Programming the Cloud

James Larus

eXtreme Computing Group, Microsoft Research

PPoPP / HPCA

February 14, 2011
Client + Cloud Computing

- New computing platform
  - Ubiquitous access
  - Inherently distributed
  - Many, diverse clients (single function → rich)
  - “Unlimited” computation and storage, on demand
Cloud – or Fog?
Evolution
New Hardware → New Software → New Experiences
Quantitative Becomes Qualitative

• Dennard scaling ("Moore’s Law") looks to end within a few years
  • Transistors no longer running faster
  • Soon transistor count will stop increasing
• Finally, break von Neumann bottleneck
  • Parallelism at all levels: circuits, processors, systems
  • Distributed too
• Heterogeneous, diverse world
  • Processors
  • Client devices
  • Sensors
  • Programming languages
• Communication as important as computation
  • For users as well as computers!
We’ve Never Seen This Before, cont’d

- Rapid global adoption
  - High standard for availability and reliability
  - Continual security challenges

- Rapid evolution
  - Amazon / EBay → Google → Facebook / Twitter
  - Unprecedented engineering challenges at each step

- Sensors
  - GPS, camera, temperature, ...

- Social
  - Mail, Messenger, Facebook, Twitter, on-line games, ...
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, …)
  • Lack makes building scalable, reliable web services challenging, even for experts
  • No integration with rapidly diversifying clients

• Challenge: enable run-of-the-mill developers to build client + cloud apps
Client + Cloud System

- Sensors
- Client
- Network
- Service
- Servers
- Storage
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

• Processors becoming heterogeneous
  • Non-isomorphic functional units

• 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!
Communications

• Data movement and placement is fundamental concern
• Network bandwidth, latency, 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
Performance

• System-level concern
  • Extends far beyond code running on single machine
  • Most performance tools focus on low-level details (CPU, cache, ...)

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

• Scalable system must 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
Power

- Power constraints at both extremes
- Data centers
  - Limited by available power
  - Costs heavily influence by power and cooling (CapEx and OpEx)
- Clients
  - Battery constrains computation and communications
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
Centralized computation and data creates attractive targets
- Vandalism and fraud ⇒ theft and espionage
- Is current, self-administrator model 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

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

• Programming framework for client + cloud computing
  • Under development in XCG
• Goals
  • Simple, accessible programming model
  • Scalable, resilient systems
  • Correct, maintainable systems
  • Single model for client and server
Orleans

- Key concepts
  - Grains
  - Activations
  - Message passing
  - Promises
  - Transactions
Grain - Unit of Computation

• Encourage scalable architecture and enable automated management
  • Sharded data partitioned among servers
  • Encapsulation allows migration for load balancing and reliability
    • Replication for throughput and reliability
• Internally single-threaded
  • Simple programming model
  • Concurrency through replicated grains (transactional shared state)
• Strongly typed and versioned interfaces
Activation - Parallelism Across Requests

- Multiple activations of a grain execute concurrently
  - Single request processed completely by one grain
  - Each activation isolated
- Use to reduce latency and increase system throughput
- Multi-master branch-and-merge updates for persistent state
Message Passing - Asynchronous Stranger in a Synchronous World

- Fundamental in distributed systems
- Better, scalable programming model
  - Performance / correctness isolation
  - Clearly-defined points of interaction
- Little language support beyond message-passing libraries
  - Message passing is asynchronous
  - Functions calls are synchronous
Promises - Resolve Impedance Mismatch

• Bind future computation to future result
  • Remote operations return promise
  • Caller binds closure that will be evaluated when result appears
    • Can block and wait for result (bad)
• Makes concurrency explicit
• Mechanism for handling remote errors
Transactions – Consistency and Error Handling

• Grains process request in a transaction
  • Computation either commits and leaves state consistent
  • Or, terminates cleanly
• Goals
  • Isolate distinct computations
  • Consistent (replicated) activations
  • Simplify error handling
• Lightweight, optimistic implementation
  • Less than serializable
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
Application Model

• Challenge: scale to 10K’s nodes and provide service-level reliability on unreliable platform
  • Data partitioning, service replication, automated monitoring and control

• Orleans architecture is collection of interdependent, composable services with declarative service-level agreements (SLAs)
  • Each built from one or more types of grain
  • Independently deployable and updatable
  • SLA, with Marlowe performance monitoring, supports dynamic, adaptive resource allocation and optimization
Operation Model

• Deployment, monitoring, maintenance are major challenges in building and running cloud service
  • Ad-hoc, one-off solutions raise cost and complexity of building and running a service

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

• New challenges require new solutions
  • Parallelism / concurrency / distribution
  • Reliability / availability
  • Communications
  • Power
  • Security / privacy

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