# Why Orleans

The purpose of this document is to explain in a concise form the benefits of the actor based programming model as implemented by Orleans for building modern cloud applications, to highlight the problems it addresses, and briefly explain how it addresses them.

## Motivation

Cloud applications and services are inherently parallel and distributed. They are also interactive and dynamic, often requiring near real time direct interactions between cloud entities. Such applications are very difficult to build today. The development process demands expert level programmers and typically requires expensive iterations of the design and the architecture, as the workload grows.

Most of today’s high scale properties are built with the SOA paradigm. Rendering of a single web page by Amazon or Google or Facebook involves complex interactions of hundreds of SOA services that are independently built, deployed and managed. The fact that each individual service scales well by itself does not guarantee scalability of a composition of such services.

The data scale-out mechanism of SOA is partitioning. As data size and load grow and “hot spots” come and go, a service has to dynamically repartition its state and do so without interrupting its operation. SOA challenges the programmer with a high degree of concurrency of requests within process boundaries. However, existing tools do not provide good support for safe and efficient concurrency and distributed parallelism.

The stateless N-tier model delegates the partitioning problem to the storage layer. It generally requires caching in the stateless layer to get acceptable performance, which adds complexity and introduces consistency issues.

## Actor Model

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. This makes balancing of load easier compared to hash-based partitioning of SOA.

Erlang is the most popular implementation of the actor model. Facing the above-mentioned challenges of SOA, the industry started rediscovering the actor model, which stimulated renewed interest in Erlang and creation of new Erlang-like solutions: Scala actors, Akka, DCell.

## 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 .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.

### Orleans: Developer Productivity

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 remote method invocations: sending messages, routing them to the right endpoints, invoking the target actor’s methods and dealing with failures and corner cases in a fully transparent way.
* **Single-threaded execution of actors**. The Orleans runtime guarantees that actor’s code 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 an uninterrupted access to the full “memory space” of logically created actors, whether or not they are in the 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.

### Orleans: Transparent Scalability by Default

The Orleans programming model is designed to guide the programmer down a path of likely success in scaling their application or service through several orders of magnitude. This is done by incorporating the proven best practices and patterns, and providing an efficient implementation of the 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, the programmer implicitly breaks down the overall state of their application. 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 effort 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 of 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. Combined with asynchronous messaging and efficient scheduling, this enables a large degree of distributed parallelism and overall throughput without the explicit use of multi-threading.

## Applications

### Applications Most Suited for Orleans

Applications that benefit the most from Orleans are those that deal with a large number of relatively independent entities that directly interact with each other and do not require massive group operations. A graph of entities that establish ad-hoc connections to each other at run time and interact with a relatively limited number of other entities at any point in time maps perfectly to the actor model implemented by Orleans, especially if those entities are active for a period of time and may be garbage collected or dehydrated after that.

One example of such applications are social network graphs with massive numbers of entities, such as users, pictures, posts, and messages that have relations to a set of other entities in the system. At any point in time only a small subset of the entities need to be activated in memory in response to requests from clients. As a client adds/deletes entities and adds/removes relations between them, the entities need to be dynamically activated or rehydrated. After requests stop coming to the entities, the entities may be garbage collected until they are needed again. A large number of interconnected actors implement such an application in a relatively straightforward manner.

Another example that perfectly fits the Orleans model is multiplayer games. In those applications a fraction of a very large number of registered users go online and form short-lived groups for playing a session of the game. In that process they create or activate a number of entities that establish connections to each other, interact intensely for a while, and dissolve the group at the end of the session only to form new groups with other sets of entities. The fluid nature of actors and their interactions can encode such logic in a natural way.

### Applications Not Suitable for Orleans

Applications that require bulk operations on large sets of entities (SIMD) would likely be more efficient if they operate on relative small number of large shards combining many entities. A local operation across multiple entities with direct access to their memory will be more efficient than exchanging messages with per entity actors. For throughput, a shard will need to be able to execute multiple concurrent requests against its state as the single-threaded model of grains will likely be insufficient. An example would be a database application where large queries or bulk updates are frequent.

### Hybrid Applications

Many real life cloud applications likely need both SOA/SIMD and actor models to coexist and cooperate. While the interactive user-facing part may be best implemented with an actor model like Orleans, there are also backend aggregate tasks that may be more suitable for a SOA-based service or map-reduce style batch processing. The two or three parts of the application can easily be connected at the persistent storage layer but that may not be enough from the consistency perspective. This is an interesting problem to explore.

## Conclusion

Orleans is a programming model and distributed runtime that is focused on providing a high level of developer productivity and built-in scalability. It is based on the asynchronous actor model. Orleans makes cloud programming accessible to non-expert programmers by providing the familiar and convenient OOP paradigm, single-threaded execution guarantees, transparency of location and activation, and automatic persistence. The resulting code is simple and concise yet robust with regards to handling failures of the underlying hardware and network, and changes in the load characteristics. At the same time, applications built with Orleans are scalable by default because of the natural partitioning of the state, multiplexed communication, explicit asynchrony, cooperative multitasking and efficient scheduling on a small number of threads, and adaptive resource management. Orleans applications remain stable and performant at very high load and utilization.