"Afterthought"

Adding Streaming Abstractions to Virtual Actors

Sergey Bykov, *Microsoft*

Orleans – Virtual Actor Framework

Distributed stateful runtime

Grains (Actors)

- .NET objects
- Always exist, virtually
- Messaging through .NET interfaces
- Asynchronous via async/await in C#
- Automatic error propagation

Silo: runtime execution container

- Implicit activation & lifecycle management
- Coordinated placement
- Multiplexed communication
- Failure recovery

Async RPC Calls to Grains

```
public interface IUser : IGrainWithGuidKey
{
        Task<bool> AddFriend(Guid friendId);
}


IUser me = GrainFactory.GetGrain<IUser>(myId);

try
{
        bool alreadyFriends = await me.AddFriend(friendId);
}
catch(Exception exc)
{
        Console.WriteLine("Failed to add {0} as friend: {1}", friendId, exc);
}
```

- GetGrain() to obtain a reference to a grain for a given key
- Invoke interface methods on the reference (proxy)
- Asynchronously *await* result
- Just like in a desktop app

Want More Than RPC

How to return a sequence of values?

- Task<IEnumerable<string>> GetStatusUpdates()?

Durability of events

- Send over persistent queues instead of TCP

How to manage publishers/subscribers?

- Task SubscribeForUpdates(handler)?

Scalability & granularity of streams

- Small number of ‘fat’ streams vs. millions of ‘anorexic’ ones

Virtual Streams for Virtual Actors

- Virtual stream
  - Stream is a logical abstraction
  - Stream always exits, cannot be created, cannot fail
  - Identified by a GUID and an optional string namespace
  - Stream subscriptions are durable
- Choice of behaviors and semantics via configuration
  - Different Delivery guarantees
    - Durable queues, pub/sub, best-effort one-way channels, etc.
  - Different Ordering guarantees
- Provider model for adapters and behaviors

Reactive Style Stream API

```
public interface IAsyncObserver<T>
{
        Task OnNext (T item);
        Task OnComplete();
        Task OnError(Exception ex);
}
```

```
public interface IAsyncObservable<T>
{
        Task Subscribe (IAsyncObserver<T> observer);
}
```

```
var provider = GetStreamProvider(“KafkaProvider");
var stream = provider.GetStream<T>(id, “updates");
```

```
await stream.OnNext(update1);
await stream.OnNext(update2);
```

```
IAsyncObserver<T> handler = new MyHandler();
await stream.Subscribe(handler);
        or
await stream.Subscribe(this);
```

Implementation

Host process (*Orleans Silo*) instantiates a set of *Pulling Agents* for getting messages from the queues

Each *Pulling Agent* loads a *Queuing Adaptor* for specified queuing service

Partitioning Manager coordinates division of work between pulling agents across silos

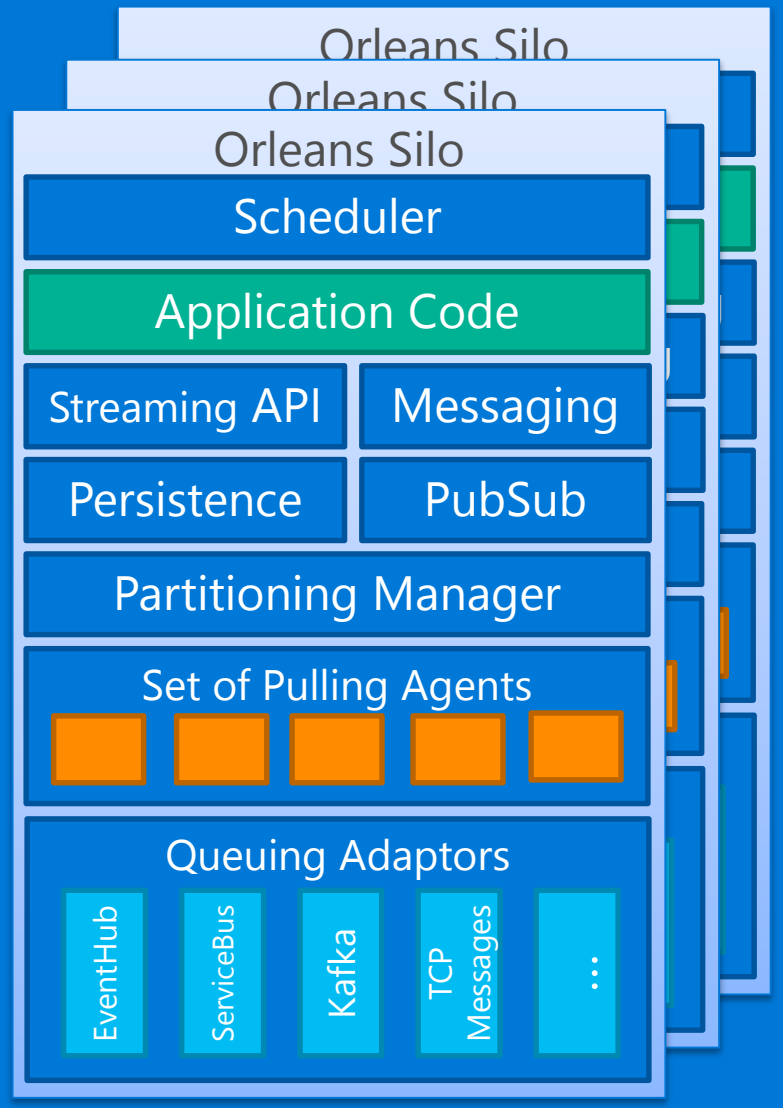
PubSub service stores mapping of producers and consumers to streams

Application Code invokes *Streaming API* to produce or consume events to/from streams

Pulling Agents and *Queuing Adaptors* use system services of silo for persistence, messaging, etc.

Execution is single-threaded, governed by Orleans *Scheduler*

Orleans Silos are units of scale and fault tolerance



Conclusion

Programming model for interactive workloads

Large number of independent contexts: user, device, session

Maps well to real-time communication, gaming, IoT, monitoring…

Large number of logical streams over much fewer physical queues

Good scalability

Runs in production for Halo 5

Thank you!

http://github.com/dotnet/orleans