Human-in-the-Loop Simulation of Cloud Services

Nikolaos Bezirgiannis[1] Frank de Boer[2] **Stijn de Gouw**[3]

[1]Leiden Institute for Advanced Computer Science, Leiden, Netherlands

[2]Centrum Wiskunde & Informatica (CWI), Amsterdam, Netherlands

[3]Open University, Netherlands

OUrsi
April 24, 2018

Background

General background

- Work carried out in EU FP7 Envisage project for leveraging cloud service-level agreements (SLA) into software models and resource management
- Published @ ESOCC 2017 (best paper award)

Background

General background

- Work carried out in EU FP7 Envisage project for leveraging cloud service-level agreements (SLA) into software models and resource management
- Published @ ESOCC 2017 (best paper award)

Motivation

- Train DevOps engineers in managing cloud services
- Training by real-time interactive simulation
- Simulation input taken from real-world logs
- Use feedback from executable SLA monitors for scaling

Outline

Outline

The ABS Language

- statically-typed executable modeling language
- two tiers: Java-like OO layer, Haskell-like functional layer
- concurrency model based on actor-model
- resource-aware modeling
- amenable for (semi)-automated tool-supported analyses: *simulation, testing, verification, deadlock analysis, cost analysis, deployment synthesis*

Concurrency in ABS

- Objects partitioned in concurrently executing groups
- Inside groups 1 active task (each group owns a task queue)
- Async. method call create new task in the group of callee
- Future to store/retrieve results of async call
- Cooperative scheduling: objects release control *explicitly*

```
Int m1() {
   suspend; // suspend task unconditionally
   ...
}
Int m2(I obj) {
   Fut<Int> f = obj ! m1();
   await f?; // release proc. if f is not resolved
   ...
}
```

Modeling time resource

ABS extension that allows *process-based simulation*:

```
await duration(min,max);
```

Reschedule process for execution after `min` but before `max` time steps

Clock

- Symbolic (Abstract) clock ⇒ computer simulation
- Real (hardware) clock ⇒ user-interactive simulation

Cloud resources

Virtualized Resources:

- Cores
- Speed
- Memory
- Network Bandwidth

Modeling cloud systems

```
// Infrastructure provider offers VMs
CloudProvider cp =new AmazonCloudProvider(params);

// Adding new VM type with resources
cp.addInstanceDescription(Pair( "m4_xlarge_eu",
  map[Pair( CostPerInterval , 239 ),
      Pair( Cores , 4 ), Pair( Speed , 13 ),
      Pair( Memory , 16000 )]));

// Launch new VM
DC m4_xl = cp.launchInstanceNamed("m4_xlarge_eu");

// Create and deploy objects on VM
[DC: m4_xl] new Qserver(80);

// Resources consumed by annotated statements
[Cost: 3] stmt; // consumes 3 ‘‘Speed’’ resources
```

I/O through a REST API

```
interface IMonitoringService {
  Unit addMS(Rule rule);
  Unit removeMS(Rule rule);
  [HTTPCallable] List<String> getHistory();
  [HTTPCallable] Unit executeScaleAction(Int i);
}

{
 [HTTPName:"monitor"] IMonitoringService ms =new
     MonitoringService();
}
```

Exposes object `ms` as HTTP endpoint, allows calling methods

```
curl -G http://localhost/call/monitor/getHistory
```

Outline

The Haskell-ABS backend

- Compiler translating ABS to Haskell
- Two run-time systems:
  - Parallel run-time: objects communicate through shared-memory
  - Distributed run-time: communication through TCP/IP

Support for time resource

Simulation Clock: real-time hardware clock of the OS

await duration: spawns a new thread which re-schedules the process after `sleep()` system call

Unit-of-time: configurable as run-time (CLI) option (s,ms,ns)

Support for Cost resource

`[Cost: intExp()]` annotation ⇒ `thisDC.executeCost(intExp());`

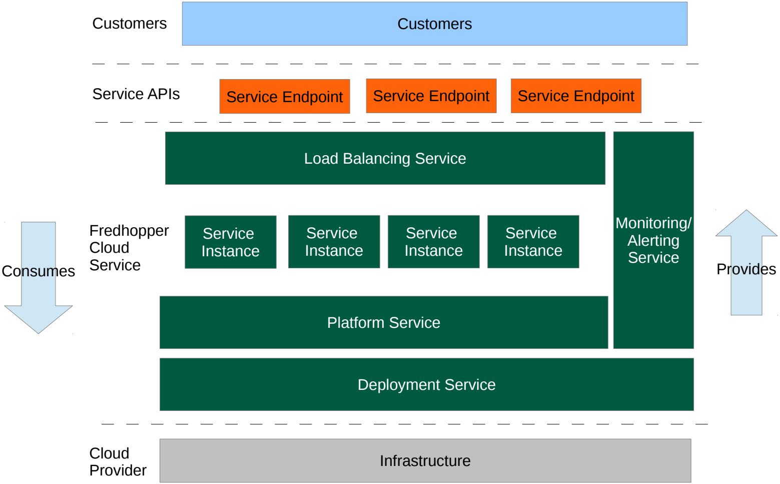
```
Unit executeCost(Int cost) {
    Int remaining = cost;
    while (remaining > this.instrPS) {
      await duration(1,1); // sleep 1 time unit
      remaining = remaining - this.instrPS;
    }
    Rat last = remaining / this.instrPS;
    await duration(last,last);
  }
```

Outline

The Fredhopper company (FRH)

- Product-catalog database&search as-a-service
- Horizontally-scalable SaaS
- Over 350 global retailer customers
- `https://www.fredhopper.com`

FRH services in a picture



Scaling FRH services

- Dedicated team of Cloud Engineers (Dev**Ops**)
- Manual scaling based on
  - scheduled promotions/campaigns
  - monitoring alerts
  - customer-signed Service Level Agreement (SLA)
  - specific deployment requirements
  - business logic

Complications of manual scaling

- Scaling process is NP-hard: many machine types, many ways to distribute objects over VMs
- Complex deployment requirements
- SLAs are written informally in natural-language
- Monitors not directly related to SLAs/KPIs
- Humans (DevOps) have essential domain knowledge, but are prone to errors

Proposed idea: train DevOps engineers by interactive simulation

Outline

Human-in-the loop Simulation (HITL)

Characteristics of HITL simulations:

- User-interactive
- Real-time responsive
- Not as fast-as-possible simulation
- Not necessarily precise
- Difficult to reproduce (non-reproducible)

Human-in-the loop Simulation (HITL)

Characteristics of HITL simulations:

- User-interactive
- Real-time responsive
- Not as fast-as-possible simulation
- Not necessarily precise
- Difficult to reproduce (non-reproducible)

Example: Flight Simulator

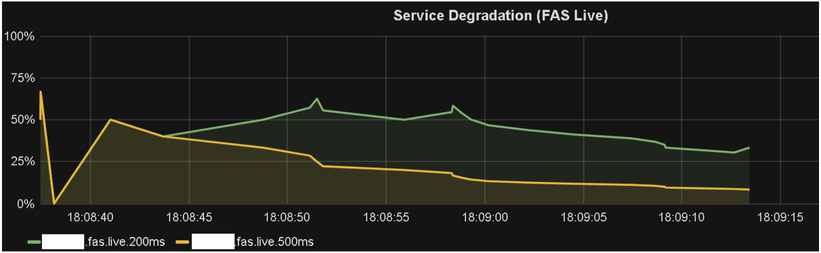
Components of our HITL-framework

SmartDeployer	synthesizing executable provisioning script for cloud actions (initialization, scale up/down)
SAGA	monitor SLA metrics, propose scaling suggestions
Grafana	visualize monitors
Logreplay	replay real-world log files
Haskell-ABS	backend for simulation with real-time support
Front-end	webpage for Ops team to select desired scaling

FRH services in an ABS model

- ABS code modeling the FRH services
- Modeled deployment requirements for SmartDeploy
- Serving the actual catalog requests is omitted
- The *logreplay* tool feeds requests and their processing time to the running ABS program through REST-API
- REST-API calls trigger `Cost`-annotated code to advance the simulation time.

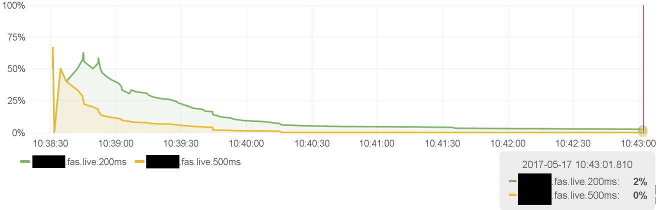
The GUI for human-interaction



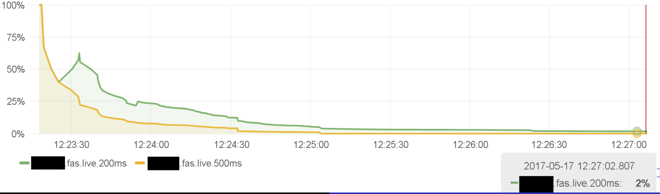
- TimeSpec {sec = 43, nsec = 642376972}: the monitor named *DegradationMonitor* recommends to Scale Up. Apply
- TimeSpec {sec = 38, nsec = 172067635}: the monitor named *DegradationMonitor* recommends to Scale Up. Apply
- TimeSpec {sec = 32, nsec = 736471691}: the monitor named *DegradationMonitor* recommends to Scale Up. Apply
- TimeSpec {sec = 27, nsec = 264342453}: the monitor named *DegradationMonitor* recommends to Scale Up. Apply
- TimeSpec {sec = 21, nsec = 830992806}: the monitor named *DegradationMonitor* recommends to Scale Down. Apply
- TimeSpec {sec = 16, nsec = 400021617}: the monitor named *DegradationMonitor* recommends to Scale Down. Apply
- TimeSpec {sec = 10, nsec = 928609813}: the monitor named *DegradationMonitor* recommends to Scale Down. Apply
- TimeSpec {sec = 5, nsec = 453975631}: the monitor named *DegradationMonitor* recommends to Scale Up. Apply

Outline

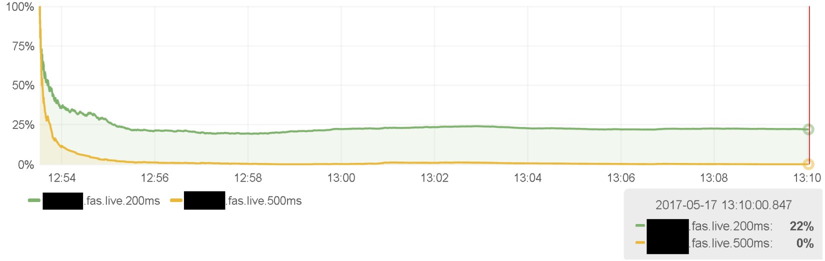
Versus symbolic time



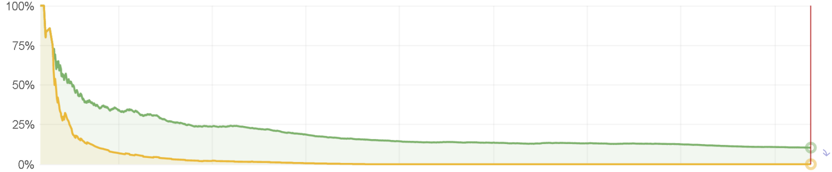
(a) Haskell simulation of the degradation when simulating the original log (Time elapsed: 4m 30s)



Exercise for FRH engineers



(c) No scaling - 200ms metric breaks SLA



Future work

Run HITL simulation side-by-side with production system, driving simulation with real-time data *from* production and feed back scaling suggestions *to* production.

Further increase automation. Challenge: resolving conflicting scaling suggestions from different monitors.

- Non-reproducible simulation for two reasons:
  1. Haskell runtime guarantees the re-activation of a "sleeping" thread no sooner than prescribed *but may be later*
  2. No notion of simultaneous method calls, because of no total ordering of symbolic time.
- Next step: parallel discrete-event simulation

Thank you!

Tool & Case-study code:

`http://github.com/abstools/habs-frh`

Info on the ABS language: `http://abs-models.org`