Journées GDR GPL

Engineering Scale:
Software and Distribution for Tomorrow's World

François Taïani
Ideal software artefact

- structured, predictable, open, evolvable
A Distributed System Today ...

External services

facebook
twitter
bit.ly

Standards

OAuth
JSON

External developers

Geosocial app, est. 2009

t foursquare

Middleware

mongoDB

45M Users
Today's distributed systems

➔ sprawling, chaotic, complex, unmanageable?
Outline

- A call to arms: engineering large scale
- Examples of ways forward
Outline

- A call to arms: engineering large scale
- Examples of ways forward
Today's distributed systems

⊂ sprawling, chaotic, complex, unmanageable?


Sprawling
Complex

Portability

Interoperability

Transparency

...
Unmanageable?

- **Globus client**
  - 1 creation, 4 requests, 1 destruction

- **Projection w.r.t.**
  - stack depth
  - package

**client:** 1,544,734 local method call (sic)
**server:** 6,466,652 local method calls (sic) [+time out]

The Impact of Web Service Integration on Grid Performance. Taïani, Hiltunen, Schlichting, HPDC-14, 2005
Netflix never used its $1 million algorithm due to engineering costs

By Casey Johnston | Published April 13, 2012 4:25 PM

Netflix awarded a $1 million prize to a developer team in 2009 for an algorithm that increased the accuracy of the company's recommendation engine by 10 percent. But today it doesn't use the million-dollar code, and has no plans to implement it in the future, Netflix announced on its blog Friday. The post goes on to explain why: a combination of too much engineering effort for the results, and a shift from movie recommendations to the "next level" of personalization caused by the transition of the business from mailed DVDs to video streaming.
Netflix never used its $1 million algorithm due to engineering costs

By Casey Johnston | Published April 13, 2012 4:25 PM

Netflix awarded a developer team in 2009 for an algorithm that increased the accuracy of the company's engine by 10 percent. But today it doesn't use the million-dollar code, and has no plans to implement it in the future, Netflix announced on its blog Friday. The post goes on to explain why a combination of too much movie recommendations to the "next level" of personalization caused by the transition of the business from mailed DVDs to video streaming.

$1 million prize
recommendation
too much engineering effort
Netflix never used its $1 million algorithm due to engineering costs

By Casey Johnston | Published April 13, 2012 4:25 PM

Netflix awarded a developer team in 2009 for an algorithm that increased the accuracy of the company’s movie recommendation engine by 10 percent. But today it doesn’t use the million-dollar code, and has no plans to implement it in the future, Netflix announced on its blog Friday. The post goes on to explain why a combination of too much engineering effort to the “next level” of personalization caused by the transition of the business from mailed DVDs to video streaming.
Why is distribution hard?

- **Information** takes **time** to travel
  - Some DS protocols inspired from general relativity

- Machines and networks **fail**
  - If MTTF 4 years: 1M machines $\rightarrow$ 1 failure every 2 minutes
Impossibility Results

Asynchronous system with crash failures

- **Consensus** impossible (even if only one node crashes)
- Consistency + Availability + Partition tol. Impossible

Consequences

- N crash prone machines not Turing complete

Progress so far: Middleware

- Goal: "nice" programming abstractions
  - Challenge: to hide or not to hide distribution?

![Diagram showing distributed applications, middleware, network OS services, and kernels on machines A and C connected by network.]
In Practice

![Diagram showing the layers of an operating system with App, MW, Network OS Services, and Kernel layers.]

F. Taiani
Most of today's effort centred on programming nodes
Tomorrow's systems will require a holistic approach.
The Holistic Challenge

- (Strong) **consistency** is very **costly**
  - The **one-entity** metaphor only goes so far.

- **Large scale**: embrace an **inconsistent** world
  - Co-existence of past and present in the same system
  - Partial adaptation
  - Emerging behaviour

- **Challenges**
  - Programming Models
  - Interoperability
  - Safety
  - Security
Outline

- A call to arms: engineering large scale
- Examples of ways forward
Example 1

Dionasys project (2014-2017)

- Target
  - Large scale, heterogeneous systems
  - E.g. IoT + cloud + VANETs + mobiles

- Aim
  - Principled holistic SE approach

- Tools
  - Self-stabilizing overlays
  - Declarative language
  - Components

Jelasity et al. [JMB09]
Example 2

Application of components + DSL to gossip protocols

⇒ Whisper + GossipKit

Gossip Protocols

- Historical Distributed System
  - Deterministic with strong guarantees
  - Does not scale well

- Gossip (aka epidemic) Protocols
  - Introduce some ‘chaos’
  - Goal: system to converge to a desirable outcome
  - But some nodes might be left out

- Trading determinism for scalability & robustness
Gossip Protocols (cont.)

- **Principles**
  - leverage **rumour-like** propagation of information
  - large applicability: aggregation, broadcast, clustering
  - often **composed** to realised higher-level services

- **Conceptually simple**
  - typically symmetric behaviour
  - key notions of **state**, information **flows**, and **decisions**

- **But implementation can be time consuming**
  - multithreading, distributed coordination, network intricacies, co-existence
Applying Components to Gossip

- Component successfully applied to distributed systems
  - industry: EJB, CCM, OSGi, SCA
  - research: Fractal, OpenCOM, FraSCAti
  - middleware Frameworks: GridKit, Rapidware, Ensemble, Cactus, Open Overlays

- Clear **structure**, explicit **dependencies**

- **Benefits**
  - 😊 promote **reuse**
  - 😊 easily **composable** and **configurable** (SPL..)
  - 😊 lend themselves to **runtime reconfiguration**
The problem with components

- Drawbacks
  - 😞 low intelligibility (where is the intent?)
  - 😞 conceptual mismatch for developers focusing on behaviour
  - 😞 high learning curve for unfamiliar frameworks

Components tend to focus on structure, not behaviour.
Applying SDL to Gossip

- **Spec. lang. and DSL:** High-level per node description
  - Lotos, Estelle, PLAN-P, Mace …

- **Macro-programming:** system as one entity
  - E.g. Kairos, Regiment, TinyDB, MIT-Proto
  - centralised shared-memory parallel abstraction
  - main program compiled into code for each node

- **Benefits**
  - high level of abstraction (in particular for macro-prog)
  - intelligible
  - good conceptual match for developers looking at behaviour
Behaviour rather than structure

Can we build a hybrid approach that combines the strengths of components & high-level languages?

- Drawbacks

😊 we loose the benefits of components (reuse, adaptation, …)
structure + behaviour = ?

- **tangling** behaviour & structure
- ‘breaks’ **encapsulation**
- tension **flexibility** vs. scattering

- **complex** composition
- tension structural needs vs. programmatic ones
structure + behaviour = ?

encapsulation

orchestration

bake

synthesis

transparent componentisation
Transparent Componentisation

- **Separation of concern** between behaviour / structure
- **Developers** can focus on **high level logic**
- **Systems** takes care of **modularity**, reuse, and evolution

- 🌟 simple
- 🌟 concise
- 🌟 high-level
- 🌟 modular
- 🌟 reusable
- 🌟 (re)configurable

*F. Taiani*
The WhispersKit Architecture

GossipKit Framework

Metamodel

Abstract Model

Runtime

Gossip Developer

Whipsers

- Configuration Description
- Component Architectural Abstraction
- Reconfiguration Management
- Event-driven Architecture
- OpenCom Component Framework
The WhispersKit Architecture

GossipKit Framework

Metamodel

- Configuration
- Description

Abstract Model

- Component Architectural Abstraction
- Reconfiguration Management
- Event-driven Architecture

Runtime

OpenCom Component Framework

Gossip Developer

Whipsers
A component **framework** for **epidemic** protocols

- based on analysis of 30 gossip protocols
- **event-based**
- **XML-based configuration** for component composition
- targets abstraction, modularity, reuse, evolvability
GossipKit Examples

RPS

Wireless broadcast

Anti-Entropy

T-Man

SCAMP
The WhispersKit Architecture
Whispers

- macro-programming language for gossip protocols
  ➞ system as one entity

- primitives

```javascript
protocol {...}  // protocol block
every (time) {...}  // periodic behaviours
wait (Event e type T) {...}  // reactive behaviours
foreach (n in nodeSet)  // distribution
  synchronised {...}  // pairwise data exchange
State state = new State[fields][size] ;  // state decl.
state.field ;  // get a column of data
state.add([fields])  // add
state.remove(row_ID)  // remove
i.RandomStateCompress(...)  // library call
```
Whispers Example: RPS

RPS {
    State sample = new State[Node:PeerID][Size=5];
    Node n, i;
    every (5000) { // do the following every 5000 ms
        foreach (n in AllNodes) { // for each node n
            i = n.RandomPeerSelection(n.sample)[Size=1];
            n.sample.add([n]);
            i.RandomStateCompress(i.sample,n.sample)[Size=5];
            n.RandomStateCompress(i.sample,n.sample)[Size=5];
        } // end of foreach
    } // end of every
} // end of RPS protocol block

The WhispersKit Architecture

GossipKit Framework

Metamodel

Abstract Model

Runtime

Configuration Description

Component Architectural Abstraction

Reconfiguration Management Event-driven Architecture

OpenCom Component Framework

Gossip Developer

Whispers
```java
State sample = new State[Node:PeerID][Size=5];
Node n, i;
every (5000) { // do the following every 5000 ms
    foreach (n in AllNodes) { // for each node n
        i = RandomPeerSelection(n.sample)[Size=1];
        n.sample.add([n]);
        n.RandomStateCompress(i, n.sample);
    }
} // end of foreach
} // end of every
} // end of RPS protocol block
```

**Compilation**

1. **Periodic thread**
   - **every 5 seconds**
   - **Node neighbour = RandomPeerSelection**
   - **retrieve local_sample**
   - **push local_sample to neighbour**

2. **Reactive thread on receipt of a message**
   - **message push**
   - **message reply**
   - **retrieve local_sample**
   - **reply local_sample**
   - **extract remote_sample from message**
   - **RandomCompress (local_sample, remote_sample)**
Distributed Reconfiguration

- A developer describes new behaviour in Whispers.
- The platform uses component representation to compute minimal set of changes;
  to propagate and enact reconfiguration.

System Behaviour A

- Component mapping
  - C1
  - F1
  - S1
  - S2

Component Configuration A

Unbind C1 and S1
Unload S1
Replace C1 by C2
Replace Net1 by Net2

Component Configuration B

System Behaviour B

- Component mapping
  - C2
  - F1
  - Net2
  - S2

Transparent reconfiguration
Distributed Reconfiguration

Example: RPS \(\rightarrow\) T-Simple (Ring) \(\rightarrow\) T-Simple (Grid)

- coarse grained
- fine grained

Figure 5.6: Initial random graph maintained by RPS
Figure 5.7: 5th rounds since 1st reconfiguration
Figure 5.8: Ring constructed at the 11th round

Figure 5.9: Topology at the 20th round
Figure 5.10: Grid constructed at the 23rd round
Conclusion

- The world is distributed, the world is large.
- Distribution is more than concatenation:
  - Failures and uncertainties
- Large-scale distributed systems even more so:
  - Information takes time to travel
- Novel software engineering approaches needed:
  - Away from node-centric view
  - Holistic yet loosely coupled approaches ideal
Thank you
Task Failures at Google

Figure 3: Task-eviction rates and causes for production and non-production workloads. Data from August 1st 2013.

Source: Large-scale cluster management at Google with Borg
Abhishek Verma, Luis Pedrosa, Madhukar R. Korupolu, David
Oppenheimer, Eric Tune, John Wilkes
EuroSys'2015, Bordeaux, France (2015)


(Some) References


