# A Geo-Distributed Object-Oriented Cache

Philip A. Bernstein, Sebastian Burckhardt, Sergey Bykov, Natacha Crooks, Jose Faleiro, Gabriel Kliot,   
Alok Kumbhare, Vivek Shah, Adriana Szekeres, Jorgen Thelin

Microsoft Research[[1]](#footnote-1)

V 1.0, October 23, 2015

## ABSTRACT

Many service application programs cache data from storage and devices as objects, primarily for improving performance. This benefit is magnified when a service application spans datacenters, since network latency between datacenters can be large. In a geo-distributed environment, caching may also improve availability, since the cache may be accessible even if the underlying data source is not.

We describe a programming model and implementation for an object-oriented cache that can be distributed across many datacenters. The programming model has two interfaces, one that exposes a single instance of each object across datacenters and one that distinguishes the global state that all datacenters have agreed upon and each datacenter’s local state that includes the effects of its local updates that have not yet been synchronized with the global state. We propose a protocol that ensures there is at most one instance of an object worldwide, and we prove it correct. We describe an implementation of the model in Orleans, an object-oriented programming framework for developing scalable, fault-tolerant applications.

## 1 INTRODUCTION

The widespread availability of cloud platforms has made it easy for developers to geo-distribute their interactive applications at multiple datacenters. There are two main benefits to geo-distribution: better response time and better availability.

Geo-distribution offers the possibility of better response time than deploying a service to just one datacenter. In the latter case, a user can experience network latency from one millisecond to a few hundred milliseconds, depending on the datacenter network connection. By contrast, if the service is geo-distributed, more users are able to get service from a nearby datacenter, thereby minimizing network latency.

Geo-distribution also offers the possibility of higher availability by masking datacenter failures. Datacenter failures are sufficiently common that large-scale websites routinely engineer for them (e.g., [13]). If a user is connected to a datacenter that fails, the user can reconnect to another datacenter. If the time to reconnect the user and offer service is smaller than the datacenter’s mean-time-to-repair, then this results in improved availability. Moreover, if there is a network partition, the user will get service as long as one of the datacenters is reachable.

Service availability can be improved by having a backup service in another datacenter to which the service can failover. The backup service executes user requests only after the active service fails and requests are directed to the backup. This active-passive arrangement is somewhat fragile, in that the backup may fail to come up during the failover. A safer approach is active-active, where two or more datacenters each handle part of the normal applica­tion load and have sufficient headroom to take on additional load if one of the datacenters fail. An active-active configuration also offers better response time than active-passive. Therefore, it is usually the preferred approach when it is affordable and technically feasible. We will focus entirely on active-active configurations in this paper.

For stateless applications, geo-distribution simply requires deploying the application code to each datacenter, enabling it to run, and exposing communication endpoints by which it can be accessed. For stateful applications, there are many more issues that arise due to state management. Some of those issues are made more complicated by geo-distribution, such as data placement, partitioning, and replication. In this paper, we focus on one such issue, namely caching.

For good performance, it is desirable that each datacenter has cached copies of state that it accesses frequently. However, this raises many challenging questions, such as how to expose caches in the programming model, which mechanisms are best for a given workload, how to architect a system for pluggable cache mechanisms, and how to predict the performance that can be expected for different configurations. Before describing our problem more precisely, we briefly review datacenter-local caching as a prelude to exploring what changes when an application is geo-distributed.

### 1.1 A Caching Layer for Interactive Services

Interactive service applications that run in a datacenter are typically organized using a three-tier architecture with front-ends, middle-tier servers, and storage servers. Front-ends, such as web servers, are typically stateless. Their main role is to answer requests for static data that they can cache and to forward other requests to an appropriate middle-tier server. The middle-tier performs more compute-intensive application functions and accesses storage servers as needed to retrieve the freshest state or to update that state. It simplifies the design to have the middle-tier be stateless too. However, this limits scalability due to latency and throughput limits of the storage layer that has to be consulted for every request. A record-oriented caching layer is often added between the middle-tier and storage to improve performance [6][9][10][14]. Within a datacenter, latency between the middle-tier and cache servers is on the order of one millisecond, which is satisfactory for most interactive services and much faster than the tens of milliseconds often required to access storage.

To enable the cache to scale out independently of middle-tier servers and storage, it is often stored on cache servers, which sit in between the application middle-tier and storage servers. For example, memcached [6] and Redis [10] are popular cache management packages that are usually configured in this way. A write-through or write-back strategy can be used, depending on latency and fault tolerance requirements.

A caching layer improves performance, but loses most of the consistency and semantic guarantees of the underlying storage layer. To prevent inconsistencies caused by concurrent updates to a cached item, the application or cache manager has to implement a concurrency control protocol [7]. The advent of social graphs where a single request may touch many entities connected dynamically with multi-hop relationships makes it even more challenging to satisfy required application-level semantics and consistency on a cache with fast response for interactive access.

For these reasons, it is often desirable to combine middle-tier application functions with the caching layer to create an object-oriented cache. The cached state is hidden inside objects, which can be accessed only via methods. Methods maintain the semantic integrity of that state and synchronize concurrent accesses. Localizing this code avoids errors that arise from spreading it throughout the application. It also encourages function shipping, i.e. method calls, rather than data shipping, which is more efficient for large cached items.

In this paper, we focus on object-oriented caching. We assume that each object has an identity that the application can use to access the object. For example, the identity could be of the form serverID.objectID, as in Erlang [2] or Akka [1]. Or it could be a location-independent object id that a runtime library maps to a physical location using a directory system, as in Orleans [5]. We also assume that objects do not share state. Thus, the state of each object can be managed independently of the state of any other object.

### 1.2 Geo-Distributed Caching

Thus, there are two cases to consider: single-copy objects and replicated objects. We consider each case in turn.

#### 1.2.1 Single-Copy Objects

In the **single-copy case**, there is at most one copy of each cached object worldwide. This works well if the vast majority of accesses to the object are in one datacenter, since the object can be cached at that datacenter. Moreover, it avoids the problem of cache coherence, since there is only one copy of each object. However, it introduces a problem if objects are cached on demand: synchronization across datacenters is needed to guarantee that there is indeed at most one cached copy of each object.

For example, suppose a user Bob connects to a datacenter *c*, which causes Bob’s profile to be loaded. Now suppose Bob logs off, goes to the airport, and takes a flight to another continent. Since he is off-line for a while, the object *oBob* that represents him in *c* is deactivated. When Bob logs in to a local datacenter *d* at his destination, *oBob* should be loaded into *d*.

The same situation can arise if Bob’s workstation loses connectivity with *c* and is re-connected to *d*. In this case, *oBob* could be deactivated in *c* and then re-activated in *d*. Or, if *oBob* remains active in *c*, the process in *d* that communicates with Bob’s workstation could remotely access *oBob* in *c.* The important constraint is that *oBob* is active in at most one datacenter at any given time, which requires coordination between datacenters.

The previous scenario applies to any information that is associated with a user and frequently accessed by that user, such as: a shopping cart at an e-commerce site, a customer profile at an airline reservation website, or achievements and weapons in an on-line game. Similar scenarios apply to a device in an Internet of Things application, such as an object that represents the device or its location.

#### 1.2.2 Replicated Objects

In the **replicated case**, cached copies of an object exist at all datacenters where the object is frequently accessed. This avoids the latency of remote access, which is especially important in a geo-distributed setting, since a geo-remote access can be 100x slower than a datacenter-local access. However, it requires a mechanism to keep the cached copies of an object consistent.

Examples include inventory information in an e-commerce application, the leaderboard in a game, or a chat room whose users are connected to datacenters worldwide. In these cases, an update to the logical object needs to propagate to all of its copies. This is the classic problem of synchronizing updates to replicated data, in this case main memory objects rather than persistent storage objects.

#### 1.2.3 Storage Configurability

In both the single-copy and replicated cases, different storage configurations could be used, with different tradeoffs of availability, latency of update propagation, and throughput. For example, in the single-copy case, the data could be kept in persistent storage in the same datacenter as the cached copy, to minimize the latency of propagating cache updates to storage. For fault tolerance, the persistent storage may be replicated within the datacenter and/or across datacenters using a cloud storage service. Or if the application can reconstruct the cached object state from that of its clients (e.g., if they are devices), the cached copy might not maintain a persistent copy in storage.

The same storage configurations make sense as options for the replicated case. For example, the cached copies might all checkpoint their updates to a centralized persistent store, which may be replicated within a datacenter and/or across datacenters. The storage system could maintain the consistency of the persistent copies, using either a primary-copy or multi-master synchronization algorithm. Or, if the cache coherence protocol maintains the consistency of the cached copies, then the persistent copies might be regarded as independent of one another with no storage-level mechanism to synchronize them.

For a given application, the choice of storage configuration can change over time. To make it easy to reconfigure storage after the application is written, it is desirable that the application programming interface be unaffected by the choice of configuration.

## 1.3 Contributions

In this paper, we describe the design and implementation of a system for geo-distributed object-oriented caching. The design supports both the single-copy and replicated cases. It includes an application programming interface (API) that is convenient for application developers and can be implemented efficiently. And the implementation is configurable for high availability, scalability and performance for interactive application workloads.

Our system includes several novel aspects. First, for the single-copy case, it includes a new optimistic protocol for datacenters to synchronize with each other to ensure that there is at most one instance of each object worldwide. Second, for the multi-copy case, it introduces a new API that distinguishes the global state that all datacenters have agreed upon and each datacenter’s local state that includes the effects of its local updates that have not yet been synchronized with the global state. It supports operations to read and update either the local or global state. We will argue that when replication is desirable, applications can meet most of their state management needs using local operations, thereby avoiding synchronization delay with other datacenters. Third, we propose an implementation architecture that allows plugging in different storage systems, without affecting the API. We report on our implementation of the architecture in the Orleans programming framework [3][5][8]. A future version of this paper will include a performance evaluation of the implementation.

The paper is organized as follows. After providing some basic definitions in Section 2, we describe our solution for the single-copy case in Section 3 and the multi-copy case in Section 4. Section 5 sketches the implementation of the model in Orleans, a programming framework for distributed actors.

## 2 CLUSTERS AND MULTI-CLUSTERS

We abstract the informal notion of a datacenter by the term cluster. We define a **cluster** to be a named set of servers connected by a high-speed network. A datacenter might contain multiple clusters, and those clusters might reside in distinct failure zones that operate and fail independently. Also, a cluster could span two datacenters, for example, if they are nearby and connected by high-speed communications. For the purposes of this paper, it is unimportant how the set of servers in a cluster is determined.

A geo-distributed system consists of a named set of clusters, called a **multi-cluster**, connected by a wide-area network, typically the Internet. The set of clusters in a multi-cluster can change only by the execution of the administrator operations **add-clusters** or **remove-clusters** and changes slowly. These operations send a multi-cluster-change notification to all members of the multi-cluster.

For now, we assume the set of clusters in the multi-cluster is static and there are no network partitions. We will drop these assumptions later.

## 3 GLOBAL SINGLE-COPY OBJECTS

### 3.1 Motivation

For some single-copy objects, it is statically known that they will be instantiated only at a particular cluster. For example, it might be statically known that in a cloud game, a player object and a game object representing the game that the player is playing are always instantiated at the cluster where the player’s game console is connected. Such objects are oblivious to geo-distribution, so no inter-cluster synchronization is required.

Inter-cluster synchronization is needed if an object can be instantiated in different clusters, but should be instantiated at only one cluster at any given time. For example, consider an object representing a geographic region of a game. At any given moment, the region is instantiated at one cluster—the one where most of the players in that region are connected. If all players leave that region, then the object might be deactivated and later reinstantiated at a different cluster when players re-enter the region. We call this a **global single-instance object.**

A global single-instance object must have an identity. This enables the system to avoid creating duplicate copies of the object and enables clients to make a remote access to the object.

It is possible that two clusters try to instantiate the same global single-instance object concurrently. A mutual-exclusion protocol is needed to avoid this conflict. There are two strategies a cluster could use, pessimistic or optimistic. Using a pessimistic strategy, before instantiating the object, a cluster *c* asks all other clusters for permission. A cluster *c’* grants that permission if it does not know of an instantiation and is not trying to create one itself. If all clusters grant permission, then *c* can instantiate the object. A problem with this strategy is that it adds substantial latency to object instantiation to avoid a race condition that in most applications is unlikely to occur.

By contrast, *c* could use an optimistic strategy, where it instantiates the object before asking other clusters for permission, thereby avoiding the latency of object instantiation. Of course, if *c* allows callers to update the object before it obtains permission from all of the other clusters, then it risks a later conflict. That is, it could be racing with another cluster *c’* that also instantiated and then updated the object.

If a cluster denies permission to *c* because another cluster has instantiated the same object, then one of them needs to give it up. It is undesirable that this situation can arise, because the application developer needs to write code to handle the conflict and that code might not be able to hide the confusion completely from the end user. After presenting our optimistic protocol, we will discuss ways to avoid duplicate activations and why we believe that most application developers will choose the optimistic strategy.

### 3.2 Global Single-Instance Protocol

We present an optimistic protocol for a cluster to obtain exclusive ownership of an object. *A* cluster *c* tries to gain ownership of an object *o* by sending a request to all other clusters in its multi-cluster *m.* If all clusters in *m* reply **Pass**, then *c* is the sole owner of *o*. If only a proper subset of *m* replies Pass and none reply **Fail**, then *c* is in doubt about its ownership of *o* and needs to try again periodically to get permission from all of the clusters. Mean­while, since the protocol is optimistic, an in-doubt cluster that has an instance of *o* can process method calls on it.

A cluster *d* might reply Fail for one of two reasons. The first reason is that *d* owns *o* or is in doubt about its ownership. In this case, *d* reports this fact to *c* and *c* caches a reference from *o* to *d*. The second reason is that *d* is trying to gain ownership of *o* too. In this case, since *c* and *d* are in a race, one should give up. We define a function **Precedence** to make the decision. For each object, Precedence defines a total order over clusters. If Precedence(*o*, *c*) > Precedence(*o*, *d*), then *d* gives up by replying Pass and delays briefly before retrying, to give *c* a chance to obtain ownership. If Precedence(*o*, *d*) > Precedence(*o*, *c*), then *d* replies Fail which tells *c* to give up and retry after a short delay, to give *d* a chance to obtain ownership.

After *c* obtains ownership, it could send a second message to announce that fact to all clusters in *M*. However, this is wasteful in the common case that most other clusters never access *o*. So we take the opposite approach. When a cluster *d* wants access to *o*, it simply attempts to instantiate *o*. If another cluster *c* owns *o*, then *d* will receive a reply with that information.

The protocol for instantiating global single-instance objects maintains a directory at each cluster *c*. The directory maps each object id *o* to a pair [d, *state*], where *d* is the name of a cluster (possibly null) and *state* is one of {Owned, Requested, Doubtful, Cached, Loser}. In the directory for cluster *c*, the directory entry for *o* can be one of the following:

* [*c*, Owned] – *c* has exclusive ownership of *o*;
* [*c*, Doubtful] – *c* has an instance of *o* but has not obtained exclusive ownership;
* [*c*, Requested] – *c* has an instance of *o* and is running the protocol to obtain ownership of *o*;
* [*d*, Cached] – *c* believes that *d* has an instance of *o*, where *d≠c*;
* [null, Loser] – *c* lost a race with another cluster that is requesting ownership of *o*.

If a cluster *d* receives a request from cluster *c* to instantiate object *o*, it replies based on its directory entry for *o* as follows:

* no directory entry => reply (Pass, null);
* [*d*, Owned] => reply (Fail, *d*)
* [*d*, Doubtful] => reply (Fail, *d*)
* [*d*, Requested] => if precedence(*d*, *o*) > precedence(*c*, *o*), then reply (Fail, null)   
   else replace *d’*s directory entry by [*d*, Loser] and reply (Pass, null)
* [*e*, Cached] => reply (Pass, null)
* [null, Loser] => reply (Pass null)

Cluster *c* waits until it receives replies from all clusters or times out waiting for some of them. It then processes the replies in the following critical section, during which it does not process any messages from other clusters on the same object:

[Start critical section] If *c* is in state Loser, then while it was waiting for replies to its instantiation request, it must have received a competing request for the same object from a higher precedence cluster. So it delays a short time to give the race winner a chance to obtain ownership. It then restarts the protocol by broadcasting a request to obtain ownership.

If *c* is not in state Loser, then it must still be in state Requested and proceeds as follows:

1. If *c* received (Pass, null) from all clusters, then it exclusively owns *o*, so it updates its directory entry to [*c*, Owned].
2. Otherwise, if *c* receives (Fail, *e*) where e*≠*null from one or more clusters, then another cluster owns *o*. Therefore, *c* updates its directory entry to [*f*, Cached], where *f* is the highest precedence cluster that returned a reply of the form (Fail, *f*).
3. Otherwise, if *c* receives (Fail, null) from one or more clusters, then it restarts the protocol by broadcasting a request to obtain ownership.
4. Otherwise, *c* either received no replies or it received only (Pass, null) replies from a proper subset of the clusters. It changes its directory entry to [*c*, Doubtful].

[End critical section]

Periodically, a cluster broadcasts a request to obtain ownership for all of the objects in the Doubtful state in its directory. It changes each object’s directory entry from Doubtful to Requested and runs the same protocol as above, but batched to avoid sending a separate round of messages per object.

The critical section is needed to avoid a race between two requestors, as follows:

1. Cluster *c* sends a request to cluster *d*.
2. Cluster *d* receives the request and replies (Pass, null)
3. Cluster *d* sends a request to *c*
4. Cluster *c* determines that it is not in state [*c*, Loser] and then is interrupted by the incoming request from *d*, thereby stopping in the middle of the critical section.
5. Cluster *c* is still in the [*c,* Requested] state. If precedence(*c*) < precedence(*d*), then *c* moves to state [*c*, Loser] and sends (Pass, null) to *d*
6. Cluster *c* continues step 4 processing replies. If it did not receive any (Fail, -) replies, then it moves into state [*c*, Doubtful] or [*c*, Owned] (because it checked earlier that it is not in state Loser).
7. Cluster *d* is now free to move ahead into state [*d*, Doubtful] or [*d*, Owned], thereby potentially violating the single-owner constraint.

In an alternative protocol design, if a cluster is in state [*d*, Doubtful], instead of replying (Fail, *d*), it could follow the same logic as in state [*d*, Requested]. (I.e., if Precedence(*d*, *o*) > Precedence(*c*, *o*), then reply (Fail, null) else replace its directory entry by [*d*, Loser] and reply (Pass, null).) This would ensure that the highest precedence cluster that wants to instantiate the object is able to do so. The downside is that a cluster could be preempted by a higher precedence cluster, even though it has been processing operations on the object for a long time (because it is in the Doubtful state and is optimistic). Since we are biased toward choosing optimistic execution, we decided against this approach.

### 3.3 Discussion

The protocol description above assumes a unique multi-cluster. If a multi-cluster can partition into two independent sets of clusters that cannot communicate with each other, then each partition will run the protocol as if it were the one-and-only multi-cluster. In this case, each partition can independently instantiate an object, thereby defeating the purpose of the protocol.

One can avoid the problem by adopting the pessimistic protocol, where a cluster instantiates an object only after it has obtained permission from all other clusters in its multi-cluster. Or one could use a Paxos-like protocol to ensure that all available clusters in a quorum know about all instantiated objects. However, both are expensive. Moreover, they lead to unavailability when a partition occurs. That is, relative to the CAP tradeoff, in the presence of a partition they offer consistency instead of availability [4][11]. In our experience, when managing cached items, application developers usually make the opposite choice, that is, availability over consistency.

Given this choice, a developer needs to do something about the possible inconsistency of two instances of a cached object, *o*. If *o* has an external home *h* for its authoritative value, such as a record in storage or a register of a device, then the inconsistency problem can be handled using a compare-and-swap, i.e., optimistic concurrency control. That is, *o* propagates its updated value to *h* only if *h’*s value is unchanged since the last time *o* read or wrote *h*. If the instance of *o* detects that *h* has changed, then it knows that another copy of *o* is running and special action is required to address the conflict. For example, it could re-read *h* and apply its last update to that value. Or it could reject the update to *o* because it cannot write it through to *h*. Or it could simply discard the value of *o*.

We offer another approach to the problem in the Section 4, where the API makes it visible that there are replicated copies of the cached object and offers convenient semantics on how to synchronize them.

A proof of the following correctness guarantees of the global single-instance protocol appears in Appendix A.

**Proposition 1:** In the absence of communications failures, the global single-instance protocol ensures that for a given object at most one cluster can have a directory entry for the object in the Owned or Doubtful state.

**Proposition 2:** Evenin the presence of communications failures, the global single-instance protocol ensures that for a given object at most one cluster can have a directory entry for the object in the Owned state.

### 3.4 Configuration Changes

Our description of the global single-instance protocol assumed it executes in the context of a fixed multi-cluster. We now relax that assumption and consider what happens if the set of clusters in the multi-cluster changes while the protocol is executing.

Recall that a multi-cluster configuration can change only by having an administrator invoke the operations add-clusters or remove-clusters. We impose an additional constraint that add-clusters or remove-clusters should execute only if all clusters in the current multi-cluster configuration know about that configuration. This is relatively easy to enforce by assigning a unique monotonically-increasing timestamp to each configuration and having each cluster in a multi-cluster periodically announce the timestamp of the multi-cluster under which it is operating. As we will see, these restrictions are needed to ensure that the global single-instance-protocol correctly maintains mutual exclusion of activations under configuration changes.

A cluster should reject any message it receives from another cluster that is not in its multi-cluster. Such a message could indicate that the recipient cluster is not aware of the latest multi-cluster. So the recipient could poll for the current state of the multi-cluster, if the multi-cluster maintenance subsystem offers a polling option. In any case, if there is indeed a new multi-cluster configuration, the recipient cluster will eventually hear about it.

In the next two subsections, we analyze the effect on the global single-instance protocol of adding clusters to the multi-cluster and removing clusters from the multi-cluster. For each cluster that is still in the multi-cluster or that is added or removed, we need to ensure that no two clusters have an object in the Owned state (i.e., consistency), and for any object for which a cluster can become owner, the cluster will attempt to do so (i.e., liveness).

#### 3.4.1 Adding a Cluster

To ensure single-instancing of objects, we add two rules:

1. When a cluster *c* is added to a multi-cluster, it changes all of its Owned objects to Doubtful.
2. When a cluster *c* is added to a multi-cluster, it changes all of its Requested objects to Loser, thereby stopping truncating any in-flight executions of the global single-instance protocol.

We require (1) because objects in the multi-cluster that *c* is joining might have objects in the Owned state that *c* owns too. By changing those states to Doubtful in *c*, *c* will recheck that it has ownership during its next periodic broadcast to request ownership of its Doubtful objects.

We require (2) to prevent an in-flight execution of the protocol in *c* to violate the single-instance guarantee. To see how this might arise in the absence of rule (2), consider the following example:

1. Clusters *c* and *c’* are in multi-cluster *m.*
2. *c* starts the protocol to become owner of object *o*
3. *c’* replies Passto the request from *c*
4. *c* is removed from *m*, which creates cluster *m’*
5. *c’* runs the protocol to become Owner of *o* in *m’*
6. *c* is added back to the multi-cluster, creating multi-cluster *m”*
7. *c* now finishes its protocol for *o*, which is started while it was in cluster *m*. Since it already received Pass from *c’*, it does not ask *c’* again. Therefore, it can become Owner, which violates single instancing.

An alternative to rule (2) to allow *c* to complete its executions of the protocol. But if *c* finishes the protocol in the Owned state and is in a different multi-cluster than when it started the protocol, it changes the Owned state to Doubtful. Like rule (2), this forces *c* to retry the protocol later in the new multi-cluster.

For each case, a consistency violation is possible only for objects in the Owned state. For other states, only liveness is at stake.

**Clusters added to the multi-cluster**: Consider object *o* in the directory of cluster *c’* that was added to multi-cluster *m*, thereby creating *m’*:

* Owned – By rule (1), *c’* has no objects in the Owned state, so it cannot violate consistency.
* Doubtful – *c’* will periodically broadcast a request for ownership in *m’* of its Doubtful objects.
* Loser – There are two cases: (i) in its previous multi-cluster, *c’* was in a race with another cluster to obtain ownership of *o*; and (ii) in its previous multi-cluster, *c’* was in state Requested and by rule (2) the entry was changed to Loser. In either case, *c’* will periodically retry gaining ownership of *o* in *m’*.
* Cached – If the state is [*d*, Cached] and *d* is not in *m’*, then this directory entry for *o* should be deleted. Otherwise, it is still potentially valid. If it is not valid, *c’* will find out the next time it tries accessing *o*.
* Requested – By rule (2), *c’* does not have objects in this state.

**Clusters that are still in the multi-cluster:** Consider object *o* in the directory of a cluster *c* that learns that a new cluster *c’* has been added to its multi-cluster *m*, thereby creating *m’*:

* Owned – By rule (1), no objects in *c’* are Owned, so *c* still has exclusive ownership of *o*.
* Doubtful – Cluster *c* will try to obtain ownership of *o* in *m’* via its periodic broadcast to request ownership of its Doubtful objects.
* Loser – Cluster *c* was in a race with another cluster that is attempting ownership of *o*. It will periodically retry its attempt to gain ownership of *o* in *m’*.
* Cached – Since all clusters in *m* are in *m’* too, the cached entry [*d*, Cached] is still valid.
* Requested – The protocol is in progress and can continue until it completes using *m*. Although a newly added cluster does not participate in the protocol, it can obtain ownership of *o* only by running the protocol in *m’*, which will cause it synchronize properly with *c*.

#### 3.4.2 Removing a Cluster

We now consider the effects of the remove-clusters operation.

**Clusters that are still in the multi-cluster:** Suppose a cluster *c* learns that some clusters have been removed from its multi-cluster *m*, resulting in multi-cluster *m’*. For a given object *o*, let us consider the effect on the state of *o*:

* Owned – Cluster *c* has confirmed that no other cluster has instantiated *o*. Removing a cluster from *m* does not change this fact, so consistency is preserved.
* Doubtful – One or more clusters have not told *c* whether they have instantiated *o*. It is therefore possible that *c* can now claim the Owned state. It will check this during its next periodic broadcast to request ownership of its Doubtful objects.
* Loser – Cluster *c* was in a race with another cluster that is trying to obtain ownership of *o*. It will periodically retry its attempt to gain ownership.
* Cached – If the state is [*d*, Cached] and *d* was removed from the multi-cluster, then *c* should delete this directory entry for *o*. Otherwise, it is still valid.
* Requested – The protocol is in progress and can continue until it completes. The result is unaffected by the removal of some clusters.

**Clusters removed from the multi-cluster**: Suppose a cluster *c′* was removed from the multi-cluster but does not know it. If *c’* is still operational, then the multi-cluster is partitioned. In-flight executions of the single-instance protocol by *c’* are based on *m* and will finish up normally. Cluster *c’* could incorrectly conclude that it is Owner of *o*, until it knows about the new multi-cluster. This will be resolved when c’ is added back to the multi-cluster, if not sooner by an administrator.

If *c’* knows it’s not part of the multi-cluster, then it should not run the global single-instance protocol, thereby favoring consistency over availability. If it is operational and must instantiate objects, then it could continue operating under multi-cluster *m*, as in the previous paragraph.

## 4 QUEUED OBJECTS

In Section 1.2, we distinguished two cases of geo-distributed caching: the single-copy case and the replicated case. The previous section presented a protocol for the single-copy case, where an object has one cached instance worldwide. This section presents a programming model for the replicated case in which cached copies of an object exist at all datacenters where the object is frequently accessed.

We have two goals in defining a programming model. First, it should be convenient for developers. This means it should have simple and precise semantics, which makes it easy for developers to reason about and customize its behavior. This also means that its functionality should match the requirements of most interactive applications that benefit from a geo-distributed cache. Second, it should support a variety of storage configurations that offer tradeoffs of availability, latency of update propagation, and throughput. Since the application workload and choice of storage configurations can change over time, it should be easy to reconfigure storage without modifying application code.

In our model, updates to an object are specified as functions. The state of a copy of an object is the sequence of updates that has been applied to it (see Figure 1). Conceptually, there are many local copies of an object and one global copy. The global copy is the true state of the object. Updates can only be appended to its state. Each local copy consists of a recent snapshot of the global copy, called its **confirmed state**, plus a queue of local updates that have been applied to that local copy but have not yet propagated to the global state.

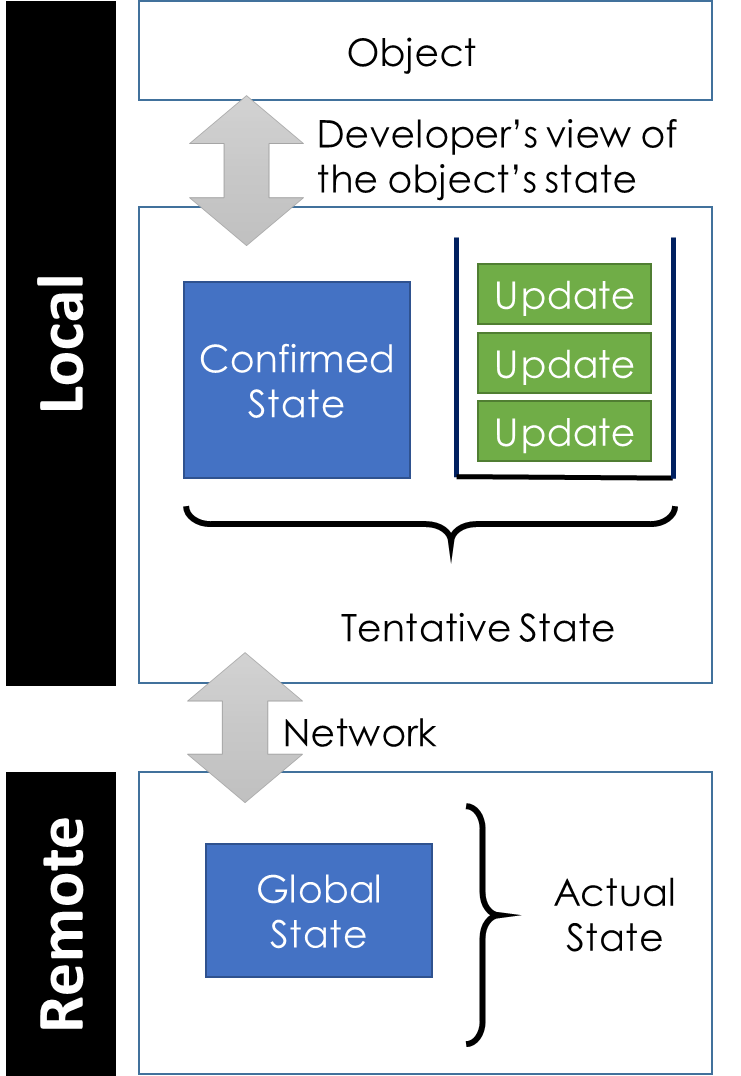


Figure 1 Programming model for replicated objects

The developer’s view of the object’s state consists of a local state and the global state, which are accessed using the following operations:

* UpdateLocal – append an update to the local state of the object.
* ReadLocalConfirmed – return the value of the local copy’s latest snapshot of the global copy.
* ReadLocalTentative – return the result of tentatively applying the local copy’s queue of updates to the confirmed state.
* UpdateGlobal – append an update to the local copy’s queue of updates, apply the queue of updates to the global state, and then refresh the local copy’s confirmed state with the resulting global state.
* ReadGlobal – same as UpdateGlobal, but without supplying a new update.

Operations on the local state are intended to have low latency and, we claim, are sufficient for most application operations on the object. However, reads are either stale (via ReadLocalConfirmed) or an approximation of the state that is subject to revision (via ReadLocalTentative). Operations on the global state have higher latency, but they return a fresh state that is not subject to revision.

An UpdateLocal operation is low latency because it simply appends the update to the sequence of updates in the caller’s local copy of the object. A background thread *t* periodically but eagerly propagates that sequence *s* of local updates to the global state. In response, *t* receives a new snapshot of the global state that includes *s*, and possibly other updates that were issued against other local states. Thread *t* replaces the local copy’s confirmed state by the new global snapshot and deletes the sequence of local updates that it sent to the global state.

The updates in *s* are applied to the global state in the order they appear in *s*, thereby preserving causal dependencies between those updates. However, updates from other local states might be interleaved between those in *s*. Thus, the result of applying each update in *s* might be different than its result when applied to the local state. In the terminology of the Bayou system [12], ReadLocal supports read-your-writes and monotonic reads.

The database field distinguishes between primary-copy replication and multi-master replication. Our model covers both approaches. The primary-copy approach is covered by UpdateGlobal and ReadLocalConfirmed. Updates to the global copy are sequenced by the order in which they are applied, and the result of that sequence is periodically used to refresh each local copy. The multi-master approach is covered by UpdateLocal and ReadLocalTentative. Updates are applied to each copy. Each read sees the result of all previous local updates by the same caller, and updates from other copies are applied asynchronously.

Unlike some multi-master systems, the model does not allow gossiping between local copies, where pairs of copies exchange updates. Rather, updates from one local copy have to be sequenced against the global copy before they are applied to other local copies. This adds a degree of predictability for an application that uses a local copy. The cost is potentially higher latency of applying some remote updates.

The distributed computing field would characterize the behavior of this local-global model as totally-ordered broadcast. That is, updates from different clients are totally ordered by the global copy and then broadcast to all local copies in that order.

There are many storage configurations that can be used to support the local-global model. One could interpret the model literally, by having a central storage server that holds the global copy, and other servers that hold local copies. The global and local copies could be in main memory or stable storage. In a geographically distributed setting, this configuration corresponds to storing the global copy in one datacenter and local copies in other datacenters.

Another approach is completely decentralized by using a Paxos-style protocol to totally order updates. Each copy can attempt to lead a round of the protocol to get all copies to accept an update. This can support synchronous replication, if all copies (or a quorum of copies) accept an update before the next update is processed.

## 5 IMPLEMENTATION

We implemented the global single-instance and replicated object models of Sections 3 and 4 in Orleans, an object-oriented programming framework for developing scalable, fault-tolerant applications. In an Orleans application, objects are commonly used as a cache for state whose official value resides in an underlying storage system. Moreover, objects do not share state. Therefore, an Orleans application that spans datacenters experiences the types of latency and consistency problems that our single-instance and replicated object models are designed to solve.

In Orleans, each class has a key, each of whose values uniquely identifies an object of the class. Given the key of an object *o*, an application can invoke a method *m* on *o*. If *o* is active, the Orleans runtime will find it and invoke *m* on it. If *o* is not active, the Orleans runtime will choose a server on which to activate *o*, invoke *o’*s constructor on that server (e.g., to populate *o’*s state), and then invoke *m* on *o*. To support this behavior, the Orleans runtime maintains a distributed, fault-tolerant directory that maps object id’s to server id’s. Details can found in [3].

The main technical components of our implementation of global single-instance objects and replicated objects are shown in Figure 2. We describe each component in turn.

Multi-Cluster Network

Global   
Single-copy Object

Replication Adaptor

Replicated State Object

Physical Network

Storage Adaptor

Figure 2 Implementation Components

### 5.1 Multi-Cluster Network

The multi-cluster network is a set of clusters that can propagate information about multi-cluster configurations. Each multi-cluster must be a subset of the clusters in its multi-cluster network.

A multi-cluster network assumes the existence of a physical network that enables server-to-server communication between all servers across all of the clusters. In Windows Azure, this can be done by creating VNETs within each region and then connecting those VNETs using gateways.

A multi-cluster is defined by a list of cluster names and the timestamp when the multi-cluster was defined. Each server in the multi-cluster network supports an operation to inject a new multi-cluster definition *D*. Multi-cluster *D* replaces all multi-clusters that include any cluster in *D* and have smaller timestamp. To accomplish this replacement, each cluster propagates the latest multi-cluster definition it knows about to every other cluster in its multi-cluster network. Thus, all operational clusters in the multi-cluster quickly converge to the same multi-cluster definition.

We rely on an administrator to configure the multi-cluster, instead of just working with some automatically determined list of currently active clusters, for two reasons. First, many communication protocols do not work correctly if clusters chaotically appear, disappear, and re-appear. In particular, the single-instance protocol could erroneously create multiple instances without detecting the problem. Second, adding or removing a cluster from the multi-cluster is usually just one part of a process that involves steps that are not directly under the control of Orleans, such as creating, deleting, or updating deployments, or changing the routing of user load.

### 5.2 Global Single-Copy Object

An object is identified as a global single-copy object by annotating its class with “GlobalSingleInstance.” Each such object is instantiated using the protocol described in Section 3.2.

Suppose a method is invoked on a global single-copy object *o* in cluster *c*, and *c’*s object directory does not have a mapping for *o*. In this case, the Orleans runtime in *c* will (optimistically) activate *o* in *c* and then execute the global single-instance protocol. If *c* learns that *o* is active in another cluster *d*, then it deactivates the local copy of *o* that it created, caches “*o* maps to *d*” in its directory, and directs the method call to *d*. If *o’*s state in *c* was modified before it was deactivated and it is important not to lose that modification, then its deactivation method should synchronize that state change with the copy in *d*. A common approach is to do a compare-and-swap on the storage record that contains *o’*s state.

### 5.3 Replicated State Object

Recall that a queued object has a local state, which includes a queue of pending updates, and a global state. An implementation must manage each of these states and propagate updates from the local state to the global state.

The local state is a cached copy that is co-located with each object. Conversely, there is a single conceptual copy of the global state of a replicated object. There are many possible ways that a storage system could be configured to offer this abstraction. The storage system could be a literal implementation of this abstraction, where there is a single copy that can be read and updated by all local copies worldwide. Or the global state could itself be replicated, but provide the illusion that there is just one copy that is read or updated. The choice of configuration depends on the storage systems that are offered by the underlying computing platform and on the relative importance of latency, throughput, and fault tolerance, and perhaps other factors.

The implementation of replicated state objects has two components, a generic replication adaptor and a storage adaptor. The replication adaptor implements the local queue of pending updates, i.e., those updates that have not yet been applied to the global state. This function is generic in the sense that it is independent of how storage is structured or accessed. By contrast, the storage adaptor has the storage-specific functionality. It holds the local tentative state and confirmed state and it uses whatever read and update operations are supported by storage.

The types of read and write operations can have a major effect on how the storage adaptor works. As a simple example, the interface to the storage system could accept update operations and apply them atomically to the state of the object. Thus, the state of the object would reflect a sequential execution of the updates applied to it, as desired.

As a more complex example, suppose the storage system is a key-value store that supports simple read and write operations on records. Furthermore, suppose the state of each object is mapped to a record whose key is the object id of the object that is mapped to it. In this case, to apply an update to the global state of an object, the storage adaptor must read the record corresponding to that object, apply the update to that record, and write the record back to the key-value store. Following this strategy, if two local copies each try concurrently to apply their update to the same object to the global copy, there could be a race condition; both attempts to apply their update could read the current state of the record before either of them update the record. This would violate the semantics of the programming model, since the update that was applied first would be lost.

One way to avoid this race condition is to use optimistic concurrency control. For example, in Azure Table, a record has a state variable, called an ETag, which is updated to a unique value every time the record is written. A record’s ETag is returned with each read operation. A write operation can do a compare-and-swap, where the record is written only if the record has the ETag supplied with the write. Thus, to apply an update, a storage adaptor for Azure Table can use the ETag to ensure the object’s record has not changed between the time it read the record and the time it wrote it.

## 7 FUTURE WORK

This paper is a work in progress. Future versions will include a summary of prior work and performance measurements.

## REFERENCES

1. Akka documentation, <http://akka.io/docs/>
2. Armstrong, J.: Erlang. CACM, 53, 9 (Sept. 2010), 68-75
3. Bernstein, P.A., S. Bykov, A. Geller, G. Kliot, and J. Thelin: Orleans: Distributed Virtual Actors for Programmability and Scalability, MSR-TR-2014-41, <http://research.microsoft.com>
4. Brewer, E.A.: Pushing the CAP: Strategies for Consistency and Availability. IEEE Computer 45(2): 23-29 (2012)
5. Bykov, S., A. Geller, G. Kliot, J. Larus, R. Pandya, and J. Thelin: Orleans: Cloud Computing for Everyone. In *SOCC* 2011, 16:1-16:14
6. Memcached, <http://memcached.org/>
7. Miller, M. S., E.D. Tribble, and J. Shapiro: Concurrency Among Strangers: Programming in E as Plan Coordination. In *Proc. of the Int'l Symp. on Trustworthy Global Computing*, 2005, Springer, 195-229.
8. Orleans – Distributed Virtual Actor Model, https://github.com/dotnet/orleans
9. Power, R., and J. Li: Piccolo: Building Fast, Distributed Programs with Partitioned Tables. OSDI 2010: 293-306
10. Redis, <http://redis.io/documentation/>
11. Rothnie, J.B. Jr., N. Goodman: A Survey of Research and Development in Distributed Database Management. VLDB 1977: 48-62
12. Terry, D.B., M. Theimer, K. Petersen, A.J. Demers, M. Spreitzer, C. Hauser: Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System. SOSP 1995: 172-183
13. The Netflix Simian Army, <http://techblog.netflix.com/2011/07/netflix-simian-army.html>, Sept. 2011.
14. Windows Azure Cache, http://www.windowsazure.com/en-us/documentation/services/cache/

## A. CORRECTNESS PROOF OF THE GLOBAL SINGLE-INSTANCE PROTOCOL

We will prove the correctness of the protocol described in Section 3.2 in two failure models. In the first model, we assume there are no communication failures. That is, if a cluster is not responding to messages that are sent to it, then it has failed and is taking no other actions. In particular, this means there are no network partitions. In the second model, we allow communications failures. 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.

The goal of the protocol is to disallow two instantiations of the same object. Assuming communications failures do not occur, the protocol ensures that at most one cluster can be in the Owned or Doubtful state. If communications failures can occur, the protocol ensures that at most one cluster can be in the Owned state, but more than one can be in the Doubtful state. We prove each case in turn.

**Proposition 1:** In the absence of communications failures, the global single-instance protocol ensures that for a given object at most one cluster can have a directory entry for the object in the Owned or Doubtful state.

**Proof sketch:** Suppose two clusters, *c* and *d*, have such a directory entry. Each of them must have executed the following sequence of protocol steps:

Cluster *c* Cluster *d*

c1. Send Request d1. Send Request

c2. Wait for Replies d2. Wait for Replies

c3. Process replies and update state d3. Process replies and update state

to Owned or Doubtful to Owned or Doubtful

Assume that *c* landed in state Owned or Doubtful in step c3. We will show that *d* could not have landed in Owned or Doubtful in d3 too. We proceed with a case analysis, based on when *d* received the request sent by step c1.

In the first case, suppose *d* received the request from c1 before d1. In that case, *d* had no directory entry and replied (Pass, null). There are two sub-cases, depending on when *c* received *d’*s request from d1.

* Suppose *c* received *d’*s request from d1 after c3. Then *c* replied (Fail, *c*) to *d*, which caused *d* to set its state to [*c*, Cached] in step (ii), contradicting that *d* landed in d3 in state Owned or Doubtful.
* Suppose *c* received *d’*s request from d1 before c3. Then *c* was still in state [*c*, Requested]. If precedence(*c*) < precedence(*d*), then *c* updated its state to [*c*, Loser]. Therefore, when *c* processed replies to its request, it restarted the protocol and did not set its state to Owned or Doubtful. If precedence(*d*) < precedence(*c*), then it replied (Fail, null) to *d*, in which case *d* performed step (iii) and restarted the protocol. Thus, *d* did not reach step c3 to set its state to Owned or Doubtful.

In the second case, *d* received the request from c1 after d1 but before d3. Thus, *d* was in state [*d*, Requested] when it received the request. There are two sub-cases:

* If precedence(*c*) < precedence(*d*), then *d* replied to *c* with (Fail, null), in which case *c* performed step (iii) and restarted the protocol. Thus, *c* did not reach step c3 to set its state to Owned or Doubtful.
* If precedence(*d*) < precedence(*c*), then *d* replied (Pass, null) and set its state to [*d*, Loser]. Therefore, *d* was forced to restart the protocol. So it did not reach step d3 and set its state to Owned or Doubtful.

In the third case, *d* received the request from c1 after d3. So *d* was in the Owned or Doubtful state and replied (Fail, *d*) to *c’s* request. Therefore, *c* executed step (ii) and updated its state to [*e*, Cached] for some *e*. Thus, *d* is not in state Owned or Doubtful. This completes the case analysis and proves the result.

**Proposition 2:** Even in the presence of communications failures, the global single-instance protocol ensures that for a given object at most one cluster can have a directory entry for the object in the Owned state.

**Proof Sketch:** The proof is essentially the same as for Proposition 1, with additional cases when a cluster failed to receive a message it expected. Suppose two clusters, *c* and *d*, have such a directory entry. Each of them must have executed the following sequence of protocol steps:

Cluster *c* Cluster *d*

c1. Send Request d1. Send Request

c2. Receive Replies d2. Receive Replies

c3. Process replies and update state d3. Process replies and update state

to Owned or Doubtful to Owned or Doubtful

Assume that *c* landed in state Owned in step c3. We will show that *d* could not have landed in Owned in d3 too. We proceed with a case analysis, based on when *d* received the request sent by step c1.

If *d* did not receive *c’*s request, then *c* could not have received a reply from *d*. Hence, *c* could not have set its state to [*c*, Owned] in step (i).

Suppose *d* received the request from c1 before d1. In that case, *d* had no directory entry and replied (Pass, null). If *c* did not receive *d’*s reply, then it could not have set its state to [*c*, Owned] in step (i). So suppose *c* did receive *d’*s reply.

* If *c* received *d’*s reply after c3, then *c* replied (Fail, *c*) to *d*. If *d* received the reply, then it set its state to [*c*, Cached] in step (ii), and therefore did not land in state Owned in d3. If *d* did not receive *c’*s reply, then it could not have set its state to [*d*, Owned] in step (i).
* If *c* received *d’*s request from d1 before c3, then *c* was still in state [*c*, Requested].
  + If precedence(*c*) < precedence(*d*), then *c* updated its state to [*c*, Loser]. Therefore, when *c* processed replies to its request in c3, it restarted the protocol and did not set its state to Owned.
  + If precedence(*d*) < precedence(*c*), then *c* replied (Fail, null) to *d*. If *d* received the reply, then it performed step (iii) and restarted the protocol. Thus, *d* did not reach step c3 to set its state to Owned. If *d* did not receive *c’*s reply, then it could not have set its state to [*d*, Owned] in step (i).

Suppose *d* received the request from c1 after d1 but before d3. Thus, *d* was in state [*d*, Requested] when it received the request.

* If precedence(*c*) < precedence(*d*), then *d* replied to *c* with (Fail, null). If *c* received *d’*s reply, it performed step (iii) and restarted the protocol, and hence did not set its state to [*c*, Owned], a contradiction. If it did not receive *d’*s reply, then it could not have set its state to [*c*, Owned] in step (i).
* If precedence(*d*) < precedence(*c*), then *d* replied (Pass, null) and set its state to [*d*, Loser]. Therefore, *d* was forced to restart the protocol. So it did not reach step d3 and set its state to Owned.
* Suppose *d* received the request from c1 after d3. Thus, *d* was in the Owned state and replied (Fail, *d*) to *c’s* request. If *c* received the reply, it executed step (ii), updated its state to [*e*, Cached] for some *e*, and did not set its state to Owned. If *c* did not receive *d’*s reply, then it could not have set its state to [*c*, Owned] in step (i).

This completes the case analysis and proves the result.

1. All work was done at Microsoft Research. Current affiliations of authors’ no longer at Microsoft: Natacha Crooks, University of Texas at Austin; Jose Faleiro, Yale University; Alok Kumbhare, University of Southern California; Vivek Shah, University of Copenhagen; Adriana Szekeres, University of Washington [↑](#footnote-ref-1)