained Through Baseball. *Commun. ACM* 56, 12 (Dec. 2013), 82–89. DOI: http://dx.doi.org/10.1145/2500500

TPC-W 2005. TPC-W: Benchmarking An Ecommerce Solution. Revision 1.2, available at http://www.tpc.org/tpcw/tpc-w_wh.pdf. (2005).

Robbert van Renesse, Kenneth P. Birman, and Silvano Maffeis. 1996. Horus: A Flexible Group Communication System. *Commun. ACM* 39, 4 (April 1996), 76–83. DOI: http://dx.doi.org/10.1145/227210.227229

Windows Azure Cache 2016. http://www.windowsazure.com/en-us/documentation/services/cache. (2016).