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

Geosocial app, est. 2009

Middleware

MongoDB

Standards

External developers

45M Users

Middleware

Flume

Amazon web services
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?

Google* | 900,000
Microsoft* | 518,000
HP/EDS* | 380,000
OVH | 100,000
Intel | 100,000
SoftLayer | 86,000
Akamai Technologies | 84,000
Intel | 75,000
Rackspace | 74,028
1&1 Internet* | 70,000
GoDaddy* | 70,000
Facebook | 60,000
eBay* | 50,000
The Planet | 48,500
Amazon EC2 | 40,000
LeaseWeb | 36,000
Interigen (PlusServer/Server4You) | 30,000
SBC Communications | 29,193
Verizon | 25,788
Time Warner Cable | 24,617
HostEurope | 24,000
AT&T | 20,268
Peer 1/Serverbeach | 10,227
iWeb | 0,000


Sprawling
one RPC request,
• **2065** individual invocations
• > **50** C-functions
• > **140** C++ classes

Source: [TKF2009]
Unmanageable?

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

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

client: \(1,544,734\) Java method calls (sic)
server: \(6,466,652\) Java 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 its one million-dollar prize to 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 and too much personalization caused Netflix to transition 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 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 the combination of too much complexity and movie recommendations 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 → 1 failure every 2 minutes

F. Taiani
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?
In Practice
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

F. Taiani 23
Example 2

Application of **components** + **DSL** to **gossip** protocols

⇒ Whisper + GossipKit


"Traditional" Distributed Systems
- 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
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 lose 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

F. Taiani
Transparent Componentisation

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

- bake
- synthesis
- behaviour
- structure

- simple
- concise
- high-level
- modular
- reusable
- (re)configurable
The WhispersKit Architecture
The WhispersKit Architecture

GossipKit Framework

Metamodel

Abstract Model

Runtime

Configuration Description

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

Anti-Entropy

Wireless broadcast

SCAMP
The WhispersKit Architecture
Whispers

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

- primitives

```plaintext
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 - macro-programming language for gossip protocols

- system as one entity

- primitives

```plaintext
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
  - Configuration
  - Description

- Abstract Model
  - Component Architectural Abstraction

- Runtime
  - Reconfiguration Management
  - Event-driven Architecture
  - OpenCom Component Framework

Gossip Developer

Whispers

bake


Compilation

```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
        RandomPeerSelection(n, sample)[Size=1];
        n.sample.add([n]);
        RandomStateCompress(i, i);
    } // end of foreach
} // end of every
} // end of RPS protocol block
```

---

### Periodic thread

1. `every 5 seconds`
2. `Node neighbour = RandomPeerSelection`
3. `retrieve local_sample`
4. `push local_sample to neighbour`

---

### Reactive thread on receipt of a message

5. `retrieve local_sample`
6. `reply local_sample`
7. `extract remote_sample from message`
8. `RandomCompress(local_sample, remote_sample)`

---

**Gossip (Push)**

1. **Periodic Trigger**
2. **Random Peer Selection**
3. **State**
4. **TCP**

**Gossip (Reply)**

5. **Random Compress**
6. **Reply**

---

**bake**
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.
Distributed Reconfiguration

Example: RPS → T-Simple (Ring) → 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
Figure 3: Task-eviction rates and causes for production and non-production workloads. *Data from August 1st 2013.*
(Some) References


(Some) References


(Some) References