Orleans
A Programming Model for the Cloud

Roger S. Barga
Microsoft Corporation

Jim Larus, Sergey Bykov, Gabriel Kliot, Alan Geller, Ravi Pandya, Jorgen Thelin, & Phil Bernstein
ORLEANS = PROGRAMMING MODEL + DISTRIBUTED RUNTIME

Radically simplified concurrent programming model
- Distributed, replicated actors
- No shared memory
- Location transparency
- Local sequential code
- Global asynchronous concurrency

Distributed runtime
- Messaging & serialization
- Scheduler
- Directory (DHT)
- Persistent state management

Transparent scalability
Adaptive resource management

Application
Orleans Runtime
.NET + Azure
Orleans

A Framework for Cloud Computing

Orleans is a project in the eXtreme Computing Group to simplify development of scalable cloud services.

Programmability is a major challenge in large-scale cloud systems. As cloud applications and services are inherently parallel and distributed, the development process usually demands expert-level programmers and typically requires expensive iterations of the design and the architecture as the workload grows.

Orleans is a software framework for building reliable, efficient and scalable cloud applications. Its programming model is based on distributed virtual actors, which are isolated units of state and computation that communicate through asynchronous messages. Orleans has two design goals: to greatly simplify the development of cloud applications, and to ensure that those applications are effortlessly scalable and reliable. Simplicity, however, sometimes comes at the expense of efficiency, so realizing both goals simultaneously is a challenging task. In this paper we outline the design principles behind Orleans and demonstrate how Orleans balances the above two goals.

The design choices in Orleans were affected by its successful production usage. We describe some of these usage scenarios by services running on Windows Azure and demonstrate how Orleans helped simplify their development, while producing robust and scalable applications. We outline the lessons learned from those production systems and how those lessons influenced the design of Orleans.
Orleans: A Framework for Cloud Computing

Sergey Bykov, Alan Geller, Gabriel Kliot, James Larus, Ravi Pandya, and Jorgen Thelin
30 November 2010

Client + cloud computing is a disruptive, new computing platform, combining diverse client devices – PCs, smartphones, sensors, and single-function and embedded devices – with the unlimited, on-demand computation and data storage offered by cloud computing services such as Amazon’s AWS or Microsoft’s Windows Azure. As with every advance in computing, programming is a fundamental challenge as client + cloud computing combines many difficult aspects of software development.

Orleans is a software framework for building client + cloud applications. Orleans encourages use of simple concurrency patterns that are easy to understand and implement correctly, building on an actor-like model with declarative specification of persistence, replication, and consistency and using lightweight transactions to support the development of reliable and scalable client + cloud software.

Related Projects
- Orleans

Related People
- Alan Geller
- Gabriel Kliot
- Jorgen Thelin
- Jim Larus
- Ravi Pandya
- Sergey Bykov

Related Groups
- Advanced Cloud Technologies
- Cloud Computing Research Group
- XCG Cloud Systems

Related Labs
- eXtreme Computing Group
Orleans: Cloud Computing for Everyone

Sergey Bykov, Alan Geller, Gabriel Kliot, James Larus, Ravi Pandya, and Jorgen Theiln
October 2011

Cloud computing is a new computing paradigm, combining diverse client devices – PCs, smartphones, sensors, single-function, and embedded – with computation and data storage in the cloud. As with every advance in computing, programming is a fundamental challenge, as the cloud is a concurrent, distributed system running on unreliable hardware and networks.

Orleans is a software framework for building reliable, scalable, and elastic cloud applications. Its programming model encourages the use of simple concurrency patterns that are easy to understand and employ correctly. It is based on distributed actor-like components called grains, which are isolated units of state and computation that communicate through asynchronous messages. Within a grain, promises are the mechanism for managing both asynchronous messages and local task-based concurrency. Isolated state and a constrained execution model allow Orleans to persist, migrate, replicate, and reconcile grain state. In addition, Orleans provides lightweight transactions that support a consistent view of state and provide a foundation for automatic error handling and failure recovery.

We implemented several applications in Orleans, varying from a messaging-intensive social networking application to a data- and compute-intensive linear algebra computation. The programming model is a general one, as Orleans allows the communications to evolve dynamically at runtime. Orleans enables a developer to concentrate on application logic, while the Orleans runtime provides scalability, availability, and reliability.

Related Projects
- Orleans

Related People
- Alan Geller
- Gabriel Kliot
- Jorgen Theiln
- Jim Larus
- Ravi Pandya
- Sergey Bykov

Related Groups
- XCG Cloud Systems

Related Labs
- eXtreme Computing Group
Motivation: Services Must Be Reliable and Available
What Can Programming Languages Do?

First-class language support
- Shared memory and message passing
- Replicated data
- Failure handling
- Domain-specific languages

Massively parallel development tools
- Failure detection and notification at scale
- Correctness and performance debugging for services

Services life cycle support
- Strong versioning
- Live update
- Introspective monitoring and control
Existing Languages

Many languages in use, no focus on Cloud
Does Programming Need to Change?
Cloud Programming Models

- Programming model ≠ programming language
  - Paradigm for thinking about and structuring solutions

- Cloud programming models do not exist
  - Web 1.0 technology (PHP, Ruby, ASP.NET, ...) doesn’t scale far enough
  - Building scalable, reliable cloud services is very hard, even for experts
  - There’s no simple model for integration with rapidly diversifying clients

- Challenge: enable mainstream developers to build client + cloud apps
  - And let expert application developers focus on their application logic, not on Azure “plumbing” (web roles, worker roles, queues)
New Programming Model – New Problems (and some old, unsolved ones)

Concurrency
Parallelism
Availability
Performance
Distribution
Reliability
Security & Privacy
Power
Concurrency

• Services are inherently concurrent
  • Simultaneously process multiple requests
  • Built from parallel processes and machines

• Threads or events?
  • Threads: familiar sequential model
    • But, not sequential since state can change while thread is preempted
    • Thread and context switching overhead limit concurrency
  • Events: handlers fracture program control flow
    • Program logic split across handlers
    • Explicit manipulation of local state

• High-level (state machine, Actor, ...) models?

• Parallel programming, redux
  • Absence of consensus retards research, development, reuse, interoperability, ...
Parallelism

- Modern processors are parallel
  - Increased performance + power efficiency

- Data centers are parallel, message-passing clusters
  - Driven by cost, throughput, availability concerns

- Shared-memory parallel programming is long-standing mistake
  - State of the art: threads and synchronization (assembly language)
  - Can’t even agree on shared memory semantics

- Abolish shared memory!
Data movement and placement is a fundamental concern.

Network bandwidth, latency, and reliability constraints:
- Why are 40 computers a fundamental building block?

Data is usually too large to move:
- Reverse of CPU-centric worldview: bring computation to data.

Mobile clients connected by narrow straws:
- Cellular service limited by shared bandwidth and spectrum allocation.
- "WiFi" limited by power and access point availability.
Distribution

- Distributed systems are rich source of difficult problems
  - Replication
  - Consistency
  - Quorum
- Well-studied field with good solutions
  - Replication
  - Persistence
  - Relaxed consistency
- Techniques are complex
  - Integrate into programming model
Availability

• Services must be **highly available**
  • Building national infrastructure
  • Blackberry/Google/Twitter... outage affect millions of people and gets national attention

• High availability is difficult
  • Hard to eliminate “single points of failure”
  • Murphy’s Law rules
  • Antithetical to rapid software evolution

• Programming languages offer little support for systematic error handling
  • Disproportionate number of bugs in error-handling code
Reliability

• Considerable progress in past decade on defect detection tools
  • Tools focused on local properties (e.g., buffer overruns, test coverage, races, etc.)
  • Little effort on system-wide properties

• Modular checking
  • Whole program analysis expensive and difficult and not practical for services
  • Assertions and annotations at module boundaries

• New domain of defects
  • Message passing
  • Code robustness
  • Potential performance bottlenecks
Orleans Approach

- Constrained programming model that leads developers down the garden path
  - "Scalable by default"

- Asynchronous, **isolated, distributed** actors as the core abstraction
  - No shared memory, **only message passing**, to prevent locality assumptions
  - Encourages **early partitioning of state**
  - **Actors are similar to objects**, so relatively familiar to programmers

- Single-threaded actors
  - No data races, no locks

- Replication of actors for throughput and to handle hotspots

- Declarative persistence
  - To shield developers from partitioning/sharding

- Lightweight (non-serializable) transactions
  - Provide a consistent view of state without sacrificing performance
Orleans Features

- Grains
- Grain activations
- Messages
- Promises
- Adaptive performance
- Persistence
Roger’s Summary of Orleans...

- **Object-oriented programming** paradigm is familiar, turning method calls into messages, runtime routes to the right endpoints;
- **Single threaded execution** of actors, when combined with isolation from other actors, the programmer never faces concurrency issues;
- **Transparent activation**, the runtime activates an actor as-needed, only when there is a message for it to process;
- **Location transparency**, an actor reference is logical and the translation to its physical location is handled by the Orleans runtime;
- **Automatic propagation of errors**, automatically propagates unhandled errors up the call chain.
Grain – Actor (see papers for comparison with Erlang, Scala,...)

- Encapsulates behavior and state, **they are isolated**
- Uniquely addressable via grain references (**message passing**)
- Encourage scalable architecture and enable automated management
- Internally **single-threaded**
  - Simple programming model
  - Strongly typed and versioned interfaces

![Diagram of message passing and state management](image-url)
Why Actors?

Fine-grain distributed objects
- Widely used, natural abstraction: computation as reusable service
- Isolation and message passing mirror physical hardware

Secure and isolated computation with clear communications
- Singularity OS
- Computation replication

Encapsulated and partitioned data
- Scalability and replication

Natural integration with persistent storage
- Grain resides on disk until activated
Activation – Grain Replicas

- Multiple activations of a grain execute concurrently (single or stateless);
  - Single request processed completely by one activation
  - Each activation isolated from all other activations
- Programmers don’t see activations, only grains (activated on demand...)
- Always exist logically, pick a server, instantiate on that server the actor .Net class;
- Reduce latency through parallelism and optimized placement of activations
- Deactivate, persist state to disk;
Asynch Message Passing – Communicating with Grains

- Fundamental in distributed systems, these message exchanges are **method calls**
- Isolation mechanism, also applied for local calls
- Messages are sent to a grain through a proxy object
  - The runtime directs the message to a specific activation
- Non-blocking execution, returns immediately
- A **promise** for a future result
Promises - Resolve Impedance Mismatch

Bind future computation to future result

- Grain operations return promises
  1. Initially unresolved, result at some point in the future;
  2. Fulfilled, result becomes the value of the promise;
  3. Resolved, either broken or fulfilled;

- Caller binds closure that is evaluated when result available
  - Produces promise for result of closure: result or completion;

- Errors propagate through dataflow
  - Error handling can be added where needed – ‘asynchronous try/catch’
Replication and Consistency

- Grain replication introduces the possibility of inconsistency
  - In the worst case, not even read-your-own-writes consistency
- Strong, single-image consistency is very costly and doesn’t scale
  - Required for a small subset of applications
- Weak, eventual consistency is a challenge to program against
Orleans Runtime

- Factor out common, important functionality to cloud apps
- Complex to implement
- Hard to get correct
- Typically afterthought
- Deployment, management, maintenance challenging for services
Similar Frameworks

• Enterprise Java Beans
  • Java framework for building enterprise-scale apps
  • Orleans: larger scale, more diverse clients, and simpler model

• Microsoft App Fabric
  • CLR-based, server component model combine components in existing technologies (.NET, WCF, Workflow, ...)
  • Incremental solution that makes it easier to build and deploy solutions based on existing technologies and provides some support for solving distributed system problems
  • Enterprise-scale systems
  • Orleans: focused replacement for existing Windows and .NET programming models for building cloud apps

• Salesforce.com Service Cloud
  • Integrated environment for building 3-tiered business apps
  • New, Java- / C#-like language for middle tier
  • Multi-tenancy support
  • Hosted by Salesforce.com

• Google App Engine
  • Python or Java frameworks
  • BigTable for data storage – schema-less, strongly consistent, optimistic concurrency, transactional
  • Designed for serving Web applications – passively driven by web requests, queues, or scheduled tasks
  • Stateless app logic tier, must complete within 30 seconds
Conclusion

• Client + cloud marks fundamental change in computing
  • Inflection point, paradigm shift, <insert buzzword here> ...

• New challenges require new solutions
  • Parallelism / concurrency / distribution
  • Reliability / availability
  • Communications
  • Power
  • Geo-distribution

• Opportunity to rethink computing
  • Reduce aggravation and management burden
  • Move to seamless, ubiquitous computing
  • Develop new, natural user interfaces