ORLEANS: CLOUD PROGRAMMING FOR EVERYONE

James Larus
Microsoft Research

Bell Labs
9/30/11
21st century implicit and natural computing

- Increasingly natural interfaces
- Embedded intelligence in everyday objects
- Ubiquitous network access and cloud services
**What Is Changing?**

- System on a chip
  - Powerful mobile devices
- Graphics processing units
  - High quality graphics
- Explosive data growth
  - Ubiquitous sensors and media

- Inexpensive, embedded computing
  - Smart objects, CIP, …
- Wireless spectrum pressure
  - Mobile device growth
- New software models
  - Social networks, clients+clouds …
Cloud Computing = Clients + Cloud

- Next computing platform
  - Functionality split between client and cloud
  - Multiplicity of devices
  - “Unlimited” compute and storage on demand in cloud
- Augmentation not substitution
  - Smartphone + tablet + PC
  - Many devices, some specialized
- Buy services, not bits
Cloud as Glue for Multiple Devices
CHALLENGES OF CLOUD COMPUTING

- Inherently parallel and distributed
  - Many servers, possibly in multiple data centers
  - Diverse clients with disparate capabilities

- Unreliable, failure-prone platform
  - Commodity computers and communication links
  - Non-negligible failure rates with complex failure modes

- Services, not stand-alone applications
  - Varying and unpredictable loads
  - High availability and reliability

- And, historic challenges of constructing secure, reliable, scalable, elastic, and efficient software
WHAT’S MISSING FROM PROGRAMMING MODELS & LANGUAGES?

- First-class support for asynchronous communications
- Appropriate concurrency models
- Data consistency models
- Support for data partitioning or computation replication
- Distributed, adaptive monitoring and control
ORLEANS

- Software framework and runtime to make cloud programming easier and more productive
  - Experimental system from Microsoft Research
- Shift burden of correctness and performance from developer to Orleans system
- Radically simplified, prescriptive programming model
TARGET AUDIENCE

- Novice cloud software developers, unfamiliar with distributed systems
- Rapid development of cloud applications
ORLEANS OVERVIEW

- Grains
- Grain activations
- Messages
- Promises
- Transactions
- Adaptive performance
- Persistence
**Orleans is Actor Based**

### Customer Grain

- **Methods**
  - Checkout
  - AddProduct
  - RemoveProduct

### Grain (actor)

- **Field**
  - **Value**
    - Name: “John Doe”
    - Email: “john.doe@hotmail.com”
    - Address: “123 Main St., Anywhere UR 01234”
    - Products

### State

- **Message Queue**

### Messages

- **Message Queue**

- **Methods**
  - Checkout
  - AddProduct
  - RemoveProduct
Why Actors?

- Fine-grain distributed objects are natural abstraction
- Grains partition data
- Enable computation replication
- Secure and isolated computation with clear points of communications
  - Singularity
- Natural integration with persistent storage
  - Grain resides on disk until activated
**Appropriate Size of a Grain**

Reducing grain overhead enables fine-grain objects
Share communication channels and OS threads among grains in silo (process)

- **Chirper**
  - Simplified Twitter-like system
  - Less than 200 lines of application code

- **Horton**
  - Distributed graph database

- **Sparse Linear Algebra**
  - Eigenvectors of large matrices (1Billion x 1Billion)
  - PageRank calculation

**Graph partition**

**User Message Address book**

**Messaging intensive**

**Data intensive**

**Compute intensive**

**Processor**
<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>“Canon EOS T3i”</td>
</tr>
<tr>
<td>Number</td>
<td>B004J3V90Y</td>
</tr>
<tr>
<td>Quantity</td>
<td>12345</td>
</tr>
<tr>
<td>Price</td>
<td>$800.0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>“Canon EOS T3i”</td>
</tr>
<tr>
<td>Number</td>
<td>B004J3V90Y</td>
</tr>
<tr>
<td>Quantity</td>
<td>12345</td>
</tr>
<tr>
<td>Price</td>
<td>$800.0</td>
</tr>
</tbody>
</table>

Grain ID 1

Grain ID 2

Grain ID 1

Grain ID 2
Replication

- Scalability arises from partitioning and replication
  - Grains encourage partitioning
  - Orleans runtime replicates grains
- Concurrently process independent requests
- Results in shared (persistent) state
  - Handle with transactions and reconciliation
IN-GRAIN PROGRAMMING MODEL

<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>“John Doe”</td>
</tr>
<tr>
<td>Email</td>
<td>“<a href="mailto:john.doe@hotmail.com">john.doe@hotmail.com</a>”</td>
</tr>
<tr>
<td>Address</td>
<td>“123 Main St., Anywhere UR 01234”</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Products</th>
</tr>
</thead>
<tbody>
<tr>
<td>Grain ID₁</td>
</tr>
<tr>
<td>Grain ID₂</td>
</tr>
</tbody>
</table>
Radically Simple Programming Model

- Eliminate shared-memory parallelism
  - Distributed concurrency is (usually) sufficient
- Explicit asynchrony
  - Explicit error-propagation and handling
- Simplify challenging aspect of programming
  - Force developers to use restricted, less error-prone form of parallelism
  - Discourage synchronous communication because of effects on scalability and performance
- Facilitate automatic grain replication
  - Transparent performance improvement
SYSTEM IS COMPOSED OF GRAINS
Silo (process)
CONCEPTUAL MODEL

WHAT COULD GO WRONG?
CONFLICT
CONSISTENCY
FAILURE
TRANSACTIONS

Isolation

Atomicity

Consistency
**Why Transactions**

- **Isolation**: grain activations running in transaction are isolated from grains responding to other requests
  - Appears as only one request being processed

- **Atomicity**: computation either successfully completes, or no persistent state changes
  - If computation fails, original request can re-execute

- **Consistency**: only one activation of grain allowed in transaction
  - Multiple messages to grain see consistent view of its state
  - Persistent state of all grains in transaction written atomically
CHALLENGES

- Efficiency and scale
- Distributed transactions are notoriously difficult and expensive
- Use simple, non-serializable transactions
- Avoid single point of conflict

- Not serializable!
  - Local properties (atomic writes), but no global guarantees across grains in transaction or between transactions
AND, OVERLOAD ...
ADAPTIVE PERFORMANCE MANAGEMENT

Grain Placement

Grain Migration
AUTOMATIC PERFORMANCE TUNING

- Measure performance of every grain
  - # of requests, latency, throughput
- Create more activations if needed
- Shed load from overloaded servers by moving activations
- Transparent to application
  - Possible due to grain programing model
PERSISTENCE
ACTIVATION STATE RECONCILIATION – BRANCH & MERGE

- Branch revisions for activations
- Merge activation changes into store
- Update activations
- Activations accept & acknowledge updates
- Activations forget change deltas
Merging

- Many policies for merging conflicting changes
  - Default “last writer wins”, atomic across grain
- Library of richer data structures
  - Set, tree, hashtable
  - Consistent semantics for distributed changes
  - cf Marc Shapiro, INRIA
Chirper throughput grows **linearly**

Scaling by automatically creating **multiple activations** of heavy subscribers
CONCLUSION

- Orleans provides explicit support for key aspects of cloud software
  - Communications
  - Concurrency
  - Data partitioning
  - Computation replication
  - Error handling
  - Performance monitoring and control
- Shifts burden from developer to Orleans
- Make the cloud accessible to everyone