Programming the Cloud

James Larus
eXtreme Computing Group, Microsoft Research
Cornell University, November 11, 2010
Client + Cloud Computing

New computing platform

- Ubiquitous access
- Inherently distributed
- Many, diverse clients (single purpose → rich)
- “Unlimited” computation and data on demand
Evolution
New Hardware Enables New Software
Moore’s Law (Dennard scaling) is running out
• Scale out, not scale up
Inescapably concurrent
• Finally broke von Neumann bottleneck!
• Parallel computing at all levels
• Distributed too
Inherently heterogeneous
• Processors
• Cloud + client device
• Programming languages
Communication as important as computation
• Consumers and computers!
We’ve Never Seen this Before

Short time to global adoption
• High standard for availability and reliability
• Continual security challenges

Rapid evolution
• Amazon / Ebay → Google → Facebook / Twitter → …
• Big engineering challenges at each step

Sensors
• GPS, camera, temperature, …

Social
• Mail, messenger, Facebook, Twitter, on-line games, …
Does Programming Need to Change?
Programming Cloud-Scale Apps

Cloud programming models do not exist
• Still using web 1.0 technology (PHP, Ruby, ASP.NET, …)
• Building scalable, reliable web services is challenging, even for experts
• Clients rapidly diversifying

Existing languages will evolve
• Programming model ≠ programming language
• Model offers paradigm for understanding and structuring problems

Challenge: enable run-of-the-mill developers to build client + cloud apps
• Throw away and rewrite service is not practical model
• Can language / runtime led developers to build scalable, reliable systems?

HPC community found they needed to rewrite systems for every 10x increase in processors.

-- Kathy Yelick
New Programming Model – New Problems (and some old, unsolved ones)

Concurrency

Availability

Security & Privacy

Parallelism

Performance

Distribution

Reliability

Power
Concurrency

Services are inherently concurrent
• Simultaneously process multiple requests
• Communicate through asynchronous messages

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 inhibits research, development, reuse, interoperability
Parallelism

Modern processors are parallel
• Increased performance + power efficiency

Processors becoming heterogeneous
• Non-isomorphic functional units

Data centers are parallel, message-passing clusters
• Cost, throughput, availability

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

Abolish shared memory!
Distributed systems are rich source of difficult problems

- Replication
- Consistency
- Quorum

Well-studied field with good solutions

- Replication
- Persistence
- Relaxed consistency

Integrate these techniques into programming model

- Libraries
- Language integration
- New models
Availability

Services must be highly available
- Become part of national infrastructure
- Blackberry/Google/Twitter… outage affect millions of people and gets national media attention

High availability is hard
- Starts with design and engineering
- 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
  - Afterthought
  - Run in inconsistent state
  - Difficult to test
Performance

System-level concern
• Extends far beyond code on single machine
• Most performance tools focus on low-level details

Build, observe, tweak, overprovision, pray, …
• Wasteful and uncertain

Scalable system grow by adding machines, not rewriting software
• Big-O notation for scalability

Architecture is starting point
• Model and simulate before building system

Adaptivity
• Performance SLAs should be specified behavior
• Systems need to be introspective and capable of adapting behavior to load
Considerable progress in past decade on defect detection tools
- Tools focused on local properties (buffer overruns, test coverage, races, etc.)
- Little work on system-wide properties

Modular checking
- Whole program analysis too expensive and impractical for services
- Assertions and annotations at module boundaries
- Check global properties locally

New defects
- Message passing
- Code robustness
- Performance bottlenecks

Scale makes hardware reliability first-class concern
- Large systems are continually failing
- Smaller semiconductor devices are inherently less reliable
Centralized computation and data creates more attractive targets
- Vandalism and fraud ⇒ theft and espionage
- But, is current self-managed model better?
  - Professional system administration

Is current system infrastructure fatally flawed?
- Should we start afresh?
- Do we know how to start afresh and do better?

Centralized data also creates incentives for misuse by service providers
- No general understanding or community standards for acceptable use of PII
- Unfortunately closely tied to (large) sums of money
Orleans

Goals
- Simple, accessible programming model
  - Higher level of abstraction
- Scalable, resilient systems
- Correct, maintainable systems
- Single model for client and server

Programming framework for client + cloud computing under development in XCG
Orleans

Key concepts

Grain
Activation
Message passing
Promises
Transaction
Library
Orleans Application Model

Challenge: scale to 10’s K nodes and provide service-level reliability on unreliable platform

- Data partitioning, service replication, automated monitoring and control

Architecture is collection of interdependent, composable services with declarative service-level agreements (SLAs)

- Comprised of one or more grain types
- Independently deployable and updatable
- SLA, combined with Marlowe performance monitoring, forms basis for resource allocation and optimization
- Transactions at request-level provide isolation and failure tolerance
Orleans Grain Model

Grains are basic unit of computation that encourages scalable cloud architectures and enables automated management

Sharded
- Data partitioned (and relocatable) among servers

Encapsulate state
- Migratable between servers and data centers for load balancing and reliability
- Replicatable for throughput and reliability

Internal single-threaded
- Simpler, less error-prone programming model
- Concurrency by replicating grains (transactional shared state)
- Facilitates migration

Strongly typed and versioned interfaces
Ad-hoc, one-off solutions raise cost and complexity of building and operating service

Operations is integral part of Orleans design
- Strongly versioned grains and services
- Run multiple concurrent version of application
- Support for single-box, test cluster, rolling deployment, and emergency rollback
- Integral monitoring and control of applications (Marlowe)
Grains

Basic unit of computation that encourages scalable cloud architectures and enables automated management

Sharded
• Data partitioned (and relocatable) among servers

Encapsulate state
• Migratable between servers and data centers for load balancing and reliability
• Replicatable for throughput and reliability

Internal single-threaded
• Simpler, less error-prone programming model
• Concurrency by replicating grains (transactional shared state)
• Facilitates migration

Strongly typed and versioned interfaces
Activations

Parallelism across requests

- Single request processed completely by one grain
- Multiple activations of a grain execute in parallel
- Each activation completely isolated
- Multi-master branch-and-merge update model for changes to persistent grain state

Reduce latency and increase system throughput
Fundamental to distributed systems

Better programming model
• Performance / correctness isolation
• Clearly-defined points of interaction
• Scalable

Little language support beyond message-passing libraries
• Fundamental mismatch: asynchronous stranger in a synchronous world
Promises

Message passing is asynchronous
Functions calls are synchronous

Promises resolve impedance mismatch

Returned on invocation of remote operation

Bind closure to be evaluated on result returned by operation
   Or wait for result
Grains process request in a transaction
• Computation either commits and leaves state consistent
• Or, terminates cleanly

Simpler than eventual consistency
• Isolate concurrent computations
• Make replicated copies consistent
• Simplify error handling

Lightweight, optimistic implementation
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 makes fundamental change in mainstream computing
- Inflection point, paradigm shift, …

New challenges need new solutions
- Parallelism / concurrency / distribution
- Reliability / availability
- Communications
- Power
- Security / privacy

Opportunity to rethink relation of people to computing
- Reduce aggravation and management burden
- Move to seamless, ubiquitous computing
- Develop new, natural user interfaces