WHO AM I?

- Software Development Manager for Berklee College of Music online learning platform
- Basho Technologies, developer of Riak database
- Worked on roughly 6 different NoSQL databases

Academia (2016+)
- Ph.D. candidate based in Portugal & Belgium
- Creator of the Lasp programming system for large-scale, asynchronous distributed computing
- Contributor to Microsoft Orleans distributed computing framework
1. **Implement version 1.0 features using a traditional architecture.**

   We demonstrate implementing features of our application using Martinelli with a traditional three-tier architecture.

2. **Implement version 2.0 features using an ideal architecture.**

   We demonstrate implementing features of our application using Martinelli with an ideal peer-to-peer highly-available architecture.

3. **What is Martinelli?**

   Martinelli is a *fictional* programming language demonstrating an “ideal” in distributed programming language design.

   We present the principles behind Martinelli and the techniques and tools that Martinelli uses to achieve this.
APPLICATION

What features should our application have and how do we build it?
V1.0 FEATURES

Our core feature set for our mobile application, designed using existing technologies and using a traditional data center focused design.

Users upload photos of bottles of wine they enjoy through app.

Data is processed with ML/AI algorithm to identify and classify.

Recommendations are created using an iterative algorithm.
TRADITIONAL ARCHITECTURE

- Communication through data center
- Application servers run business logic
- Clients must be online to operate

Analysis

- Application is easy to program
- Exhibits high latency (non-native)
- Exhibits low availability (DC-focused)
CODE: PHOTO UPLOAD

**Server**

```
Server
database :photos
on_photo do |user_id, photo|
  photos[user_id].add(photo)
end
```

Key-value store to store photos: in essence, a map.

**Client**

```
Client
database :photos
on_photo do |photo|
  upload(user_id, photo)
end
```

Every time a photo is taken, upload it to the server.

Store each photo in a map indexed by user.
CODE: RECOMMENDATIONS

Server

database :recs
database :favorites

process do |user_id in users |
  classify(photos[user_id], favorites[user_id])
end

process do |user_id in users |
  recommend(favorites[user_id], recs[user_id])
end

Client

database :recs
database :favorites

process do
  refresh(user_id)
end

process do
  render(recs)
end

Key-value stores for recommendations and favorites.

One process per user to classify photos into favorites.

One process per user to recommend based on favorites.

Process keyword defines a concurrent process that keeps executing.
V2.0 FEATURES

Features we would like to add to our application in the near future to enable a better experience for our users

[Fully offline]
Recommendations while offline
- Use local information when offline
- Augment information and refine recommendation when online using available local information; augment recommendations when online

[Partially offline]
Share and modify both favorites and recommendations with friends when offline

[Online]
Purchase wine off of your recommendation list with transactional guarantees
IDEAL ARCHITECTURE

- Application code at the edge
- Peer-to-peer communication redundancy
- Application is transactional

Analysis

- Application is hard to program
- Exhibits low latency
- Exhibits high availability
process do |user_id in users|
  refine(photos, favorites)
end

process do |user_id in users|
  refine(favorites, recs)
end

database :photos,
  :replicated => :user_id

database :favorites,
  :replicated => true

database :recs,
  :replicated => :user_id
on_purchase do |user_id, wine|

    atomic do

        recs[:user_id].remove(wine)
        perform_purchase(user_id, wine)

    end

end

Wrap operations in an atomic block.
“While you’re at it, why don’t you try my Martinelli?”
– Tom Frost, Naked Lunch
WHY MARTINELLI?

Why do we need a language like Martinelli?

Techniques for v2.0 features exist only in isolation
- Systems, algorithms, etc.

Development largely addressed from a systems composition perspective
- Kafka, to Hadoop with Spark, etc.

Programmers responsible for “gluing” services together at boundaries

An ad-hoc programming model
- Weak semantics
- APIs define the “programming language”
HISTORICALLY

What can history tell us about distributed programming languages?

[Well designed, no adoption]
Pure approaches, new runtime and language
- Argus (Liskov et al. 1986)
  - Transactional support, fault-tolerant handling of RPCs
- Emerald (Black et al. 1986)
  - Objects with object migration; separation of typing/implementation

[Poorly designed, high adoption]
Retrofitting existing systems
- CORBA (OMG, 1991)
  - Leverage existing language semantics, make distribution transparent to the user
  - Cross-language, cross-system, cross-architecture
Language for building applications on top of composed systems

- Not possible to reimplement all existing systems into a new runtime
- Composition, “glue” can be independently verified via existing techniques (LDFI 2015, etc.)

Fault-tolerant, highly available infrastructure for application execution

- Peer-to-peer, client-side application execution and data replication

Programming model designed for distributed applications

- Restricted language semantics depending on the network topology and environment the application is being deployed in

What exactly is Martinelli?
MARTINELLI ARCHITECTURE

- Application code at the edge
- Peer-to-peer communication redundancy
- Application is transactional

Analysis

- Application is easy to program
- Exhibits low latency
- Exhibits high availability

Meets all three of our criteria!
What are the techniques and methods Martinelli leverages?
Peer-to-peer topologies widely studied and successful, examples:
- Kademlia (BitTorrent)
- Lasp, Cassandra (HyParView)

Provide greater redundancy and efficient management of state and communication links.

Eliminate the need for a central coordination point.
APPLICATION MIGRATION

Applications (and their data) must be migrated to the edge to exploit local operation and low latency interactions.

- Edge computing moves app to the device
  - Provides a better experience to user

- Today, this implementation must be duplicated, implemented twice

Promising approaches:
- Portable VMs
- Architecture specific code targeting
- Program slicing
Concurrency is problematic for large-scale distributed applications.

Concurrent operation may generate conflicts
- How to pick “winning” update?
- How to present conflicts to the end user?

Edge introduces additional concurrency
- False concurrency (efficient tracking, false positives)
- Modifying stale data (conflicts from staleness)

Specialized data structures:
- Conflict-free Replicated Data Types
- Operational Transformations
- Cloud Types
- Mergeable Data Structures
ATOMIC OPERATIONS

Transaction protocols typically provide both atomicity and isolation for groups of updates.

Transactions provide ACID
- Atomicity (A): indivisible groups
- Isolation (I): sequentiality of groups

Distribution makes it difficult
- 2PC: fault-tolerant atomic commitment
- 2PL: isolation (serializability), but locking problematic under partition

Promising approaches:
- Distributed Sagas: atomicity, no isolation
- MSFT Orleans: 2PL/2PC at single-DC scale
- Cure: causal, weak isolation, atomicity for geo-scale
THE FUTURE

Where are we and what's next in distributed language design?
Independent solutions:

**Lasp** connects Erlang systems together using a safe distributed programming model on very large P2P clusters (1024+ nodes)

**Legion** provides P2P client interactions for vanilla JavaScript apps with CRDTs and Google AppEngine

**SwiftCloud** provides causally consistent transactions at the client

**Erlang** VM has been ported to extremely low-power computing devices enabling application migration to the edge

**MSFT Orleans** provides 2PL/2PC transactions at geo-scale

**LDFI** verifies fault-tolerance under composition; latter, for application invariants under weak ordering

**Martinelli-like languages** with peer-to-peer interactions, application code at the edge, transactional guarantees and convergent-by-design programming models.
MOVING FORWARD

We’ve seen new greenfield systems fail to gain adoption
• CORBA vs. Argus, Emerald, etc.

Therefore, we must strive to build research solutions that leverage existing tools
• Orleans with Microsoft CLR, CRDTs in Riak, Legion with Google AppEngine

Systems centric approach provides a weak foundation
• Weak semantics, hard to make guarantees about composition correctness

Strive for higher-level abstractions via programming languages and models
• Strong semantics, focus on writing applications and not gluing services together
THANKS!

Christopher S. Meiklejohn

Instituto Superior Técnico
Université catholique de Louvain

Velocity, London 2017