Applied Actor Model with Orleans

Hatay Tuna, Christian Martinez
Microsoft

v1.1

Introduction

This paper is about using Orleans, a platform built around the actor model to solve real world problems at cloud scale. Orleans originated in Microsoft Research but it is not some theoretical experiment. It is a proven platform that has operated at cloud scale powering such solutions as HALO and we are confident that it can help programmers in all industries (not just cool ones like gaming) overcome many common distributed programming challenges.

Orleans is an actor model based system but this will not be a paper on the history of the actor model, an analysis of its strengths as a distributed paradigm or a comparison of actor model implementations. That kind of abstract discussion can be found in abundance elsewhere. However we will provide a brief description of actor model and Orleans for completeness.

This paper is intended to be a practical paper about practical problems. By the end of this paper you should be able to understand how you can use Orleans to build solutions be they "enterprise" or "cloud" or just methods that solve problems, make money and keep your customers happy.

So, those who are comfortable at this point with the actor model and Orleans history or who don't care and just want to learn how Orleans can help them solve real problems should immediately skip to the *Patterns* section below.

Actor Model - New? No. Powerful? Yes.

The paper by Hewitt et al. that is the origin of the actor model was published in 1973 yet it is only comparatively recently that the actor model has been gaining more attention as a means of dealing with concurrency and complexity in distributed systems.

The actor model supports fine-grain individual objects—actors—that are isolated from each other. They communicate via asynchronous message passing, which enables direct communications between actors. An actor executes with single-thread semantics. Coupled with encapsulation of the actor's state and isolation from other actors, this simplifies writing highly parallel systems by removing concurrency concerns from the actor's code. Actors are dynamically created on the pool of available hardware resources.

Erlang [1]is the most popular implementation of the actor model. Developers have started rediscovering the actor model, which stimulated renewed interest in Erlang and creation of new Erlang-like solutions: Scala actors, Akka, DCell.

Brief Look at Orleans

Orleans is an implementation of the actor model that borrows some ideas from Erlang and distributed objects systems, adds a layer of actor indirection, and exposes them in an integrated, programming model.

The main benefits of Orleans are: 1) **developer productivity**, even for non-expert programmers; and 2) **transparent scalability by default** with no special effort from the programmer. Orleans is a set of

[1] There is some debate on whether Erlang implements the Actor model, or simply bears a resemblance to some of its features, such as its approach to concurrency. Suffice it to say that many observers view Erlang as an Actor-based system, so we cite it here as related work.

.NET libraries and tools that make development of complex distributed applications much easier and make the resulting applications scalable by design. We expand on each of these benefits below.

The Orleans programming model raises productivity of both expert and non-expert programmers by providing the following key abstractions, guarantees and system services.

- *Familiar object-oriented programming (OOP) paradigm***.** Actors are .NET classes that implement declared .NET actor interfaces with asynchronous methods and properties. Thus actors appear to the programmer as remote objects whose methods/properties can be directly invoked. This provides the programmer the familiar OOP paradigm by turning method calls into messages, routing them to the right endpoints, invoking the target actor's methods and dealing with failures and corner cases in a completely transparent way.
- *Single-threaded execution of actors.* The Orleans runtime guarantees that an actor never executes on more than one thread at a time. Combined with the isolation from other actors, the programmer never faces concurrency at the actor level, and hence never needs to use locks or other synchronization mechanisms to control access to shared data. This feature alone makes development of distributed applications tractable for non-expert programmers.
- *Transparent activation.* The Orleans runtime activates an actor as-needed, only when there is a message for it to process. This cleanly separates the notion of logical creation of an actor, which is visible to and controlled by application code, and physical activation of the actor in memory, which is transparent to the application. Orleans is similar to virtual memory in that it decides when to "page out" (deactivate) or "page in" (activate) an actor; the application has uninterrupted access to the full "memory space" of logically created actors, whether or not they are in physical memory at any particular point in time. Transparent activation enables dynamic, adaptive load balancing via placement and migration of actors across the pool of hardware resources.
- *Location transparency.* An actor reference (proxy object) that the programmer uses to invoke the actor's methods or pass to other components only contains the logical identity of the actor. The translation of the actor's logical identity to its physical location and the corresponding routing of messages are done transparently by the Orleans runtime. Application code communicates with actors oblivious to their physical location, which may change over time due to failures or resource management, or because an actor is deactivated at the time it is called.
- *Transparent integration with persistent store*. Orleans allows for declarative mapping of actors' in-memory state to persistent store. It synchronizes updates, transparently guaranteeing that callers receive results only after the persistent state has been successfully updated.
- *Automatic propagation of errors.* The Orleans runtime automatically propagates unhandled errors up the call chain with the semantics of asynchronous and distributed try/catch. As a result, errors do not get lost within an application. This allows the programmer to put error handling logic at the appropriate places, without the tedious work of manually propagating errors at each level.

The Orleans programming model is designed to guide programmers down a path of likely success in scaling their application or service through several orders of magnitude. This is done by incorporating proven best practices and patterns, and providing an efficient implementation of

lower level system functionality. Here are some key factors that enable scalability and performance of Orleans applications.

- *Implicit fine grain partitioning* of application state. By using actors as directly addressable entities, programmers implicitly breaks down the overall state of their applications. While the Orleans programming model does not prescribe how big or small an actor should be, in most cases it makes sense to have a relative large number of actors – millions or more – with each representing a natural entity of the application, such as a user account, a purchase order, etc. With actors being individually addressable and their physical location abstracted away by the runtime, Orleans has enormous flexibility in balancing load and dealing with hot spots in a transparent and generic way without any thought from the application developer.
- *Adaptive resource management***.** With actors making no assumption about locality of other actors they interact with and because of the location transparency, the Orleans runtime can manage and adjust allocation of available HW resources in a very dynamic way by making fine grain decisions on placement/migration of actors across the compute cluster in reaction to load and communication patterns without failing incoming requests. By creating multiple replicas of a particular actor the runtime can increase throughput of the actor if necessary without making any changes to the application code.
- *Multiplexed communication.* Actors in Orleans have logical endpoints, and messaging between them is multiplexed across a fixed set of all-to-all physical connections (TCP sockets). This allows the Orleans runtime to host a very large number (millions) of addressable entities with zero OS overhead per actor. In addition, activation/deactivation of an actor does not incur the cost of registering/unregistering a physical endpoint, such as a TCP port or a HTTP URL.
- *Efficient scheduling.* The Orleans runtime schedules execution of a large number of single-threaded actors across a custom thread pool with a thread per physical processor core. With actor code written in the non-blocking continuation-based style (a requirement of the Orleans programming model) application code runs in a very efficient “cooperative” multi-threaded manner with no contention. This allows the system to reach high throughput and run at very high CPU utilization (up to 90 + %) with great stability. The fact that a growth in the number of actors in the system and the load does not lead to additional threads or other OS primitives helps scalability of individual nodes and the whole system.
- *Explicit asynchrony.* The Orleans programming model makes the asynchronous nature of a distributed application explicit and guides programmers to write non-blocking asynchronous code.This enables a large degree of distributed parallelism and overall throughput without the explicit use of multi-threading.

Patterns

In this section, we will focus on a class of patterns and associated scenarios we harnessed during our engagements with customers. These patterns represent classes of problems that are applicable to a wide range of solutions our customers are building on Microsoft Azure.

While the scenarios are based on real cases we have stripped out most of the domain-specific concerns to make the patterns clearer for the reader. You may find that much of the sample code is

simple or obvious. We are including that code for the sake of completeness and not because it's anything particularly clever or impressive.

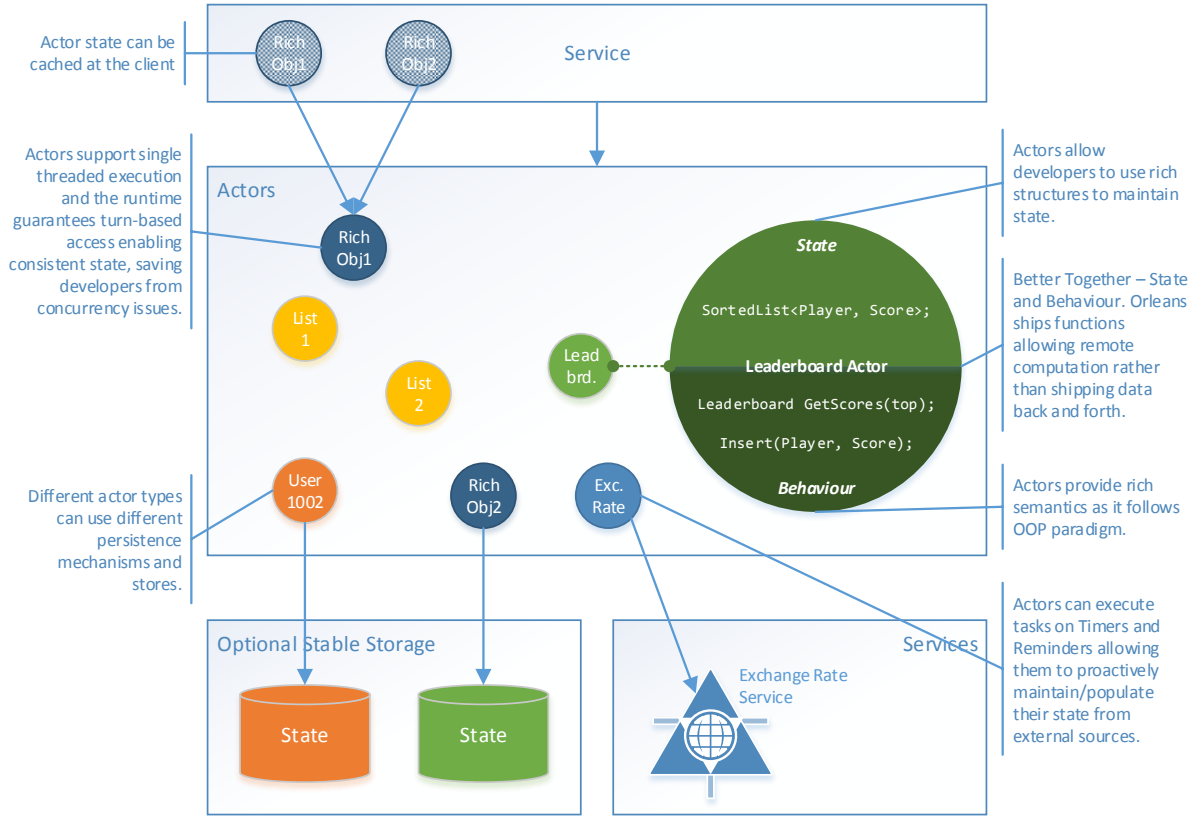
The patterns presented in this paper are not intended to be comprehensive or canonical—some developers might solve the same problem or pattern a different way than we present.

Pattern: Smart Cache

The combination of a web tier, caching tier, storage tier, and occasionally a worker tier are pretty much the standard parts of today's applications. The caching tier is usually vital to performance and may, in fact, be comprised of multiple tiers itself.

Many caches are simple key-value pairs while other systems like Redis that are used as caches offer richer semantics. Still, any special, caching tier will be limited in semantics and more importantly it is yet another tier to manage.

What if instead, objects just kept state in local variables and these objects can be snapshotted or persisted to a durable store on demand or using a timer? Furthermore, rich collections such as lists, sorted sets, queues, and any other custom type for that matter are simply modelled as member variables and methods.



Take leader boards as an example—a Leaderboard object needs to maintain a sorted list of players and their scores so that we can query it. For example for the "Top 100 Players" or to find a player's position in the leader board relative to +- N players above and below him/her. A typical solution with traditional tools would require 'GET'ing the Leaderboard object (collection which supports inserting a new tuple<Player, Points> named Score), sorting it, and finally 'PUT'ing it back to the cache. We would probably LOCK (GETLOCK, PUTLOCK) the Leaderboard object for consistency.

Let's have an actor-based solution where state and behaviour are together. There are two options:

- Implement the Leaderboard Collection as part of the grain,
- Or use the grain as an interface to the collection that we can keep in a member variable.

First let's have a look at what the grain interface may look like:

Smart Cache – Leaderboard Interface

```
public interface ILeaderboard : IGrain
{
    // Updates the leaderboard with the score - player, points
    Task UpdateLeaderboard(Score score);

    // Returns the Top [count] from the leaderboard e.g., Top 10
    Task<List<Score>> GetLeaderboard(int count);

    // Returns the specific position of the player relative to other players
    Task<List<Score>> GetPosition(long player, int range);
}
```

Next, we implement this interface and use the latter option and encapsulate this collection's behaviour in the actor:

Smart Cache – Leaderboard

```
public class Leaderboard : GrainBase, ILeaderboard
{
    // Specialised collection, could be part of the actor
    private LeaderboardCollection _leaderboard;

    public Task UpdateLeaderboard(Score score)
    {
        _leaderboard.UpdateLeaderboard(score);
        return TaskDone.Done;
    }

    public Task<List<Score>> GetLeaderboard(int count)
    {
        // Return top N from Leaderboard
        return Task.FromResult(_leaderboard.GetLeaderboard(count));
    }

    public Task<List<Score>> GetPosition(long player, int range)
    {
        // Return player position and other players in range from Leaderboard
        return Task.FromResult(_leaderboard.FindPosition(player, range));
    }
}
```

We are keeping the state in an internal variable called `_leaderboard` and providing methods to change the state. No data shipping, no locks, just manipulating remote objects in a distributed runtime, servicing multiple clients as if they were single objects in a single application servicing only one client.

Here is the sample client:

Smart Cache – Leaderboard

```
// Get reference to Leaderboard
var leaderboard = LeaderboardFactory.GetGrain(1001);

// Update Leaderboard with dummy players and scores
await leaderboard.UpdateLeaderboard(new Score() { Player = 1, Points = 500 });
await leaderboard.UpdateLeaderboard(new Score() { Player = 2, Points = 100 });
await leaderboard.UpdateLeaderboard(new Score() { Player = 3, Points = 1500 });

// Finally, Get the Leaderboard. 0 represents ALL, any other number > 0 represents TOP N
var result = await leaderboard.GetLeaderboard(0);
```

The output looks like this:

Smart Cache – Leaderboard Example

```
Player = 3 Points = 1500
Player = 1 Points = 500
Player = 2 Points = 100
```

It may feel like the example above could create a bottleneck in the Leaderboard instance. What if, for instance, we are planning to support hundreds and thousands of players? One way to deal with that might be to introduce stateless aggregators that would act like a buffer—hold the partial scores (say subtotals) and then periodically send them to the Leaderboard actor, which can maintain the final Leaderboard. We will discuss this “aggregation” technique in more detail later.

Also, we do not have to consider mutexes, semaphores, or other concurrency constructs traditionally required by correctly behaving concurrent programs.

Below is another cache example that demonstrates the rich semantics one can implement with grains. This time we implement the logic of the Priority Queue (lower the number, higher the priority) as part of the Grain implementation.

The interface for `IJobQueue` looks like below:

Smart Cache – Job Queue Interface

```
public interface IJobQueue : IGrain
{
    Task Enqueue(Job item);
    Task<Job> Dequeue();
    Task<Job> Peek();
    Task<int> GetCount();
}
```

We also need to define the Job item:

Smart Cache – Job

```
public class Job : IComparable<Job>
{
    public double Priority { get; set; }
    public string Name { get; set; }

    public override string ToString()
    {
        return string.Format("Job = {0} Priority = {1}", Name, Priority);
    }

    public int CompareTo(Job other)
    {
        return Priority.CompareTo(other.Priority);
    }
}
```

Finally, we implement the `IJobQueue` interface in the grain. Note that we omitted the implementation details of the priority queue here for clarity. A sample implementation can be found in the accompanying samples.

Smart Cache – Job Queue

```
public class JobQueue : GrainBase, IJobQueue
{
    private List<Job> data;

    public override Task ActivateAsync()
    {
        data = new List<Job>();
        return base.ActivateAsync();
    }

    public Task Enqueue(Job item)
    {
        // this is where we add to the queue

        ...

        return TaskDone.Done;
    }

    public Task<Job> Dequeue()
    {
        // this is where we remove from the head of the queue

        ...

        return Task.FromResult(frontItem);
    }

    public Task<Job> Peek()
    {
        // this is where we peek at the head of the queue

        ...

        return Task.FromResult(frontItem);
    }
```

```
        public Task<int> GetCount()
        {

            // this is where we return the number of items in the queue

            return Task.FromResult(data.Count);
        }
```

The output looks like this:

Smart Cache – Job Queue Example

```
Job = 2 Priority = 0.0323341116459733
Job = 3 Priority = 0.125596747792138
Job = 4 Priority = 0.208425460480352
Job = 0 Priority = 0.304352047063574
Job = 8 Priority = 0.415597594070992
Job = 7 Priority = 0.477669881413537
Job = 5 Priority = 0.525898784178262
Job = 9 Priority = 0.921959541701693
Job = 6 Priority = 0.962653734238191
Job = 1 Priority = 0.97444181375878
```

In the samples above, Leaderboard and JobQueue, we used two different techniques:

- In the Leaderboard sample we encapsulated a Leaderboard object as a private member variable in the grain and merely provided an interface to this object – both to its state and functionality.
- On the other hand, in the JobQueue sample we implemented the grain as a priority queue itself rather than referencing another object defined elsewhere.

Grains provide flexibility for the developer to define rich object structures as part the grains or reference object graphs outside of the grains.

In caching terms grains can *write-behind* or *write-through,* or we can use different techniques at a member variable granularity. In other words, we have full control over what to persist and when to persist. We don't have to persist transient state or state that we can build from saved state.

We also have full control over where to persist; for example, we can partition many grains across multiple stores—say a sharded storage tier—based on one or more properties. We can also use different storage mechanisms and stores for different types of actors (more on this pattern later).

And how about populating these grain caches then? There are number of ways to achieve this. Grains provide virtual methods called `ActivateAsync()` and `DectivateAsync()` to let us know when an instance of the grain is activated and deactivated. Note that the grain is activated on demand when a first request is sent to it.

We can use `ActivateAsync()` to populate state on-demand as in *read-through*, perhaps from an external stable store. Or we can populate state on a timer, say an Exchange Rate grain that provides the conversion function based on the latest currency rates. This grain can populate its state from an

external service periodically, say every 5 seconds, and use the state for the conversion function. See the example below:

Smart Cache – Rate Converter

```
...

private List<ExchangeRate> _rates;
private IOrleansTimer _timer;

public Task Activate()
{
   // registering a timer that will live as long as the grain...
   _timer = this.RegisterTimer((obj) =>
   {
      Console.WriteLine("Refreshing rates...");
      return this.RefreshRates(); // call to external service/source to retrieve exchange rates
   },
   null,
   TimeSpan.FromSeconds(0), // start immediately
   TimeSpan.FromSeconds(5)); // refresh every 5 seconds

   return TaskDone.Done;
}

public Task RefreshRates()
{
    // this is where we will make an external call and populate rates
    return TaskDone.Done;
}
```

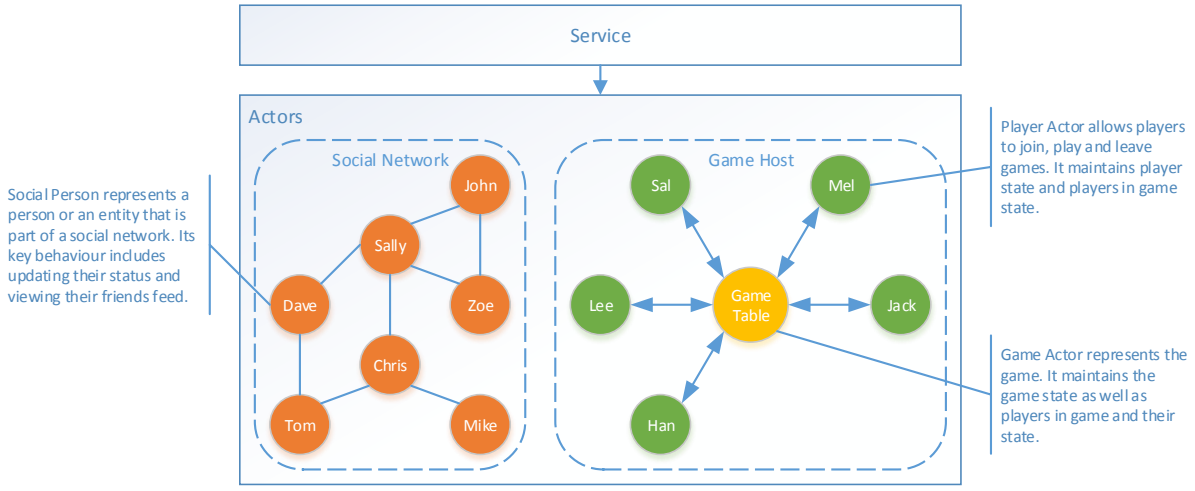
Essentially Smart Cache provides:

- High throughput/low latency by service requests from memory.
- Single-instance routing and single-threaded serialization of requests to an item with no contention on persistent store.
- Semantic operations, for example, `Enqueue(Job item)`.
- Easy-to-implement write-through or write-behind.
- Automatic eviction of LRU (Least Recently Used) items (resource management).
- Automatic elasticity and reliability.

It could be argued that this is not actually a pattern per se, but just vanilla Orleans that is at the heart of the virtual actor model; simple yet powerful, especially in the familiar and rich .NET/C# environment.

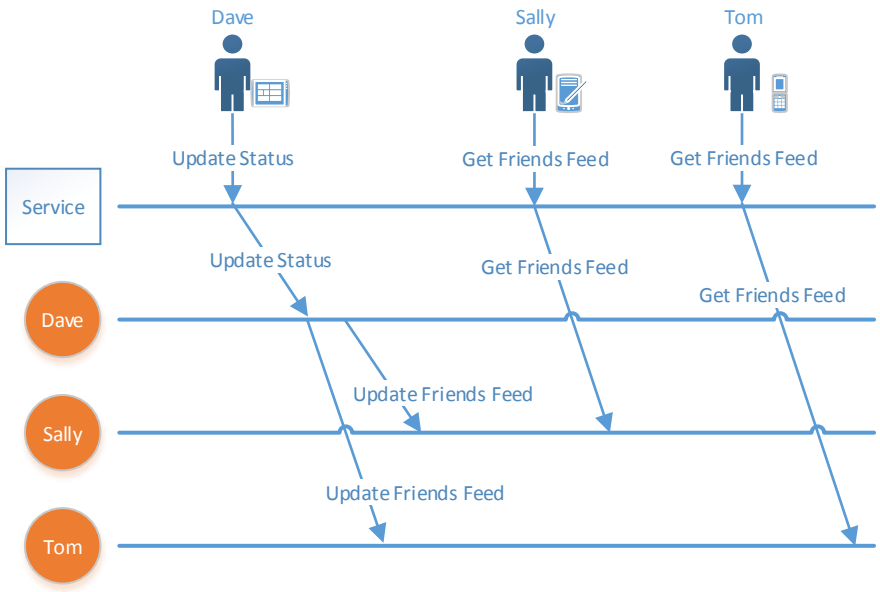
Pattern: Distributed Networks and Graphs

Orleans is a natural fit for modelling complex solutions involving relations and modelling those relations as objects.



As the diagram illustrates it is straightforward to model a user as a grain instance (node in the network). For example, the “Friends Feed” (sometimes referred as the "follower" problem) allows users to view status updates from people they are connected to, similar to how Facebook and Twitter work.

The Actor model provides flexibility to approach the materialisation problem. We can populate the Friends Feed at event time, updating the Friends Feed of all my friends at the moment an update is posted, as illustrated below:



Sample code populating Friends Feed:

Smart Cache – Social Network Friends Feed (event time)

```
public interface ISocialPerson : IGrain
{
    Task AddFriend(long person);
    Task RemoveFriend(long person);
    Task<List<SocialStatus>> GetFriendsFeed();
    Task<SocialStatus> GetStatus();
    Task<List<SocialStatus>> GetMyFeed();
    Task UpdateStatus(string status);
```

```
        Task UpdateFriendFeedAsync(SocialStatus status);
    }

    public class SocialPerson : GrainBase, ISocialPerson
    {
        private string _name; // my name
        private List<long> _friends; // list of my friends' IDs
        private List<SocialStatus> _friendsFeed; // my friends feeds
        private List<SocialStatus> _myFeed; // this is my feed, all my status updates
        private SocialStatus _lastStatus; // this is my last update

        public override Task ActivateAsync()
        {
            CreateOrRestoreState();
            return base.ActivateAsync();
        }

        public Task AddFriend(long person)
        {
            _friends.Add(person);
            return TaskDone.Done;
        }

        public Task RemoveFriend(long person)
        {
            _friends.Remove(person);
            return TaskDone.Done;
        }

        public Task<List<SocialStatus>> GetFriendsFeed()
        {
            return Task.FromResult(_friendsFeed);
        }

        public Task UpdateStatus(string status)
        {
            _lastStatus = new SocialStatus()
                { Name = _name, Status = status, Timestamp = DateTime.UtcNow };
            _myFeed.Add(_lastStatus);

            var taskList = new List<Task>();

            foreach(var friendId in _friends)
            {
                var friend = SocialPersonFactory.GetGrain(friendId);
                taskList.Add(friend.UpdateFriendFeedAsync(_lastStatus));
            }

            return Task.WhenAll(taskList);
        }

        public Task UpdateFriendFeed(SocialStatus status)
        {
            _friendsFeed.Add(status);

            return TaskDone.Done;
        }

        public Task<SocialStatus> GetStatus()
        {
            return Task.FromResult(_lastStatus);
        }

        public Task<List<SocialStatus>> GetMyFeed()
```

```
        {
            return Task.FromResult(_myFeed);
        }
    }
```

Alternatively we can model our grains to fan out and compile the Friends Feed at the query timer, in other words when the user asks for their friends feed. Another method we can use is materialising the Friends Feed on a timer, for example, every 5 minutes. Or, we can optimise the model and combine both event time and query time processing with a timer-based model depending on user habits, such as how often they login or post an update.

Another approach in addressing the problem is using Observers in Orleans, allowing grains to subscribe to each other and observe changes in another grain's state. Orleans provides built-in classes for safe subscription management. "Chirper" is a good example of the use of observable grains.

When modelling a grain in a social network, one should also consider "super users," users with millions of followers. Developers should model the state and behaviour of such users differently to meet the demand.

Similarly, if we want to model an activity that connects many user grains to a single activity grain (hub and spoke) that can be done as well. Group chat or game hosting scenarios are two examples.

Let's take the group chat example; a set of participants create a group chat grain that can distribute messages from one participant to the group as in the example below:

Smart Cache – GroupChat Example

```
    public interface IGroupChat : IGrain
    {
        Task PublishMessageAsync(long participantId, string message);
        Task<List<GroupChatMessage>> GetMessagesAsync();
        Task AddParticipantAsync(long participantId);
        Task RemoveParticipantAsync(long partitipantId);
    }

   public class GroupChatParticipant : GrainBase, IGroupParticipant
   {
      private long _groupChatId;
      private List<GroupChatMessage> _messages;

       public Task SendMessageAsync(string message)
       {
          if (_groupChatId != -1)
          {
             var groupChat = GroupChatFactory.GetGrain(_groupChatId);
             return groupChat.PublishMessageAsync(this.GetPrimaryKeyLong(), message);
          }

          return TaskDone.Done;
       }

      ...

   }

   public class GroupChat : GrainBase, IGroupChat
```

```
{
   private List<long> _participants;
   private List<GroupChatMessage> _messages;

     public Task PublishMessageAsync(long participantId, string message)
     {
         var chatMessage = new GroupChatMessage()
         {
             ParticipantId = participantId,
             Message = message,
             Timestamp = DateTime.Now
         };

         _messages.Add(chatMessage);

         var taskList = new List<Task>();

         foreach(var id in _participants)
         {
             if (id != participantId)
             {
                 var participant = GroupParticipantFactory.GetGrain(id);
                 taskList.Add(participant.ReceiveMessageAsync(chatMessage));
             }
         }
         return Task.WhenAll(taskList);
     }

   ...

 }
```

All it really does is leverage Orleans' ability to allow any grain to address any other grain in the cluster by id and communicate with it without needing to worry about *placement*, *addressing*, *caching*, *messaging, serialisation,* or *routing*.

Pattern: Resource Governance

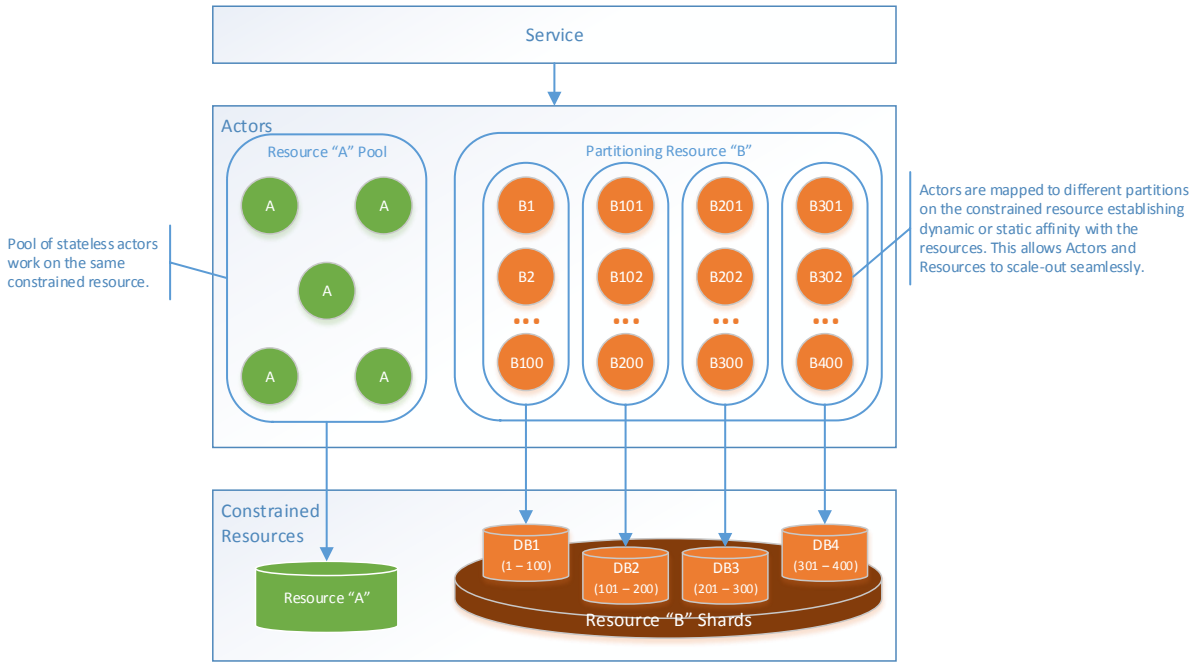
This pattern and related scenarios are easily recognisable by developers—enterprise or otherwise—who have constrained resources on-premises or in the cloud, which they cannot immediately scale or that wish to ship large scale applications and data to the cloud.

In the enterprise these constrained resources, such as databases, run on scale-up hardware. Anyone with a long enterprise history knows this is a common situation on-premises. Even at cloud scale, we have seen this situation occur when a cloud service has attempted to exceed the 64K TCP limit of connections between an address/port tuple, or when attempting to connect to a cloud-based database that limits the number of concurrent connections.

Typically in the past, this was solved by throttling through message-based middleware or by custom-built pooling and façade mechanisms. These are hard to get right, especially when we need to scale the middle tier but still maintain the correct connection counts. It's just fragile and complex.

In fact, like the Smart Cache pattern, this pattern spans across multiple scenarios and customers who already have working systems with constrained resources. They are building systems where they need to scale-out not just the services, but their state in-memory as well as the persisted state in stable storage.

The diagram below illustrates this scenario:



Essentially we model access to resources as a grain or multiple grains that act as proxies (say connection, for example) to a resource or a group of resources. You can then either directly manage the resource through individual grains or use a coordination grain that manages the resource grains.

To make this more concrete, we will address the common need of having to work against a partitioned (aka sharded) storage tier for performance and scalability reasons.

Our first option is pretty basic: we can use a static function to map and resolve our grains to downstream resources. Such a function can return, for example, a connection string with given input. It is entirely up to us how to implement that function. Of course, this approach comes with its own drawbacks such as static affinity that makes repartitioning resources or remapping a grain to resources very difficult.

Here is a very simple example—we do modulo arithmetic to determine the database name using `userId` and use `region` to identify the database server.

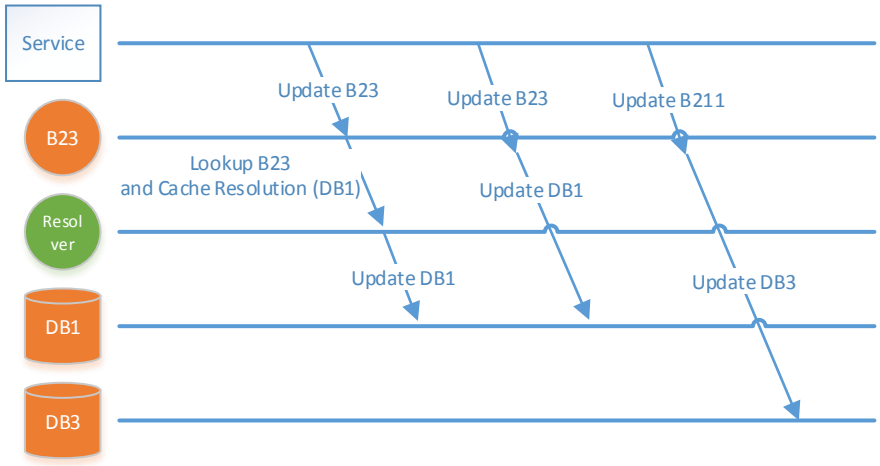
Resource Governance – Static Resolution

```
        private static string _connectionString = "none";

        private static string ResolveConnectionString(long userId, int region)
        {
            if (_connectionString == "none")
            {
                // an example of static mapping
                _connectionString = String.Format("Server=SERVER_{0};Database=DB_{0}", region,
userId % 4);
            }
            return _connectionString;
        }
```

Simple yet not very flexible. Now let's have a look at a more advanced and useful approach.

First, we model the affinity between physical resources and grains. This is done through a grain called Resolver that understands the mapping between users, logical partitions, and physical resources. Resolver maintains its data in a persisted store, however it is cached for easy lookup. As we saw in the Exchange Rate sample earlier in the Smart Cache pattern, Resolver can proactively fetch the latest information using a timer. Once the user grain resolves the resource it needs to use, it caches it in a local variable called `_resolution` and uses it during its lifetime.

We chose a look-up based resolution (illustrated below) over simple hashing or range hashing because of the flexibility it provides in operations such as scaling in/out or moving a user from one resource to another.



In the illustration above, we see that grain B23 is first resolving its resource (aka resolution) —DB1 and caches it. Subsequent operations can now use the cached resolution to access the constrained resource. Since the grains support single-threaded execution, developers no longer need to worry about concurrent access to the resource.

The `User` and `Resolver` grains look like this:

Resource Governance – Resolver Example

```
public interface IUser : IGrain
{
    Task UpdateProfile(string name, string country, int age);
}

public class User : GrainBase, IUser
{
    private long _userId;
    private string _name;
    private string _country;
    private int _age;

    private Resolution _resolution;

    public override async Task ActivateAsync()
    {
        _userId = this.GetPrimaryKeyLong();
```

```
            var resolver = ResolverFactory.GetGrain(0);
            _resolution = await resolver.ResolveAsync(_userId);
            await base.ActivateAsync();
        }
        public Task UpdateProfile(string name, string country, int age)
        {
            Console.WriteLine("Using {0}", _resolution.Resource.ConnectionString);
            // this is where we use the resource...
            return TaskDone.Done;
        }
    }
```

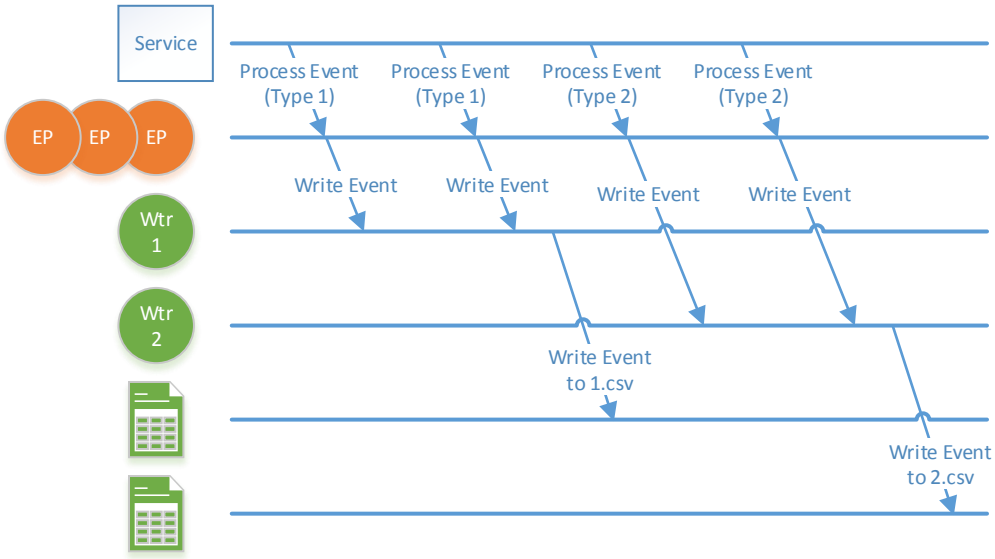
Resource Governance – Resolver Example

```
    public interface IResolver : IGrain
    {
        Task<Resolution> ResolveAsync(long entity);
    }

    public class Resolver : GrainBase, IResolver
    {
        ...

        public Task<Resolution> ResolveAsync(long entityKey)
        {
            if (_resolutionCache.ContainsKey(entityKey))
                return Task.FromResult(_resolutionCache[entityKey]); // return from cache

            var partitionKey = _entityPartitions[entityKey]; // resolve partition;
            var resourceKey = _partitionResources[partitionKey]; // resolve resource;
            var resolution =
                new Resolution()
                {
                    Entity = _entities[entityKey],
                    Partition = _partitions[partitionKey],
                    Resource = _resources[resourceKey]
                }; // create resolution

            _resolutionCache.Add(entityKey, resolution); // cache the resolution

            return Task.FromResult(resolution);
        }

        ...
    }
```

Now let's look at another example; *exclusive access* to precious resources such as DB, storage accounts, and file systems with finite throughput capability.

Our scenario is as follows: we would like to process events using a grain called `EventProcessor`, which is responsible for processing and persisting the event, in this case to a .CSV file for simplicity. While we can follow the partitioning approach discussed above to scale-out our resources, we will still have to deal with the concurrency issues. That is why we chose a file-based example to illustrate this particular point—writing to a single file from multiple grains will raise concurrency issues. To

address the problem we introduce another grain called `EventWriter` that has exclusive ownership of the constrained resources. The scenario is illustrated below:



We mark `EventProcessor` grains as "Stateless Workers," which allows the runtime to scale them across the cluster as necessary. Hence we didn't use any identifiers in the illustration above for these grains. In other words, stateless grains are a pool of workers maintained by the runtime.

In the sample code below, the `EventProcessor` grain does two things: it first decides which `EventWriter` (therefore resource) to use, and invokes the chosen grain to write the processed event. For simplicity, we are choosing Event Type as the identifier for the `EventWriter` grain. In other words, there will be one and only one `EventWriter` for this Event Type providing *single-threaded and exclusive* access to the resource.

Resource Governance – Event Processor

```
    [StatelessWorker]
    public interface IEventProcessor : IGrain
    {
        Task ProcessEventAsync(long eventId, long eventType, DateTime eventTime, string payload);
    }

    public class EventProcessor : GrainBase, IEventProcessor
    {
        public Task ProcessEventAsync(long eventId, long eventType, DateTime eventTime, string
payload)
        {
            // This where we write to constrained resource...
            var eventWriterKey = ResolveWriter(eventType, eventTime);
            var eventWriter = EventWriterFactory.GetGrain(eventWriterKey);

            return eventWriter.WriteEventAsync(eventId, eventType, eventTime, payload);
        }

        private long ResolveWriter(long eventType, DateTime eventTime)
        {
            // To simplify, we are returning event type as to identify the event writer grain.
            return eventType;
        }
    }
```

Now let's have a look at the `EventWriter` grain. It really doesn't do much apart from control exclusive access to the constrained resource, in this case the file and writing events to it.

Resource Governance – Event Writer

```
    public interface IEventWriter : IGrain
    {
        Task WriteEventAsync(long eventId, long eventType, DateTime eventTime, string payload);
        Task WriteEventBufferAsync(long eventId, long eventType, DateTime eventTime, string
payload);
    }

    public class EventWriter : GrainBase, IEventWriter
    {
        private StreamWriter _writer;
        private string _filename;

        public override Task ActivateAsync()
        {
            _filename = string.Format(@"C:\{0}.csv", this.GetPrimaryKeyLong());
            _writer = new StreamWriter(_filename);

            return base.ActivateAsync();
        }

        public override Task DeactivateAsync()
        {
            _writer.Close();
            return base.DeactivateAsync();
        }

        public async Task WriteEventAsync(long eventId, long eventType, DateTime eventTime, string
payload)
        {
            var text = string.Format("{0}, {1}, {2}, {3}", eventId, eventType, eventTime, payload);
            await _writer.WriteLineAsync(text);
            await _writer.FlushAsync();
        }
     }
```

Having a single grain responsible for the resource allows us to add capabilities such as buffering. We can buffer incoming events and write these events periodically using a timer or when our buffer is full. Here is a simple timer based example:

Resource Governance – Event Writer with Buffer

```
    public class EventWriter : GrainBase, IEventWriter
    {
        private StreamWriter _writer;
        private string _filename;
        private Queue<CustomEvent> _buffer;
        private IOrleansTimer _timer;

        public override Task ActivateAsync()
        {
```

```
            _filename = string.Format(@"C:\{0}.csv", this.GetPrimaryKeyLong());
            _writer = new StreamWriter(_filename);
            _buffer = new Queue<CustomEvent>();

            this.RegisterTimer(
                ProcessBatchAsync,
                null,
                TimeSpan.FromSeconds(5),
                TimeSpan.FromSeconds(5));

            return base.ActivateAsync();
        }

        private async Task ProcessBatchAsync(object obj)
        {
            if (_buffer.Count == 0)
               return;

            while (_buffer.Count > 0)
            {
                var customEvent = _buffer.Dequeue();
                await this.WriteEventAsync(customEvent.EventId, customEvent.EventType,
customEvent.EventTime, customEvent.Payload);
            }

        public Task WriteEventBufferAsync(long eventId, long eventType, DateTime eventTime, string
payload)
        {
            var customEvent = new CustomEvent()
            {
                EventId = eventId,
                EventType = eventType,
                EventTime = eventTime,
                Payload = payload
            };

            _buffer.Enqueue(customEvent);

            return TaskDone.Done;
        }
    }
```

While the code above will work fine, clients will not know whether their event made it to the underlying store. To allow buffering and let clients be aware what is happening to their request, we introduce the following approach to let clients wait until their event is written to the .CSV file:

Resource Governance – Async. Batching

```
    public class AsyncBatchExecutor
    {
        private readonly List<TaskCompletionSource<bool>> actionPromises;

        public AsyncBatchExecutor()
        {
            this.actionPromises = new List<TaskCompletionSource<bool>>();
        }

        public int Count
        {
            get
```

```
            {
                return actionPromises.Count;
            }
        }

        public Task SubmitNext()
        {
            var resolver = new TaskCompletionSource<bool>();
            actionPromises.Add(resolver);
            return resolver.Task;
        }

        public void Flush()
        {
            foreach (var tcs in actionPromises)
            {
                tcs.TrySetResult(true);

            }
            actionPromises.Clear();
        }
    }
```

We will use this class to create and maintain a list of incomplete tasks (to block clients) and complete them in one go once we write the buffered events to storage.

In the `EventWriter` class, we need to do three things: mark the grain class as Reentrant, return the result of `SubmitNext()`, and `Flush` our timer. The modified code is as follows:

Resource Governance – Buffering with Async. Batching

```
    [Reentrant]
    public class EventWriter : GrainBase, IEventWriter
    {
        public override Task ActivateAsync()
        {
            _filename = string.Format(@"C:\{0}.csv", this.GetPrimaryKeyLong());
            _writer = new StreamWriter(_filename);
            _buffer = new Queue<CustomEvent>();
            _batchExecuter = new AsyncBatchExecutor();

            this.RegisterTimer(
                ProcessBatchAsync,
                null,
                TimeSpan.FromSeconds(5),
                TimeSpan.FromSeconds(5));

            return base.ActivateAsync();
        }

        private async Task ProcessBatchAsync(object obj)
        {
            if (_batchExecuter.Count > 0)
            {
                // take snapshot of the batch tasks
                var batchSnapshot = _batchExecuter;
                _batchExecuter = new AsyncBatchExecutor();

                if (_buffer.Count == 0)
                    return;
```

```
                while (_buffer.Count > 0)
                {
                    var customEvent = _buffer.Dequeue();
                    await this.WriteEventAsync(customEvent.EventId, customEvent.EventType,
customEvent.EventTime, customEvent.Payload);
                }

                _batchExecuter.Flush();
            }
        }
        ...

        public Task WriteEventBufferAsync(long eventId, long eventType, DateTime eventTime, string
payload)
        {
            var customEvent = new CustomEvent()
            {
                EventId = eventId,
                EventType = eventType,
                EventTime = eventTime,
                Payload = payload
            };

            _buffer.Enqueue(customEvent);

            // we are adding an incomplete task to batch executer and returning this task.
            // this will block until task is completed when we call Flush();
            return _batchExecuter.SubmitNext();
        }
```

Seem easy? Well it is. But the ease belies the enterprise power. With this architecture we get:

- Location independent resource addressing.
- Tuneable pool size based simply on changing the number of grains that act on behalf a resource.
- Client side coordinated pool usage (as depicted) or server side (just imagine a single grain in front of each of those pools in the picture).
- Scalable pool addition (just add silos and grains representing the new resource).
- Grain (as we demonstrated earlier) can cache results from backend resource on demand or pre-cache using a timer reducing the need to hit the backend resource.
- Efficient asynchronous dispatch.
- A coding environment that will be familiar to any developer not just middleware specialists.

This pattern is very common in scenarios where developers either have constrained resources they need to develop against or building large scale-out systems.

Pattern: Stateful Service Composition

Developers spent the last decade and a half building N-Tier stateless services in the enterprise. They built services on top of databases, they built high order services on top of other services, and they built orchestration engines and message oriented middleware to coordinate these services. As the user workloads evolve, whether demanding more interactivity or scale, stateless service-oriented architecture began to show its weaknesses.

While SOA services scaled horizontally seamlessly due to their stateless nature, they created a bottleneck in the storage tier—*concurrency* and *throughput*. Accessing storage became more and more expensive. As a common practice most developers introduced caching to their solution to reduce the demand on storage but that solution was not without its drawbacks—*another tier to manage*, *concurrent access to cache, semantic limitations and changes, and finally consistency*. As detailed earlier in the Smart Cache pattern, the virtual actor model provides a perfect solution for this.

Some developers tried to solve the problem by replicating their storage tier. However, this approach didn't scale well and quickly hits CAP boundaries.

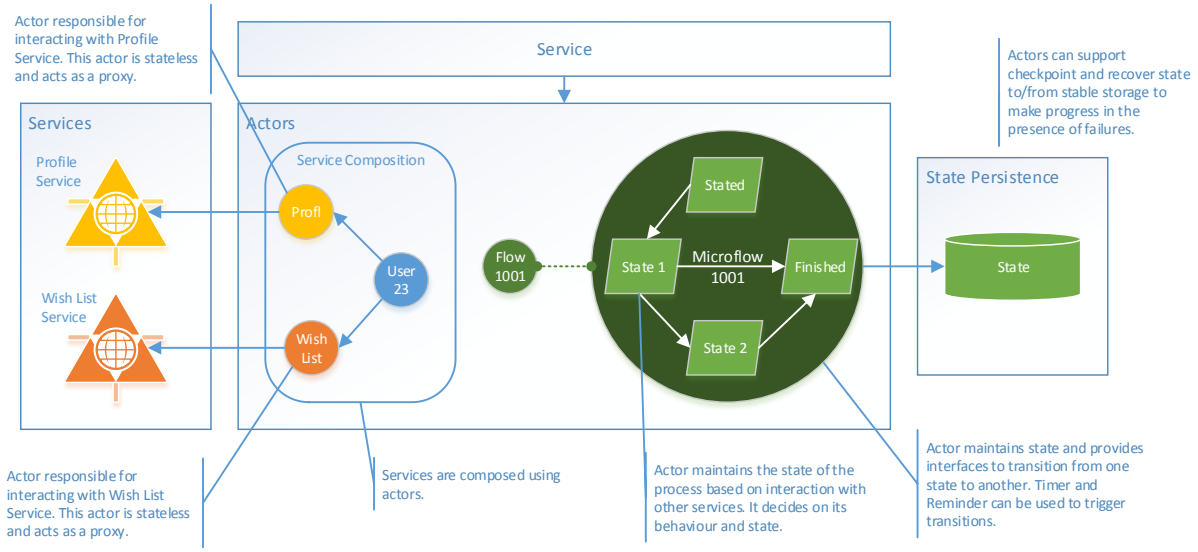
The second challenge has evolved around changing requirements; both end-users and businesses are demanding interactive services—responding to requests in milliseconds rather than seconds as the norm. To respond, developers started building façade services on top of other services, in some cases 10s of services to create user-centric services. However composing multiple downstream services quickly showed latency issues.

Once again developers turned to caches and *in-memory object stores*, in some cases different implementations to meet performance requirements. They started building backend worker processes to build the cache periodically to minimise expensive on-demand cache population. Finally, they started deconstructing their workloads to isolate *asynchronous operations* from synchronous ones to gain more room for interactive operations to *react to changes in state,* which is particularly hard in SOA.

They introduced further tiers such as queues and workers adding more complexity to their solutions.

Essentially, developers started looking for solutions to build "stateful services," in other words, collocate "state" and "service behaviour" to address user centric and interactive experiences. And this is where Orleans as a service composition tier comes in, not as a replacement for these services.

The diagram below illustrates the point:



In the case of composing services, grains can be either stateless or stateful.

- Stateless grains can used as proxies to the underlying services. These grains can dynamically scale across the Orleans cluster and can cache certain information related to the service, such as its endpoint once it is discovered.
- Stateful grain can maintain state between service calls as well as cache previous service results. State can be persisted or transient.

This pattern is also applicable across many scenarios; in most cases, grains need to make external calls to invoke an operation on a particular service.

Let's illustrate with an example using modern ecommerce applications. These applications are built on silos of services that provides various functionality such as User Profile Management, Recommendations, Basket Management, Wish List Management, Purchasing, and many more.

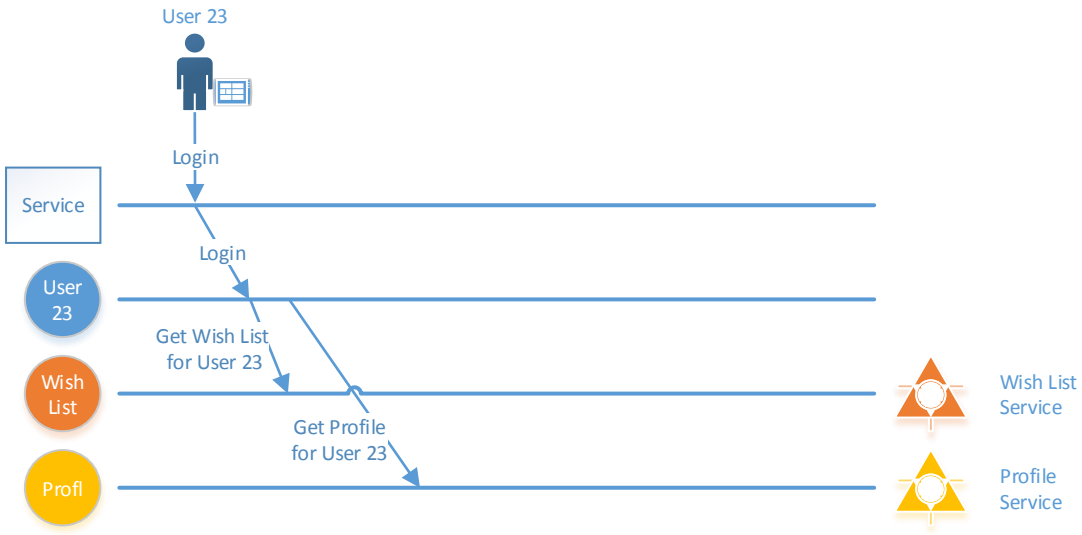
Most developers wish to take a *user-centric approach* to their architecture, very similar to those developing social experiences since ecommerce experiences primarily revolve around users and products. This is usually achieved by shipping a façade of services most likely supported by a cache for performance reasons.

Now let's talk about an actor based approach. A user grain can represent both the behaviour of the user (browsing the catalogue, liking a product, adding an item to basket, recommending a product to a friend) as well as the its composed state—their profile, items in the basket, list of items recommended by their friends, their purchase history, current geo-location, and so on.

First let's look at an example where the user grain needs to populate its state from multiple services. We are not going to provide a code sample for this one because everything we have discussed in the Smart Cache pattern is also applicable here.

We can activate the user grain at login time, populating it with sufficient data from back-end services. Of course, as we have seen on many occasions earlier in this paper, whole and partial state can be prepopulated on demand, on a timer, or a bit of both and cached in the grain.

For this example, Profile and Wish List is illustrated below:

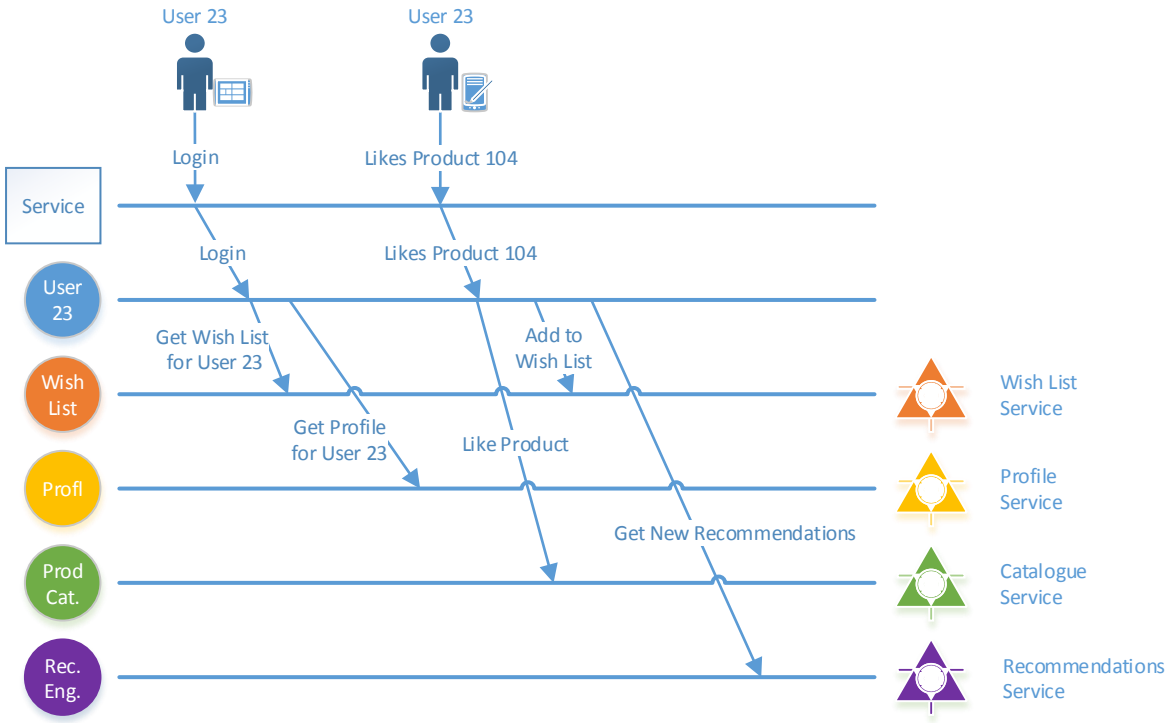


For instance we can prepopulate frequent users' state and make it ready when they login or populate it at login time for users who visit the service every month. We saw these patterns in the Smart Cache section.

When User 23 logs in, if not already activated, the user grain (23) is activated and fetches the relevant user profile information and wish list from back-end services. The user grain likely caches the information for subsequent calls. And if, for example, we need to add an item to the wish list we can write-behind or write-through as discussed earlier.

Secondly, let's have a look at an example where the user clicks on the "like" button and likes a product. This action may require multiple invocations to multiple services as illustrated below: Send a "like" to catalogue service, trigger the next set of recommendations, and perhaps post an update to a social network.

This is illustrated below:



In fact, Orleans shines when we want to compose request/response style operations together with asynchronous operations. For instance, while "Like Product" immediately puts the liked item into the user's wish list, posting to social networks and triggering the next set of recommendations can be asynchronous operations using buffers and timers.

One other key benefit of using a user grain with services is grains provide a natural place for cached state and most importantly react to changes in its state asynchronously. This is a particularly challenging scenario with stateless services.

For example, a user carries out a series of actions, perhaps part of a "user journey." These events can be captured in real time in the grain and we can assemble a stream, which we can query at event time or asynchronously on a timer to change the behaviour of the grain.

At this point SOA purists will no doubt have noticed that these are not services in the sense of grains as endpoints exposed over a language independent protocol. Orleans is neither an interoperation component nor a platform for service interoperation. Nevertheless, there really is nothing preventing us from thinking in terms of the granularity of SOA-style services when we model our grains or in modelling separation of concerns in the same way. Such services are known as "*microservices.*"

Likewise, there is absolutely nothing preventing us from putting a REST endpoint (like Halo does) or a SOAP endpoint as an interop layer in front of Orleans.

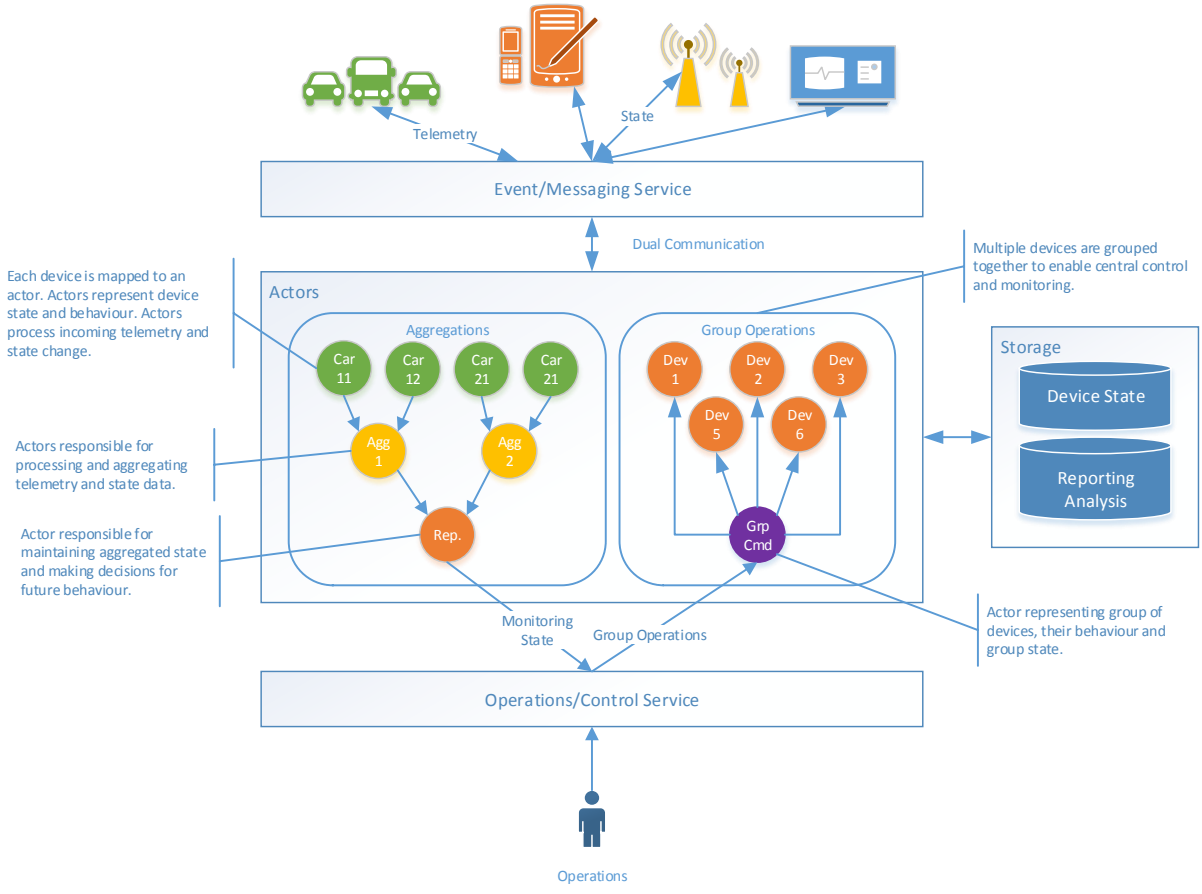
Stateful service composition also applies to workflows and not just transactional scenarios such as ecommerce. Orleans was not designed as a workflow/orchestration engine, but due to its stateful nature, persistence options, timer options and built-in location-transparent communication, it turns out we can model workflows involving service interactions and maintain the state of these interactions in Orleans quite well.

We see drawbacks of "stateless service" in building scalable services to provide dynamic experiences. Orleans, essentially by bringing state and behaviour together, helps developers build scalable and interactive experiences on top of their existing investments.

Pattern: Internet of Things (IOT)

Since IoT became the new trend alongside the technological advancement in both devices and cloud services, developers started looking at key building blocks to develop their systems on.

The following diagram illustrates the key scenarios achieved using Orleans:



Orleans is the key building block (as a middle-tier) in a system that combines a messaging system front end that supports multiple transports such as HTTPS, MQTT or AMQP then communicates with grains that represent individual devices. Because the grains can maintain state, modelling streams—especially stateful stream processing—and aggregation per device is straightforward. If the data must be persisted, then we can also easily flush on demand or on a timer while still easily maintaining the latest N bits of data in another variable for quick querying.

Note that in our samples, we deliberately omitted the details of the event/messaging tier, which will allow grains to communicate with devices, to keep the focus on the actor model.

There are essentially two scenarios usually composed together:

- *Collecting telemetry and state data from a single or a set of devices and maintaining their state*. Think about tens of thousands of mouse traps (yes, this is a real customer scenario) sending data, as basic as whether the device trapped a nasty pest inside or not. Data is aggregated by region, and with enough mice trapped in one region, an engineer is dispatched to clean up the devices. A mouse-trap as a grain? Absolutely. A group grain per region as the aggregator? You bet.
- *Pushing device behaviour and configuration to a single or set of devices*. Think about home solar power devices where the vendor pushes different configurations based on consumption patterns and seasonality.

First let's have a look at the case where devices, think tens of thousands, are grouped together and are all sending telemetry data to their associated group. In the following example, the customer has deployed devices to each region.

When the device is switched on, the first thing it does is send an ActivateMe message with relevant information such as location, version, and so on. In turn, the grain associated with the device (through the Device Id) sets up the initial state for the device, such as saving state locally (could have been persisted also) and registering a group grain. In this case, we assign a random group for our simulation.

As part of the initialization process, we can configure the device by retrieving configuration data from a group grain or some other agent. This way the devices can initially be pretty dumb and get their "smarts" upon initialization. Once this is done, the device and the grain are ready to send and process telemetry data.

All the devices periodically send their telemetry information to the corresponding grain. If the grain is already activated then the same grain will be used. Otherwise it will be activated. At this point it can recover state from a stable store if required. When the grain receives telemetry information it stores it to a local variable. We are doing this to simplify the sample. In a real implementation we would probably save it to an external store to allow operations to monitor and diagnose device health and performance. Finally, we push telemetry data to the group grain that the device grain logically belongs to.

IoT – Telemetry

```
public interface IThing : IGrain
{
```

```
        Task ActivateMe(string region, int version);
        Task SendTelemetryAsync(ThingTelemetry telemetry);
    }

    public class Thing : GrainBase, IThing
    {
        private List<ThingTelemetry> _telemetry;
        private ThingInfo _deviceInfo;
        private long _deviceGroupId;

        public override Task ActivateAsync()
        {
            _telemetry = new List<ThingTelemetry>();
            _deviceGroupId = -1; // not activated
            return base.ActivateAsync();
        }

        public Task SendTelemetryAsync(ThingTelemetry telemetry)
        {
            _telemetry.Add(telemetry); // saving data at the device level
            if (_deviceGroupId != -1)
            {
                var deviceGroup = ThingGroupFactory.GetGrain(_deviceGroupId);
                return deviceGroup.SendTelemetryAsync(telemetry); // sending telemetry data for
aggregation
            }
            return TaskDone.Done;
        }

        public Task ActivateMe(string region, int version)
        {
            _deviceInfo = new ThingInfo()
                {
                    DeviceId = this.GetPrimaryKeyLong(),
                    Region = region,
                    Version = version
                };

            // based on the info, assign a group... for demonstration we are assigning a random
group
            _deviceGroupId = new Random().Next(10, 12);

            var deviceGroup = ThingGroupFactory.GetGrain(_deviceGroupId);
            return deviceGroup.RegisterDevice(_deviceInfo);
        }
    }
```

At the group level, as per our sample, our goal is to monitor the devices in the group and aggregate telemetry data to produce alerts for engineers. In this case, our customer would like to send engineers to specific regions where there are a certain number of fault devices. Of course our customer would like to reduce costs by minimising engineering time spent on the road. For this reason, each group grain maintains an aggregated state of faulty devices per region. When this number hits a threshold, our customer dispatches an engineer to the region to replace/repair these devices.

Let's have a look how it is done:

IoT – Telemetry

```
    public interface IThingGroup : IGrain
    {
        Task RegisterDevice(ThingInfo deviceInfo);
        Task UnregisterDevice(ThingInfo deviceInfo);
        Task SendTelemetryAsync(ThingTelemetry telemetry);
    }

    public class ThingGroup : GrainBase, IThingGroup
    {
        private List<ThingInfo> _devices;
        private Dictionary<string, int> _faultsPerRegion;
        private List<ThingInfo> _faultyDevices;

        public override Task ActivateAsync()
        {
            _devices = new List<ThingInfo>();
            _faultsPerRegion = new Dictionary<string, int>();
            _faultyDevices = new List<ThingInfo>();

            return base.ActivateAsync();
        }

        public Task RegisterDevice(ThingInfo deviceInfo)
        {
            _devices.Add(deviceInfo);
            return TaskDone.Done;
        }

        public Task UnregisterDevice(ThingInfo deviceInfo)
        {
            _devices.Remove(deviceInfo);
            return TaskDone.Done;
        }

        public Task SendTelemetryAsync(ThingTelemetry telemetry)
        {
            if (telemetry.DevelopedFault)
            {
                if (false == _faultsPerRegion.ContainsKey(telemetry.Region))
                {
                    _faultsPerRegion[telemetry.Region] = 0;
                }
                _faultsPerRegion[telemetry.Region]++;
                _faultyDevices.Add(_devices.Where(d => d.DeviceId ==
telemetry.DeviceId).FirstOrDefault());

                if (_faultsPerRegion[telemetry.Region] > _devices.Count(d => d.Region ==
telemetry.Region) / 3)
                {
                    Console.WriteLine("Sending an engineer to repair/replace devices in {0}",
telemetry.Region);
                    foreach(var device in _faultyDevices.Where(d => d.Region ==
telemetry.Region).ToList())
                    {
                        Console.WriteLine("\t{0}", device);
                    }
                }
            }

            return TaskDone.Done;
        }

    }
```

As we can see it is pretty straight forward. After running simple tests, the output looks like this:

IoT – Telemetry

```
Sending an engineer to repair/replace devices in Richmond
        Device = 33 Region = Richmond Version = 4
        Device = 79 Region = Richmond Version = 5
        Device = 89 Region = Richmond Version = 3
        Device = 63 Region = Richmond Version = 2
        Device = 85 Region = Richmond Version = 4
```

By the way, you may think it would have been better if devices were grouped by region. Of course, it's entirely up to us in terms of how to group/partition devices—whether it is geo-location, device type, version, tenant, and so on.

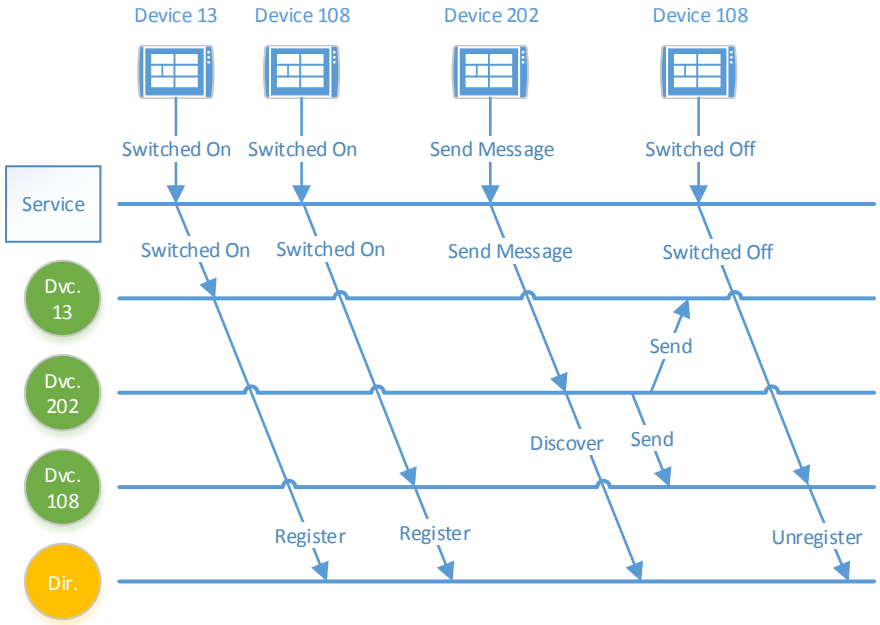
There is a point of caution here: Device State vs. Reporting/Analysis. This is why we made the pattern illustration explicit. We must avoid fan-out queries to grains to build reporting grains; unnecessary instantiations and performance are just to name two drawbacks. We recommend two approaches for reporting:

- Use group level grain, such as an aggregator, to maintain a view for the group. Let each grain push only relevant data proactively to this grain. Then this group level grain can be used to view the status of the devices in the group.
- Maintain an explicit store that is designed for reporting. An aggregator can buffer and periodically push data to a reporting store for further querying and analysis.

So far so good, but how about operations on these devices? Like for devices, grains can expose operational interfaces so an administrator can carry out operations on devices. For example, an administrator wants to push a new configuration to a group of home solar energy devices (yep, another real life scenario) based on seasonal/regional changes.

The key idea here is we have granular control over each device using “Thing” grains as well as group operations using “ThingGroup” grains—whether it is aggregating data coming in from devices such as telemetry or sending data such as configuration to large number of devices. The platform takes care of distribution of the device grains and messaging between them, which is totally transparent to the developer.

When it comes to machine to machine (M2M) communications, both the Hub and Spoke pattern we discussed in the distributed networks and graphs section or direct grain-to-grain interaction works well. For M2M scenarios, we could model a “Directory/Index” grain for a group of devices allowing them to discover and send messages to each other as illustrated below:



Orleans also takes care of the lifetime of the grains. Think of it this way, we will have always-on devices and we will have occasionally-connected devices. Why would we keep the grain that looks after the device that connects every 14 hours in memory? Orleans allow save and restore of device state when we want it and where we want it.

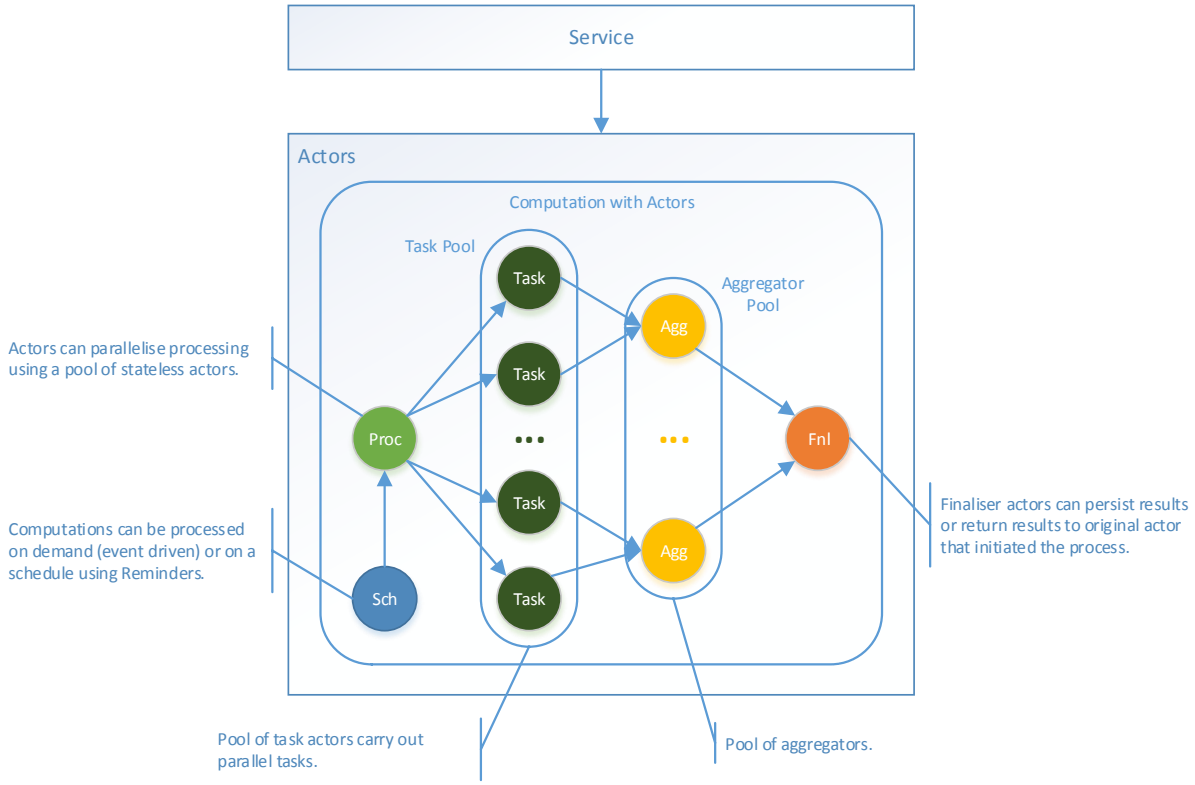
We conclude that more and more customers will look at Orleans as a key building block for their IoT implementations.

Pattern: Distributed Computation

We owe this one in part to watching a real life customer whip out a financial calculation in Orleans in an absurdly small amount of time—a Monte Carlo simulation for risk calculation to be exact.

At first, especially to those who do not have domain specific knowledge, Orleans' handling of this kind of workload, as opposed to say more traditional approaches such as Map/Reduce or MPI, may not be obvious.

But it turns out that Orleans is a good fit with parallel asynchronous messaging, easily managed distributed state, and parallel computation as the following diagram depicts:



In the following example, we simply calculate Pi using a Monte Carlo Simulation. We have the following grains:

- Processor responsible for calculating Pi using PoolTask grains.
- PoolTask responsible for Monte Carlo simulation and sending results to Aggregator.
- Aggregator responsible for, well, aggregating results and sending them to Finaliser.
- Finaliser responsible calculating the final result and printing on screen.

Distributed Computation – Monte Carlo Simulation

```
public interface IProcessor : IGrain
{
    Task ProcessAsync(int tries, int seed, int taskCount);
}

public class Processor : GrainBase, IProcessor
{
    public Task ProcessAsync(int tries, int seed, int taskCount)
    {
        var tasks = new List<Task>();
        for (int i = 0; i < taskCount; i++)
        {
            var task = PooledTaskFactory.GetGrain(0); // stateless
            tasks.Add(task.CalculateAsync(tries, seed));
        }
        return Task.WhenAll(tasks);
    }
}

[StatelessWorker]
public interface IPooledTask : IGrain
{
    Task CalculateAsync(int tries, int seed);
```

```
    }

    public class PooledTask : GrainBase, IPooledTask
    {
        public Task CalculateAsync(int tries, int seed)
        {
            var pi = new Pi()
            {
                InCircle = 0,
                Tries = tries
            };

            var random = new Random(seed);
            double x, y;
            for (int i = 0; i < tries; i++)
            {
                x = random.NextDouble();
                y = random.NextDouble();
                if (Math.Sqrt(x * x + y * y) <= 1)
                    pi.InCircle++;
            }

            var agg = AggregatorFactory.GetGrain(0);
            return agg.AggregateAsync(pi);
        }
    }
```

A common way of aggregating results in Orleans is to use timers. We are using stateless grains for two main reasons: the runtime will decide how many aggregators are needed dynamically, therefore giving us scale on demand; and it will instantiate these grains "locally" – in other words in the same silo of the calling grain, reducing network hops.

Here is how the Aggregator and Finaliser look:

Distributed Computation – Monte Carlo Simulation

```
    [StatelessWorker]
    public interface IAggregator : IGrain
    {
        Task AggregateAsync(Pi pi);
    }

    public class Aggregator : GrainBase, IAggregator
    {
        private Pi _pi;
        private bool _pending;

        public override Task ActivateAsync()
        {
            _pi = new Pi() { InCircle = 0, Tries = 0 };
            _pending = false;

            this.RegisterTimer(
                ProcessAsync,
                null,
                TimeSpan.FromSeconds(5),
                TimeSpan.FromSeconds(5));

            return base.ActivateAsync();
        }
```

```
        private async Task ProcessAsync(object obj)
        {
            if (false == _pending)
                return;

            var finaliser = FinaliserFactory.GetGrain(0);
            await finaliser.FinaliseAsync(_pi);
            _pending = false;
            _pi.InCircle = 0;
            _pi.Tries = 0;
        }

        public Task AggregateAsync(Pi pi)
        {
            _pi.InCircle += pi.InCircle;
            _pi.Tries += pi.Tries;
            _pending = true;
            return TaskDone.Done;
        }
    }

    public interface IFinaliser : IGrain
    {
        Task FinaliseAsync(Pi pi);
    }

    public class Finaliser : GrainBase, IFinaliser
    {
        private Pi _pi;

        public override Task ActivateAsync()
        {
            _pi = new Pi()
            {
                InCircle = 0,
                Tries = 0
            };

            return base.ActivateAsync();
        }
        public Task FinaliseAsync(Pi pi)
        {
            _pi.InCircle += pi.InCircle;
            _pi.Tries += pi.Tries;
            Console.WriteLine(" Pi = {0:N9}  T = {1:N0}, {2}",(double)_pi.InCircle /
(double)_pi.Tries * 4.0, _pi.Tries, _pi.InCircle);

            return TaskDone.Done;
        }
    }
```

At this point, it should be clear how we could potentially enhance the Leaderboard example with an aggregator for scale and performance.

We are by no means asserting that Orleans is a drop-in replacement for other distributed computation of big data frameworks or high performance computing. There are some things it is just built to handle better than others. However one can model workflows and distributed parallel computation in Orleans while still getting the simplicity benefits it provides.

Orleans Anti-Patterns

We identified the following potential pitfalls for customers who are learning Orleans:

- *Treat Orleans as a transactional system.* Orleans is not a two phase commit-based system offering ACID. If we do not implement the optional persistence, and the machine the grain is running on dies, its current state will go with it. The grain will be coming up on another box very fast, but unless we have implemented the backing persistence, the state will be gone. However, between leveraging retries, duplicate filtering, and/or idempotent design, you can achieve a high level of reliability and consistency.
- *Block.* Everything we do in Orleans should be asynchronous. This is usually easy because async APIs are prolific now in the Microsoft platform. But if, for some reason, we must interact with a system that only provides a blocking API, we are going to need to put that in a wrapper that explicitly uses the .NET Thread Pool.
- *Over architect.* Let the environment work. It can be hard for developers who are accustomed to worrying about concurrent collections and locks, or using tools to compile objects from XML, to simply just code a class that does simple things like assign a value to a variable or schedule work. Scheduled tasks are built in. Locks are not needed. State is not a mortal enemy. This takes some getting used to for many folks who've done a lot of server side work in large scale environments.
- *Make a single actor the bottleneck*. It is often too easy to be trapped with this one, having millions of grains funnelling into a single instance of another actor. Use the aggregation approach that we demonstrated earlier.
- *Map entity models blindly.* This is for developers who are coming from a relational universe where problems are modelled using entities and their relationships. While this approach is still useful for understanding the subject domain, it should be coupled with service-oriented thinking and blended with the behaviour.

Summary

In this paper we detailed key solution patterns applicable to wide range of scenarios. Our experience with our customers has demonstrated that building these scenarios is greatly simplified by Orleans thanks to its productive programming model, single-threaded concurrency model and is scalability-by-design properties.