Reactive Programming for Data-center Management: First Experiences and Prospects

Nicolas Berthier   Éric Rutten   Noël De Palma

November 18-22, 2013

Synchron'13

Outline

- Context & Motivations
- Reactive Programming. . .
- . . . for *n*-tier Applications Management
- Validation
- Summary

Outline

- Context & Motivations
- Reactive Programming…
- … for *n*-tier Applications Management
- Validation
- Summary

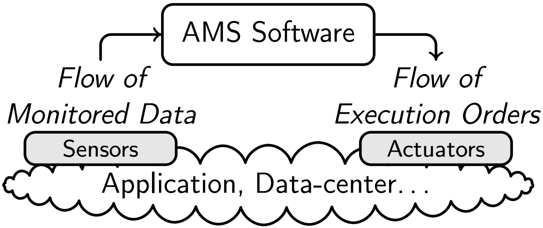
Autonomic Management Systems (AMSs)

Management Tasks

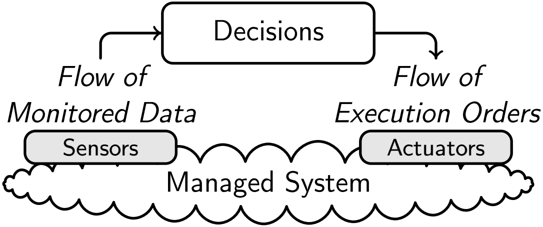
- Facilities: Energy, Cooling...
- Computation Infrastructure: Physical Machines...
- Application Management
  - Resource Allocation (Virtual Machines...), Configuration...
  - Example 3-tier Application:

$\rightarrow$ Apache $\rightarrow$ Tomcat $\rightarrow$ Mysql

Autonomic Management Systems (AMSs)



Autonomic Management Systems are Reactive Systems



Noticeable Properties

- Typical Feedback Loop
- Sensors & Actuators Spread Over the Managed System
- Most Often: Centralized Execution of the Software Taking Management Decisions
- Yet: Management Aspects Addressed Independently from Each Other
  - Sizing / Repairing / ...

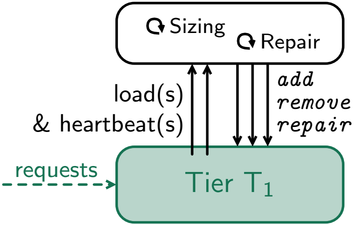
Motivating Example: The *Sizing-repair* Case

Sizing

- Over-load ⇝ *add*
- Under-load ⇝ *remove*

Repair

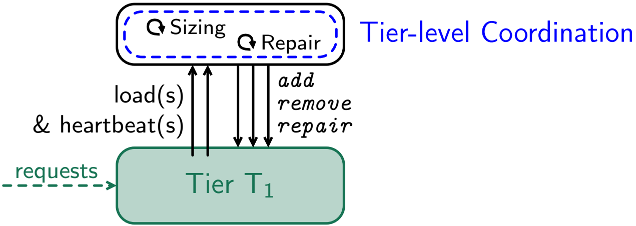
- Failure ⇝ *repair*



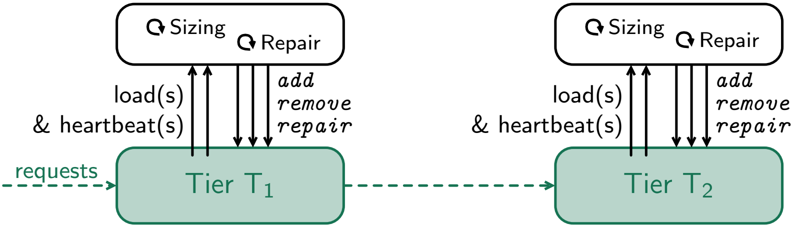
Motivating Example: The *Sizing-repair* Case

Tier-level Coordination

- Concurrency Management
  - Mutual Exclusions on "Accesses" to Shared (Actuators / Resources)
    - *e.g.*, `add`, `repair`
  - Conflicting Behaviors
    - *e.g.*, `remove`, `repair`



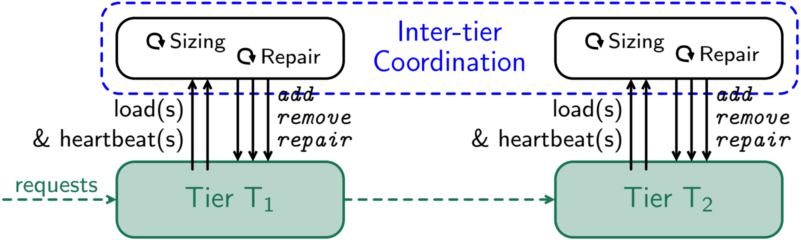
Motivating Example: The *Sizing-repair*+ Case



Motivating Example: The *Sizing-repair*+ Case

Inter-tier Coordination

- Remote Impacts of Local Events, *e.g.*,
  - → Failure in T_1 ⇝ *Transient* Under-load in T_2
  - → Failure in T_1 ⇝ *Transient* (Under-load, then Over-load) in T_2
  - ← Failure in T_2 ⇝ *Transient* Load Variation in T_1



Coordinating AMSs' Decisions

Observation

- Systems' Community: Realm of Object/Service/*-oriented Programming
  - Event-based: Handling One Event at a Time
  - Missing Global Knowledge

Coordinating AMSs' Decisions

Observation

- Systems' Community: Realm of Object/Service/*-oriented Programming
  - Event-based: Handling One Event at a Time
  - Missing Global Knowledge

Current Solutions...

- Heavy Use of Synchronization Primitives
  - Tedious, Error-prone
- Very *ad hoc*...
  - Hard to Generalize

Objective & Contributions

Objective

- Coordination-friendly Programming of AMSs

Proposal: Exploiting the *Reactive Nature* of AMSs

- New Design Methodology for AMSs
- Synchronous / Reactive Programming
- Discrete Controller Synthesis
- Prototype Implementation & Validation

Outline

- Context & Motivations
- Reactive Programming...
  - Synchronous Programming
  - Discrete Controller Synthesis
- ... for *n*-tier Applications Management
- Validation
- Summary

Synchronous Programming of Reactive Systems

Benefits

- High-level Programming Languages
  - *e.g.*, Automata
- Efficient Verification Techniques
  - Design-time Validation
- Means for Static Property Enforcement
  - Automatic Generation of *Minimally Restrictive Controllers*

$\not\rightsquigarrow$ Over-constrained System

Synchronous Programming of Reactive Systems

Benefits

- High-level Programming Languages
  - *e.g.*, Automata
- Efficient Verification Techniques
  - Design-time Validation
- Means for Static Property Enforcement
  - Automatic Generation of *Minimally Restrictive Controllers*

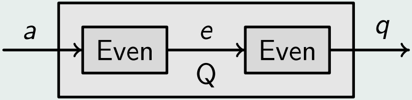
    $\not\leadsto$ Over-constrained System

Usual Design Technique

- Model-driven Programming
  - System Model
  - Expressing Decisions Based on this Knowledge

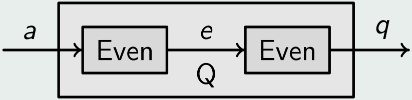
Example Synchronous Program

Data-flow Representation

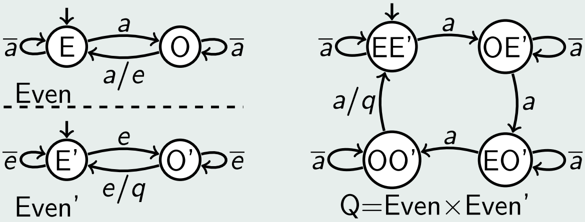


Example Synchronous Program

Data-flow Representation



As a Product of Communicating Boolean Mealy Automata



Executing a Synchronous Program

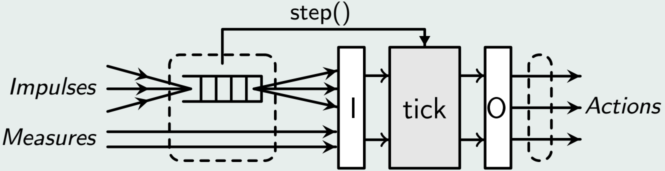
Typical Execution Pattern

- Inputs
  - Impulses (≈ Discrete Events)
  - Measures (≈ Always Relevant Data)
- Outputs
  - Actions

Executing a Synchronous Program

Typical Execution Pattern

- Inputs
  - Impulses (≈ Discrete Events)
  - Measures (≈ Always Relevant Data)
- Outputs
  - Actions
- "Using" the tick
  - Decide Relevant Instants (Not Necessarily Periodic!)
  - Build an Input Vector I
  - Execute the tick.step() Method
  - Interpret Output Vector O, Send Commands to the Actuators



Discrete Controller Synthesis Problem

Principle

- Behaviors $A_1 \cdots A_n$ (*e.g.*, Automata)
- Property Φ (*e.g.*, Mutual Exclusions, Avoiding Bad States)

$$A_1 \parallel \cdots \parallel A_n \qquad\qquad \Phi$$

Discrete Controller Synthesis Problem

Principle

- Behaviors $A_1 \cdots A_n$ (*e.g.*, Automata)
- Property Φ (*e.g.*, Mutual Exclusions, Avoiding Bad States)
- Provided $A_1 \cdots A_n$ are (made) Controllable ($\rightsquigarrow A_1{}' \cdots A_n{}'$)

$$A_1{}' \parallel \cdots \parallel A_n{}' \qquad \qquad \Phi$$

Discrete Controller Synthesis Problem

Principle

- Behaviors $A_1 \cdots A_n$ (*e.g.*, Automata)
- Property Φ (*e.g.*, Mutual Exclusions, Avoiding Bad States)
- Provided $A_1 \cdots A_n$ are (made) Controllable ($\rightsquigarrow A_1' \cdots A_n'$)

$\rightsquigarrow$ Controller C *s.t.*

$$A_1' \parallel \cdots \parallel A_n' \parallel C \models \Phi$$

Discrete Controller Synthesis Problem

Principle

- Behaviors $A_1 \cdots A_n$ (*e.g.*, Automata)
- Property Φ (*e.g.*, Mutual Exclusions, Avoiding Bad States)
- Provided $A_1 \cdots A_n$ are (made) Controllable ($\leadsto A_1{}' \cdots A_n{}'$)

$\leadsto$ Controller C *s.t.*

$$A_1{}' \parallel \cdots \parallel A_n{}' \parallel C \models \Phi$$

Available Tool

- In the Synchronous Languages Domain: Sigali (until now...)

Outline

- Context & Motivations
- Reactive Programming...
- ... for *n*-tier Applications Management
- Validation
- Summary

Principles

Driving the Managed System with *Resource Drivers*

- Operating Code Senses & Acts Upon the Resources
  - Example of Resources:
    - Tier of an Application, Sets of Virtual Machines…
  - Pieces of Sequential Code (*e.g.*, `add`)
- Associated Behavioral Model Represents Operating States
  - *e.g.*, "Stopped", "Running", "Adding a node"…

Principles

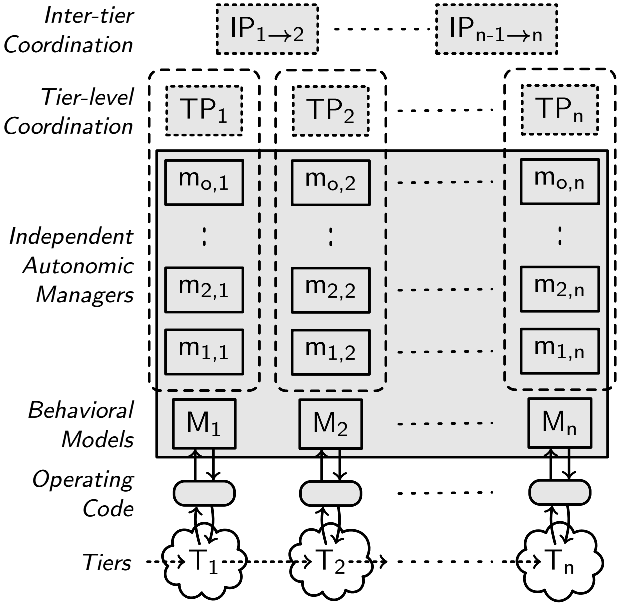
Driving the Managed System with *Resource Drivers*

- Operating Code Senses & Acts Upon the Resources
  - Example of Resources:
    - Tier of an Application, Sets of Virtual Machines…
  - Pieces of Sequential Code (*e.g.*, `add`)
- Associated Behavioral Model Represents Operating States
  - *e.g.*, "Stopped", "Running", "Adding a node"…

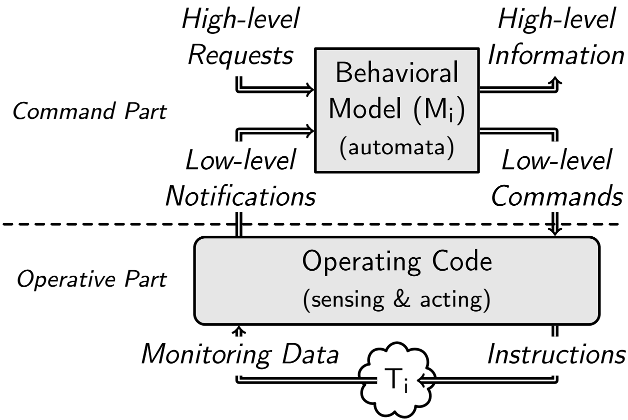
Autonomic Management Strategies & Coordination Policies

- Expressed in a *Declarative Way*
  - *e.g.*, Automata, Systems of Equations
  - Possibly Involving Costs
- Coordination Policies Enforced by using Discrete Controller Synthesis

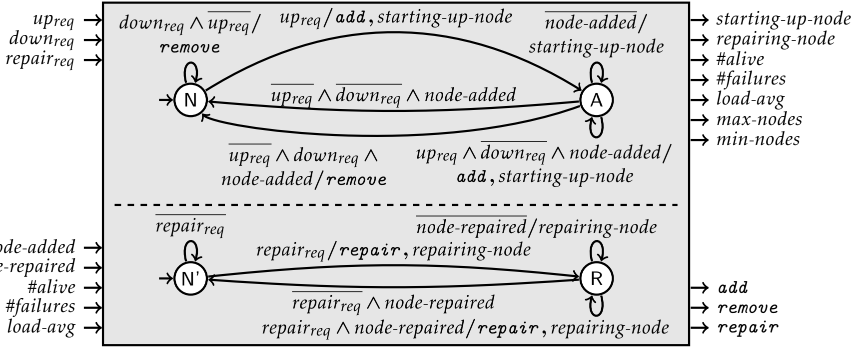
Encoding the AMS Logic for an n-tier Application



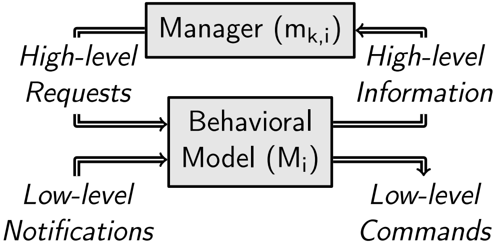
Structure of a Resource Driver



Behavioral Model for a Tier



Expressing Autonomic Management Strategies



Example Autonomic Management Strategies

Repair Strategy

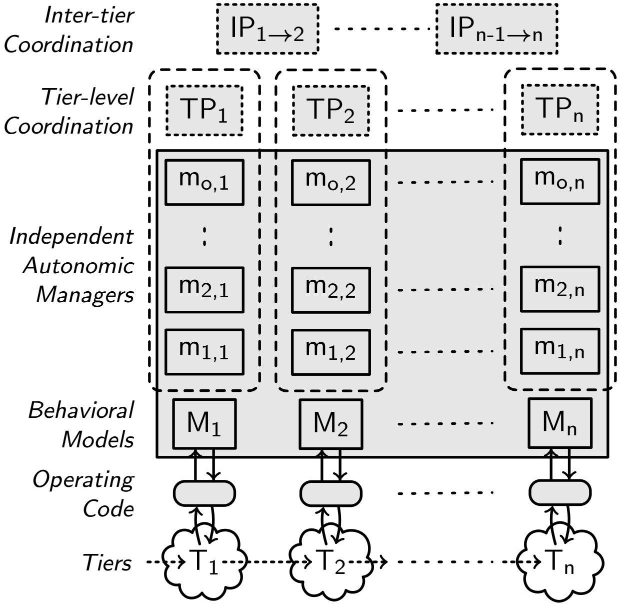
$$repair = \left(\#failures = 1 \land \overline{repairing\text{-}node}\right) \lor \#failures > 1$$

Sizing Strategy

$$\begin{aligned} up &= \overline{max\text{-}nodes} \land \text{upper-threshold}(load\text{-}avg, \#alive) \\ down &= \overline{min\text{-}nodes} \land \text{lower-threshold}(load\text{-}avg, \#alive) \end{aligned}$$

- Could be Less Trivial Synchronous Programs...

Specifying Coordination Policies



Example Tier-level Coordination Property

Allow Repair Operation ($\mathtt{repair}$) While Up-sizing ($\mathit{starting\text{-}up\text{-}node}$),
or Up-sizing Operation ($\mathtt{add}$) While Repairing ($\mathit{repairing\text{-}node}$)
In Case of Multiple "Recent" Failures Only ($\#\mathit{failures} > 1$)

Implementation

- Enforce both predicates:

$$(\mathit{starting\text{-}up\text{-}node} \wedge \mathtt{repair}) \Rightarrow \#\mathit{failures} > 1$$
$$(\mathit{repairing\text{-}node} \wedge \mathtt{add}) \Rightarrow \#\mathit{failures} > 1$$

Example Inter-Tier Coordination Property

For Each Tier $\mathsf{T_i}$,
Avoid Down-sizing Operations ($\mathtt{remove}_\mathsf{i}$)
if any of the "Preceding" Tiers $\mathsf{T_j}$ both:

- "Recently" Underwent at Least One Failure ($\#failures_j > 0$)
- Is Currently Repairing One Node ($repairing\text{-}node_j$)

Implementation

Enforce the following predicate, $\forall i \in [2, n]$:

$$\left(\bigvee_{j=1}^{i-1} (repairing\text{-}node_j \vee \#failures_j > 0)\right) \Rightarrow \overline{\mathtt{remove}_\mathsf{i}}$$

Eliciting Control Means

Filtering High-level Requests

$$\begin{aligned} repair_{req} &= ok_{repair} \wedge repair \\ up_{req} &= ok_{sizing} \wedge up \\ down_{req} &= ok_{sizing} \wedge down \end{aligned}$$

- High-level Requests: $\{ repair, up, down \}$
- Controllables: $\{ ok_{repair}, ok_{sizing} \}$
- Inputs of the Behavioral Model: $\{ repair_{req}, up_{req}, down_{req} \}$

Outline

- Context & Motivations
- Reactive Programming...
- ... for *n*-tier Applications Management
- Validation
  - Prototype Implementation
  - Validation
- Summary

Implementation Choices

Implementing the tick

- Using BZR
  - Encapsulating Sigali

Distributed Execution Platform: A^3

- Message-oriented Middleware (Agent-based)
- Fault-tolerant
- Atomic Executions
- Dynamic Deployments
  - Allows Adding & Removing Nodes (*e.g.*, Virtual Machines)

Validation

Experimental Setup

Tier	Repair	Sizing	available VMs
T_{apache}	✓	✗	1
T_{tomcat}	✓	✓	3
$T_{mysql\text{-}proxy}$	✓	✗	1
T_{mysql}	✓	✓	3

- Synthetic Workload (based on Rubis' Auction Site)
- 9 VMWare Virtual Machines
- Workload Injection from a Separate Physical Machine
- Collect Exponentially-Weighted-Moving-Average (EWMA) of CPU Utilization Percentages

Validation

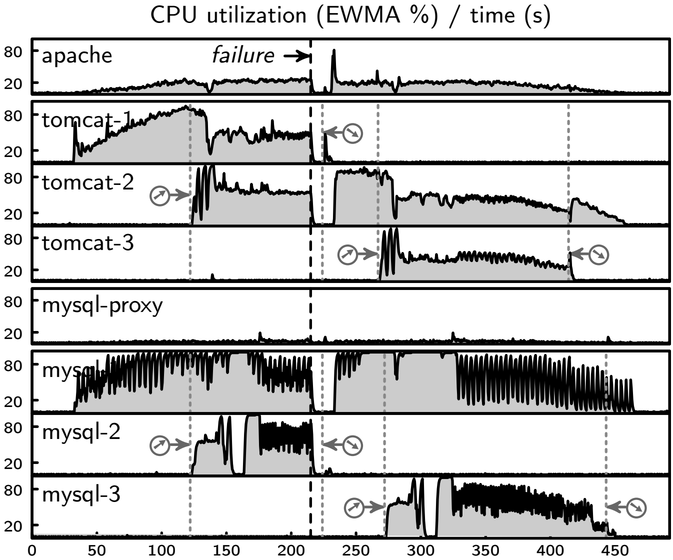
Experimental Setup

Tier	Repair	Sizing	available VMs
T_{apache}	✓	✗	1
T_{tomcat}	✓	✓	3
$T_{mysql\text{-}proxy}$	✓	✗	1
T_{mysql}	✓	✓	3

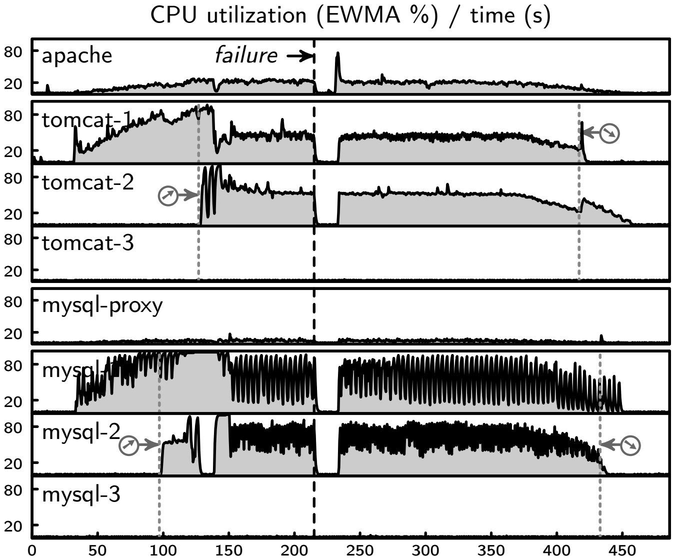
- Synthetic Workload (based on Rubis' Auction Site)
- 9 VMWare Virtual Machines
- Workload Injection from a Separate Physical Machine
- Collect Exponentially-Weighted-Moving-Average (EWMA) of CPU Utilization Percentages

Comparing Non-coordinated & Coordinated Executions

Non-coordinated Execution



Coordinated Execution



Outline

- Context & Motivations
- Reactive Programming...
- ... for *n*-tier Applications Management
- Validation
- Summary

Summary of the 1st Part

Objective

- Coordination-friendly Programming of AMSs

Proposal: Exploiting the *Reactive Nature* of AMSs

- New Design Methodology for AMSs
- Synchronous / Reactive Programming
- Discrete Controller Synthesis
- Prototype Implementation & Validation

Future Works

Quantitative Aspects

- Extending Behavioral Models with Relevant Quantitative Data
  - Enforce Quantitative Properties
    - *cf.* Next Part...

Distributed Execution of the AMS / Controller

- Centralized Design (Scalability Issue)

  $\leadsto$ Synchronous Languages Implementation Problem
- Decentralized Design (Modeling Issue)

  $\leadsto$ Decentralized Control Problem

Thanks!

Questions?

Outline

- Context & Motivations
- Reactive Programming...
  - Synchronous Programming
  - Discrete Controller Synthesis
- ... for *n*-tier Applications Management
- Validation
  - Prototype Implementation
  - Validation
- Summary

Outline

- Architecture of the Prototype

Architecture of the Prototype