Trafalgar: Docs for Orleans Devs

Version 10/23/2015

# Overview

The goal of the Trafalgar project is to provide better service latency and fault tolerance for globally operating Orleans services. This is achieved by connecting multiple Orleans deployments in one or more regions to act as a single service. The main technical components of the Trafalgar project are as follows:

1. **Multi-cluster Network**. Connects multiple Orleans clusters. It is a network that stores and propagates various information between clusters, such as gateway IP addresses. It is also used for broadcasting administrative configuration actions, such as dynamically joining or removing a cluster to/from the multi-cluster without stopping the service.
2. **Single-instance Grain**. A type of a Orleans rain for which we coordinate with other clusters to ensure that there is (eventually) no more than a single instance of the grain in all clusters. The programming model is identical to regular grains, because all grain calls are automatically routed to the cluster where the grain resides.
3. **Queued Grains**. A new variant of Grain<T> that maintains a local cache of the grain state and synchronizes it with a remote primary in the background. It uses queueing, asynchrony, and batching, which can considerably improve the grain performance. Queued grains are useful both for single-cluster and multi-cluster scenarios.
4. **Replication Providers**. Queued grains maintain the illusion of a single remote primary, for the sake of a simple programming model. Under the hood, however, there is room for a wide variety of consistency and replication protocols with various tradeoffs. Any replication protocol can be implemented as a *replication provider*. Each queued grain class can be independently configured to use a certain replication provider.

## Audience

This document is written for developers that are familiar with Orleans and its terminology. We thus refrain from explaining basic Orleans concepts such as grains or storage providers, or how Orleans clusters work. We do have another document (with largely overlapping content) targeted at a general research audience.

## Driving Motivation: Availability and Performance

All of the design choices are motivated by the desire to allow programmers to achieve high availability and good performance despite the high latency and unreliability of cross-datacenter communication. For example, if some data center goes down, or some communication link is broken, all parts of the system can tolerate the situation and keep functioning:

* The *multi-cluster network* is designed to be highly resilient: there is no single point of failure because of its gossip-style propagation of information.
* The *single-instance grain* favors availability over consistency. Thus, it is always possible to activate a single-instance grain. It means that there may temporarily exist more than one activation until communication is restored and the duplicate grain is deactivated.
* *Queued grains* that cannot reach the primary, or can reach it only slowly or sporadically, can nevertheless always read the locally cached state, and enqueue updates to the grain state. Once communication is restored, the queued updates are applied to the primary and the local cache is refreshed.
* *Replication providers* can provide any of the classic failure-tolerant replication protocols, such as viewstamp replication.

## Document Organization

Multi-Cluster Network

Queued

Grains

Single-Instance Grains

Replication Providers

We cover the four technical components mentioned above in four corresponding chapters. The logical dependencies of these chapters are shown on the right.

# Multi-cluster Network

When deploying multiple Orleans clusters, it is important that they can communicate, that they can discover each other, and that we can perform administrative actions to join new clusters and remove old clusters. These concerns are separated into multiple independent layers.

## Physical Network

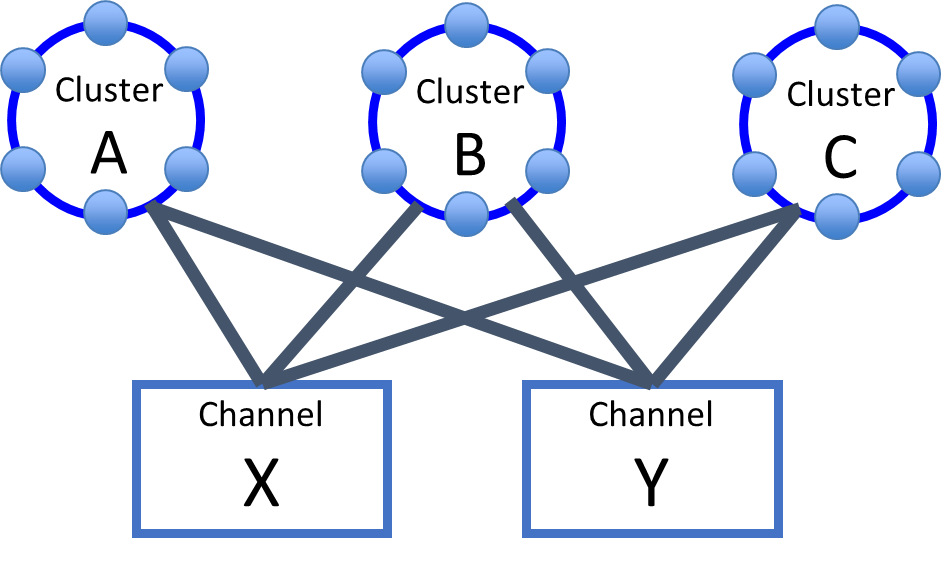
We assume that the network is configured in such a way that all Orleans nodes can directly send a message to any other Orleans node via its regular internal TCP endpoint, regardless of what cluster or datacenter. This guarantees best performance, but typically requires some work to set up the network correctly (a problem which is completely invisible to Orleans).

For example, on Windows Azure, we can achieve a fully connected network by creating VNETs within each region, and then connecting those VNETs using gateways. Then, any Orleans deployments that are started within those VNETs can directly communicate.

## Multi-Cluster Network

The Multi-Cluster network is a component (built on top of the physical network) to propagate information robustly between all clusters. It is used for exchanging routing information and for dynamic configuration purposes.

The multi-cluster network contains any number of **Clusters** and any number of **Channels**. When a cluster is started, it is configured to use one or more channels.



Our only channel type right now is an Azure Storage Table, but we plan to support more channel types. The idea is to use storage accounts in different regions for the channels X and Y, so that they fail independently. The more channels, the more reliable the propagation.

The network uses gossiping to pairwise synchronize information across all links. This means that all information propagates along all edges in the graph. Thus, there is no single point of failure. Note that we plan to have clusters also communicate directly, without going through the channel, once they have discovered each other’s IP addresses (via some channel), but this is not implemented yet.

It is o.k. if not all clusters use the same list of channels. This makes it possible to change all the clusters and all the channels over time without ever stopping the service.

### Configuration of the Multi-Cluster Network

To configure the multi-cluster network, add a section to the Orleans configuration file. For example, for the cluster **A** shown in the picture above, we may specify

<MultiClusterNetwork ClusterId="A">

<GossipChannel Type="AzureTable" ConnectionString="storageaccount\_X" />

<GossipChannel Type="AzureTable" ConnectionString="storageaccount\_Y" />

</MultiClusterNetwork>

It is safe to use the same channel for multiple services, as long as these services have their own, unique ServiceId parameter (part of the global configuration).

## Configuration Administration

An administrator must issue configuration changes by injecting them into the multi-cluster network, one at a time. Each new configuration consists of a list of cluster ids that form the multi-cluster, and must have a UTC timestamp that is larger than the timestamp of all previous configurations.

There are two main reasons why we rely on an administrator to configure the multi-cluster, instead of just working with some automatically determined list of currently active clusters:

1. Most communication protocols do not work correctly under circumstances where clusters chaotically appear, disappear, and re-appear. For example, the single-instance protocol could erroneously create multiple instances without detecting the problem.
2. Adding or removing a cluster from the multi-cluster is usually a decision that needs to be coordinated within some larger context. For example, it may be part of a routine maintenance or code updating. Almost always, a change in the multi-cluster configuration is just one part of a process that may involve steps that are not directly under the control of Orleans (e.g. creating, deleting, updating of deployments, or changing the routing of user load).

We pose some restrictions on the injection of new configurations that an administrator must follow:

* Each new configuration may add a number of clusters, or remove a number of clusters (but not both at the same time).
* An administrator should not issue a new configuration while a previous configuration change is still being processed.

These restrictions ensure that protocols such as the single-instance-protocol can correctly maintain mutual exclusion of activations even under configuration changes.

### Injecting a configuration

Timestamped multi-cluster configurations can be injected on any node in any cluster, using the Orleans Management Grain. Once injected, a configuration automatically spreads across the multi-cluster network, replacing any older configurations it encounters.

For example, suppose we want to inject a multi-cluster configuration for three clusters A, B and C. Let’s first create an enumerable containing the cluster ids:

var clusterlist = "A,B,C".Split(',');

To get a reference to the management grain, we use the following code:

var systemManagement = GrainClient.GrainFactory.GetGrain<IManagementGrain>(0);

Now we inject a new multicluster configuration:

systemManagement.InjectMultiClusterConfiguration(clusterlist,

"I am now adding cluster C [Bob]"));

The first argument is an enumerable of cluster ids, which is going to define the new multi-cluster configuration. The second argument is an (optional) comment string that can be used to tag configurations with arbitrary information, such as who injected them why.

### Injecting a default configuration

Sometimes (for example when doing local dev testing) calling the management grain explicitly can be a bit of a hassle. Thus, the multi-cluster network configuration supports an optional attribute which takes a comma-separated list of cluster ids:

<MultiClusterNetwork ... DefaultMulticluster="A,B,C" ...>

When starting the silo with this configuration:

* If no configuration is already present in the multi-cluster network, the given configuration is used with the current UTC time.
* If the multi-cluster network already contains a configuration, this attribute has no effect.

*Pitfall warning*: Persistent multi-cluster network channels (e.g. based on AzureTable) retain the last injected configuration unless they are deleted explicitly. In that case, specifying a DefaultMulticluster has no effect, because some configuration is already present.

### Behavior of non-member clusters

What happens if a cluster that is not part of the configured multi-cluster tries to access a global-single-instance grain or a queued grain backed by a replication provider? The exact results depend on the protocol.

In the current protocol implementation, the guarantees stay the same, except that:

* Instances of a single-instance grain on a non-joined cluster are not visible to the joined clusters. Thus, a joined cluster trying to access the grain will instantiate a new instance rather than route requests to the existing one.
* Instances of a replicated grain on a non-joined cluster do not receive notifications when the global state changes. Thus, the confirmed state can become very stale.

These issues can be completely avoided by following the procedures described in the next subsections.

### Procedure for adding a new cluster

1. Start a new Orleans cluster w/ channels configured. Once the silos have started, the cluster becomes visible in the multi-cluster network, but it is not part of the configuration yet.
2. Inject a configuration that contains the new cluster.
3. Wait for all clusters to acknowledge that they have processed the configuration.
4. Start routing user requests to the new cluster.

### Procedure for decommissioning an operational cluster

1. Stop routing user requests to the cluster.
2. Inject an admin-configuration that no longer contains the cluster.
3. Wait for all clusters to acknowledge that they have processed the configuration.
4. Shut down all silos.
5. Delete the deployment if desired.

Once a cluster has been removed in this way, it can be re-added by following the procedure for adding a new cluster.

### Procedure for decommissioning a failed or ailing cluster

To deal with a problematic cluster (say it has completely failed, or it is not performing correctly), the steps are the same as for decommissioning an operational cluster, except that we may resort to more drastic measures if there are problems along the way.

Concretely, if we get stuck at a step because the cluster is not responsive (e.g. it does not acknowledge the configuration change, or the silos do not shut down), the administrator may use heavier force and stop the cluster by whatever means.

## Multi-Cluster Gateways

All traffic between clusters is routed through these gateways.

Each cluster automatically determines a subset of its nodes to act as multi-cluster gateways. If a multi-cluster gateway fails, another node takes its place. The multi-cluster network automatically propagates gateway information, such as the status and the IP addresses of the gateways.

By default, the number of multi-cluster gateways per cluster is 8, but this can be changed in the configuration using an optional attribute:

<MultiClusterNetwork GlobalServiceId="MyGlobalService"

ClusterId="A"

NumMultiClusterGateways="16"

...

# Single-Instance Grains

Use this attribute to declare a single-instance grain:

[GlobalSingleInstance]

public class ItemGrain : Orleans.Grain, IItemGrain

{

…

}

Whenever this grain is called on some cluster, and its location is not already known and locally cached, all other clusters are contacted to see if they have an activation of this grain. If so, request and response are routed to/from that existing grain instance, and the location of that instance (i.e. cluster and node) is cached. Otherwise an instance is created locally.

## Programming API

All aspects of single-instance grains work exactly the same as regular Orleans Grains, except that calls to a single instance grain may be forwarded to a remote cluster (and thus experience much higher latency).

## Guarantees

If messages are not lost, or lost only infrequently, the single-instance protocol guarantees that there is at most one activation of a grain in all clusters. If communication is not functioning (for example, if a remote cluster has become unavailable), the protocol favors availability over consistency and creates a local instance.

Thus, it is possible for two instances to exist at the same time, in two clusters that cannot communicate. We still guarantee a single instance *eventually*: a periodic background process runs every 10 seconds to check if communication is available now and if there are any activations that need to be re-validated. If it discovers duplicate activations, it deactivates one of them.

# Queued Grains

Queued grains are a variation of persistent grains in Orleans. To create a grain implementation for a queued grain, simply derive from QueuedGrain<TGrainState> instead of Grain<TGrainState>.

Unlike regular grains, queued grains do not manage the grain state using the methods ReadStateAsync() and WriteStateAsync(). Instead, they use a **caching and queueing state interface** that exposes a choice of what version of the state to read, and allows writes to be queued and performed asynchronously.

Queued grains can be configured to use a variety of storage and replication providers, which may store state in memory and/or in one or more persistent storage locations. However, the programming model remains the same regardless of the underlying replication provider. We revisit this topic in the chapter on replication providers.

The new state interface is characterized by exposing (1) a locally cached state that is separate from the state of the remote primary, and (2) a queue for updating the state using update objects. We explain the various parts of this interface in the following subsections.

## The queue model

We use a mental model that visualizes the difference between what is stored locally, and what is stored remotely. It is shown on the right.

The most important thing to realize is that there is no longer a single state, but multiple copies of the state (blue boxes).

The closest thing to the “true state” is called the **Global State**, which is sometimes call the “primary” or “master”. The global state is remote and thus often slow to read and write. It may even be temporarily unreachable if we suffer from network partitions.

The global state is stored by a specified storage provider (or the default storage provider), or by a specified replication provider. Note that the global state may be just an illusion, created by a quorum of replicas.

To avoid paying for access to the global state all the time, we also store a local copy called the **Confirmed State**. The confirmed state is just a cached, and perhaps somewhat stale, version of the global state. It is guaranteed to match some past version of the global state.

Updates are performed by inserting update objects (green boxes) into an **Update Queue**. The update queue reflects updates that have been performed locally, but not yet confirmed.

### Automatic Propagation

Automatically, in the background, the replication provider is running a continuous propagation protocol to keep everything eventually consistent:

* *(Local -> Remote)* Updates in the queue are sent and applied to the global state. If needed, this process is automatically retried until successful.
* *(Remote -> Local)* The confirmed state is refreshed whenever the global state changes (with some delay depending on network conditions and configuration).

Updates are removed from the update queue as soon as they are part of the confirmed state, i.e. after they have been applied to the global state *and* the confirmed state has been updated to the latest global state.

### Robust Retry

Note that the Queued Grain *never gives up* when there are storage exceptions. Unlike the old state interface, which simply passes storage exceptions to the user, the queued grain keeps retrying until the queue is empty, or the silo is shut down. The retry logic is designed to correctly deal with (1) exceptions caused by e-tag failures due to concurrent writes, (2) exceptions thrown by failed read or write requests, and (3) exceptions thrown by write requests that were actually successful.

### Local Operations

The following properties and methods are the core of the new state interface. They all execute locally, without communication, and do not throw exceptions.

TGrainState ConfirmedState { get; }

int ConfirmedVersion { get; }

void EnqueueUpdate(IUpdateOperation<TGrainState> update);

IEnumerable<IUpdateOperation<TGrainState>> UnconfirmedUpdates { get; }

TGrainState TentativeState { get; }

The terminology corresponds to labels in the Queue Model diagram above:

* The ConfirmedState property is the most recently fetched copy of the global state, or a blank state if none has arrived yet.
* The ConfirmedVersion is the version number of the confirmed state.
* The EnqueueUpdate function inserts an update object into the queue.
* The UnconfirmedUpdates property is the current queue content.
* The TentativeState property returns an approximated view of the global state that is computed by combining (1) the confirmed state, and (2) the effect of all unconfirmed updates in the queue.

Using the TentativeState for queries is usually the right choice, since it hides the latency of confirming the update: when reading TentativeState, it looks like all updates in the queue are already applied.

### Update objects

An important part of the new interface is that update operations are objects (green boxes in the queue model diagram above). An update object must implement the interface

public interface IUpdateOperation<TGrainState>

{

void Update(TGrainState state);

}

An update object defines “how to update the state”, as a function. Updates can contain parameters, and must be serializable (since a replication provider may send updates to other clusters).

It is very important to follow this rule:

User code must never update the state directly, but must always do so indirectly by enqueueing update objects.

Modifying the state directly does not work correctly – the exact effects depend on the replication provider. Typically what happens is that the update is not saved to the global state and lost next time the confirmed state is updated.

## First Example: A Counter Grain

We now give a very simple example, a counter grain. For example, this could be a grain that maintains some sort of statistics.

The **grain interface** contains just two methods:

public interface ICounterGrain : IGrain

{

Task<int> Get();

Task Increment();

}

The **grain implementation** is likewise quite simple:

public class CounterGrain : QueuedGrain<CounterState>, ICounterGrain

{

public Task<int> Get()

{

return Task.FromResult(this.TentativeState.Count);

}

public Task Increment()

{

EnqueueUpdate(new IncrementOperation());

return TaskDone.Done;

}

}

The **grain state** is defined by the following class:

[Serializable]

public class CounterState : GrainState

{

public int Count { get; set; }

}

And the **update object** (for increment operations) is defined by the following class:

[Serializable]

public class IncrementOperation : IUpdateOperation<CounterState>

{

public void Update(CounterState state)

{

state.Count++;

}

}

## Second Example: A Chat Grain

Another typical example is a grain storing up to 100 chat messages.

The **grain interface** again contains just two methods:

public interface IChatGrain : IGrain

{

Task<IReadOnlyList<string>> GetMessages();

Task AddMessage(string message);

}

The **grain state** stores the messages. We use a default constructor to ensure the list is never null:

[Serializable]

public class ChatState : GrainState

{

public List<string> Messages { get; set; }

public ChatState() { Message = new List<string>(); }

}

The **grain implementation** reads and updates the list:

public class ChatGrain : QueuedGrain<ChatState>, IChatGrain

{

public Task<IReadOnlyList<string>> GetMessages()

{

return Task.FromResult(this.TentativeState.Messages);

}

public Task AddMessage(string message)

{

EnqueueUpdate(new MessageAddedEvent() { Content = message } );

return TaskDone.Done;

}

}

And the **update object** (for adding messages) is defined by the following class:

[Serializable]

public class MessageAddedEvent: IUpdateOperation<ChatState>

{

public string Content { get; set; }

public void Update(ChatState state)

{

state.Messages.Add(Content);

// remove oldest message if necessary

if (state.Messages.Count > 100)

Messages.RemoveAt(0);

}

}

## Third Example: A Blob Grain

A yet slightly different example is the blob grain, which stores a large monolithical byte array, such as a jpeg image.

The **grain interface** again contains just two methods:

public interface IBlobGrain : IGrain

{

Task<byte[]> Get();

Task Set(byte[] value);

}

The **grain state** is just a wrapper for the blob:

[Serializable]

public class BlobState : GrainState

{

public byte[] Value { get; set; }

}

The **grain implementation** reads and updates the blob:

public class BlobGrain : QueuedGrain<BlobState>, IBlobGrain

{

public Task<byte[]> Get()

{

return Task.FromResult(this.TentativeState.Value);

}

public Task Set(byte[] value)

{

EnqueueUpdate(new ValueChanged() { NewValue = value } );

return TaskDone.Done;

}

}

and the **update object** (for changing the blob content) is as follows:

[Serializable]

public class ValueChanged: IUpdateOperation<BlobState>

{

public byte[] NewValue { get; set; }

public void Update(BlobState state)

{

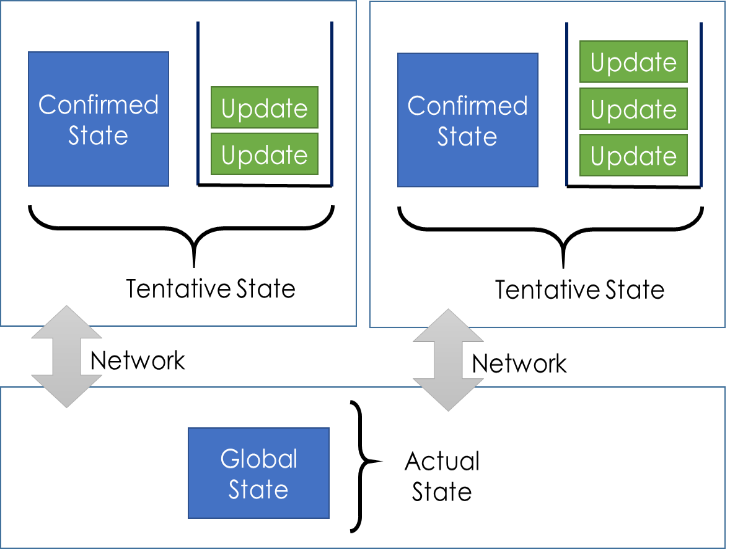
state.Value = NewValue;

}

}

## Conflicts

The global state always contains *the result of some sequence of update operations*. This guarantee, though simple, is strong enough to help us maintain desired data invariants on the global state. For example, the chat grain never contains more than 100 messages.

However, note that there is no guarantee that updates execute identically on the global state as they did on the local state! In a situation with multiple queued grains issuing updates to the same global state, updates may race. For example, the final order of chat messages may depend on such a race.

Races can be handled automatically because *an update object is a function, and therefore defines how to update any state, not just the state that was current at the time of the update*. Thus, the losing update can simply be applied to the newer global state.

In some sense, this means that the notion of “conflict resolution” is already built into the update function, since it can “re-evaluate” its effect once it is applied to the global state.

For example, for the counter grain, the increment operation succeeds even if the counter value has changed since the increment was originally issued. It simply increments the new value.

In general, we can encode complex conflict resolution logic inside the update function, by adding parameters to the operation (e.g. recording read sets) and tests (e.g. re-checking preconditions). However, in our experience, that is rarely necessary.

## Synchronization Operations

Sometimes, we want stronger guarantees than what we get from the core API – in other words, we **want to wait** as long as necessary, to get the latest global state, or to ensure an update commits.

The queued grain supports two synchronization operations, each returning a task:

// Returns a task that can be waited on to ensure all updates currently in the

// queue have been confirmed.

public Task CurrentQueueHasDrained();

// Enforces full synchronization with the global state. This both (a) drains

// all updates currently in the queue, and (b) retrieves the latest global state.

public Task SynchronizeNowAsync();

The intended use of these synchronization operations is to support grain operations that have stronger guarantees. For example, we can use them as follows to create stronger versions of the read and write operations on the counter:

public async Task Increment\_linearizable()

{

EnqueueUpdate(new IncrementedEvent());

await CurrentQueueHasDrained();

}

public async Task<int> Get\_linearizable ()

{

await SynchronizeNowAsync();

return ConfirmedState.Count;

}

Note that the relative placement of the synchronization is different for reads and writes (the synchronization comes before the read but after the write). The consistency guarantee achieved for these operations is **linearizability**, also known as single-copy-consistency. It means that the observed behavior is consistent with the read or write operation happening directly on the global state, sometime in between call and return.

Synchronization operations may block for an unlimited time, or may throw exceptions if there are problems with communication, such as when remote multi-clusters cannot be reached, or while primary storage is suffering an outage. Thus, to implement highly-available grains, it is best to avoid them altogether.

### Synchronize on Grain Activation

When a queued grain is activated, it tries to read the global state. However, by default, it does *not* wait for this state to arrive before allowing other grain methods to execute. This ensures queued grains can be activated even if the global state is unreachable. However, it may also lead to surprising results.

If, immediately after activation, grain operations read the confirmed state or the tentative state before the latest state has been received from storage, the confirmed state appears blank (is a freshly constructed state object). Note that technically, the “blank” state is the initial state of the global state, so our earlier statement remains true: the confirmed state is guaranteed to match some past version of the global state.

To enforce that a newly activated grain does not execute any grain methods before the latest global state is received, synchronize inside OnActivatedAsync:

public override async Task OnActivateAsync()

{

await base.OnActivateAsync();

await SynchronizeNowAsync();

}

## Notifications

It is often convenient to be notified when the confirmed state changes. For example, in a chat application, we may want to update the displayed messages whenever the state of the chat grain changes.

### Tracking the Confirmed State

We support subscription of listeners for changes of the confirmed state:

public bool SubscribeConfirmedStateListener(IConfirmedStateListener listener);

public bool UnSubscribeConfirmedStateListener(IConfirmedStateListener listener);

// A listener that can observe changes to the confirmed state.

public interface IConfirmedStateListener

{

// Gets called after the confirmed state has changed.

void OnConfirmedStateChanged();

}

Note the following particulars:

* There is no guarantee that OnConfirmedStateChanged is called exactly once per update, because receiving a single new state may correspond to a large number of applied updates.
* When OnConfirmedStateChanged is called, it does not always mean that the content of ConfirmedState has truly changed; some updates may have left the state unchanged, or several updates may have neutralized each other.

### Tracking the Tentative State

The confirmed state and the tentative state may change at different times and with different frequency.

TentativeState can change in response both to locally queued updates and to changes of ConfirmedState. Thus, such changes can be tracked by instrumenting all calls to EnqueueUpdate and by using the SubscribeConfirmedStateListener.

Usually, when a locally issued update becomes confirmed, the content of ConfirmedState changes while the content of TentativeState remains the same.

## Provider Configuration

Queued grains can either use a traditional storage provider, or a replication provider. If no storage or replication provider is specified, the default storage provider is used. For example, we can define three implementations of the counter grain with different configurations, based on the respective attributes:

public class CounterGrain1 : CounterGrain {

// uses default storage provider

}

[StorageProvider(ProviderName = "MyStorageProvider")]

public class CounterGrain2 : CounterGrain {

// uses MyStorageProvider

}

[ReplicationProvider(ProviderName = "MyReplicationProvider")]

public class CounterGrain3 : CounterGrain {

// uses MyReplicationProvider

}

## Diagnostics

Since the propagation of updates and states is happening automatically and continuously in the background, storage exceptions are not always visible to the grain code (the only calls that observe such exceptions are the explicit synchronization operations). For applications that want to detect problems with the automatic background propagation, we support a property to retrieve the last exception thrown:

public Exception LastException { get; }

Whenever communication with the global state fails with an exception, that exception is stored in LastException. Once communication succeeds (i.e. the automatic retry is successful), LastException is set to null.

## General Remarks

The queued grain can significantly improve performance in many cases (not only in geo-replicated scenarios) because of the built-in batching, the asynchronous writing, and the approximate reading. Also, it automatically handles common complications (such as failing e-tags). This robustness and performance comes at the expense of more boilerplate: each update operation must be encapsulated as an object, instead of directly updating the state object.

Queued grains make it very clear which operations require cross-multi-cluster communication, and which ones can complete locally. It thus enables programmers to design highly-available grain operations (where availability is favored over consistency), while still allowing strongly consistent reads and updates (where consistency is favored over availability).

# Replication Providers

There are many options for configuring the storage of a grain. For example, the grain state may exist in RAM only, in RAM and in a primary persistent storage, or in RAM and in multiple persistent storage replicas.

Shared  
Storage

Storage  
Replica

Storage

Replica

Replica

in RAM

Replica

in RAM

Replica

in RAM

Replica

in RAM

Replica

in RAM

Replica

in RAM

From left to right, the diagram above shows (1) an **in-memory replication provider**, (2) a **shared-storage replication provider**, and (3) a **replicated-storage replication provider**. The diagram in the middle represents what we get when using a standard storage provider. We now discuss the various replication providers we are working on.

## By Category

### Shared-Storage Replication Providers

When using a standard storage provider, the runtime automatically creates a wrapping replication provider for it. Alternatively, we can get the same effect by explicitly configuring a replication provider that wraps a storage provider:

<Provider Type="Orleans.Providers.Replication.SharedStorageProvider"

Name="SharedStorage"

GlobalStorageProvider="MyStorageProvider"/>

### In-Memory Replication Providers

We have three different versions of the in-memory providers under development, with different tradeoffs.

1. The **State-Based Memory Provider** (coming soon) is similar to the shared storage provider, but uses an in-memory copy located at a leader replica in one of the clusters. The leader is statically determined using a uniform hash of the grain id and the multi-cluster configuration.
2. The **Operation-Based Memory Provider** (coming soon) is similar to the state-based memory provider, but more efficient for grains whose update operations are much smaller in size than the state object.
3. The **Viewstamp-Replication Provider** (availability TBD) is similar to the operation-based memory provider, but more reliable as it stores the state not in a single replica, but in a quorum of replicas.

### Replicated-Storage Replication Providers

The **Pileus storage provider** (availability TBD) supports the use of multiple persistent storage locations, with the advantage that permanent loss of a storage location can be compensated by reconfiguration.