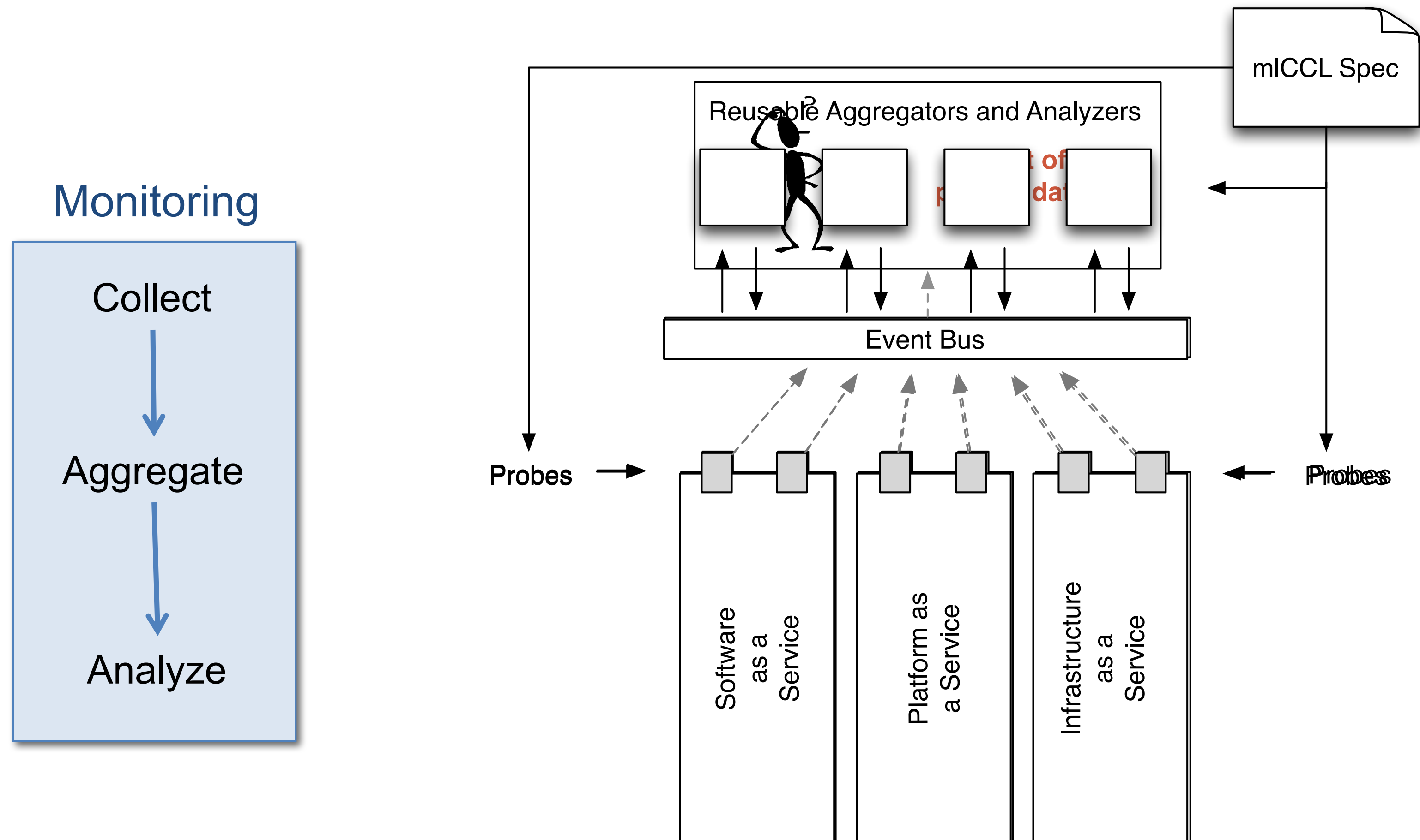
A FINE-GRAINED AUTONOMIC MANAGEMENT SOLUTION FOR MULTI-LAYERED SYSTEMS

Luciano Baresi and Sam Guinea
[luciano.baresi|sam.guinea]@polimi.it

Multi-level Monitoring



Aggregators and Analyzers collaborate to produce the knowledge we need!

Two main contributions

- Multi-layer Collection and Constraint Language (mlCCL)
  - Declarative language for defining
    - The runtime data we want to collect from the various layers
    - How to aggregate the data to build higher-level knowledge
    - How to analyze the data to identify undesired behavior

- ECoWare FrameWork
  - Event Correlation middleWare
  - Supports mlCCL specifications
  - Provides advanced data aggregation and analysis

mICCL – Data Collection

- Data described in terms of Service Data Objects (SDOs)
  - Language-agnostic
  - Set of named properties
    - Single- or multi-valued (array)
    - Primitive (number, string, boolean) or complex (SDO)

- Data Collection is about configuring probes

- Two kinds of Data Collection:
  - Message Collection
  - Indicator Collection

Message Collection

- Request or response messages exchanged during service invocation

```
locName = [before | after] endpoint;
aliasName = collect (locName);
```

```
locBefore = before(getUserWeather, TwitterWeather, Orchestrator);
locAfter = after(getUserWeather, TwitterWeather, Orchestrator);

request = collect(locBefore);
response = collect(locAfter);
```

Indicator Collection

- Periodic information about a service -- not triggered
  - 4 Key Performance Indicators (KPIs) -- for Software Services
  - 15 Resource Indicators (RIs) -- for Infrastructure Services

```
aliasName = collect (indicatorName, serviceID, outputRate, pastWindow, property);
```

  - **serviceID** is either an endpoint or the unique ID of a VM
  - **property** is a mlCCL analysis expression for identifying how many SDOs, in a pastWindow, satisfy a a given property

```
orchestratorEndpoint = (getUserWeather, TwitterWeather, Orchestrator);
avgrt = collect(avgRT, orchestratorEndpoint, 5 minutes, 1 hour, null);
rate = collect(rate, orchestratorEndpoint, 5 minutes, 1 hour, null);

vm = (VM101);
cpu = collect(cpuSystem, vm, 5 minutes, null, null);
```

Data Aggregation

- Aggregate multiple SDOs into one
  - SDOs can be collected at different times…
  - When to aggregate? What to aggregate?
  - Primary vs. Secondary Events

```
aliasName = aggregate(primary, eventList);
```

  - To aggregate a "window" of events, attach window(interval) to a secondary event

```
avgrt_rate = aggregate(avgrt, rate.window(1 hour));
```

Data Analysis 1/2

- Predicate over the contents of our SDOs
  - append get(propertyName) to alias
  - Further manipulate the data
    - Numbers: absolute value, square root
    - Strings: substring, length, replace
    - Arrays: legnth, i-th value, subset of values that satisfy a property
    - Array of Numbers: sum, avg, min, max

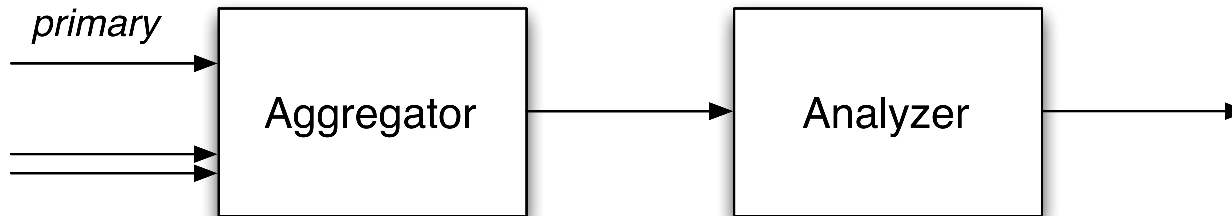
⟨prop⟩ ::= ¬⟨prop⟩ | ⟨prop⟩&&⟨prop⟩ | ⟨prop⟩||⟨prop⟩ | ⟨array⟩ . ⟨quant⟩ (⟨prop⟩) |
    ⟨term⟩⟨rop⟩⟨term⟩
⟨term⟩ ::= ⟨prim⟩ | ⟨term⟩⟨aop⟩⟨term⟩ | ⟨const⟩
⟨rop⟩ ::=<|≤|==|≥|>
⟨quant⟩::= forall | exists
⟨aop⟩ ::=+|−| × | ÷ |%

Data Analysis 2/2

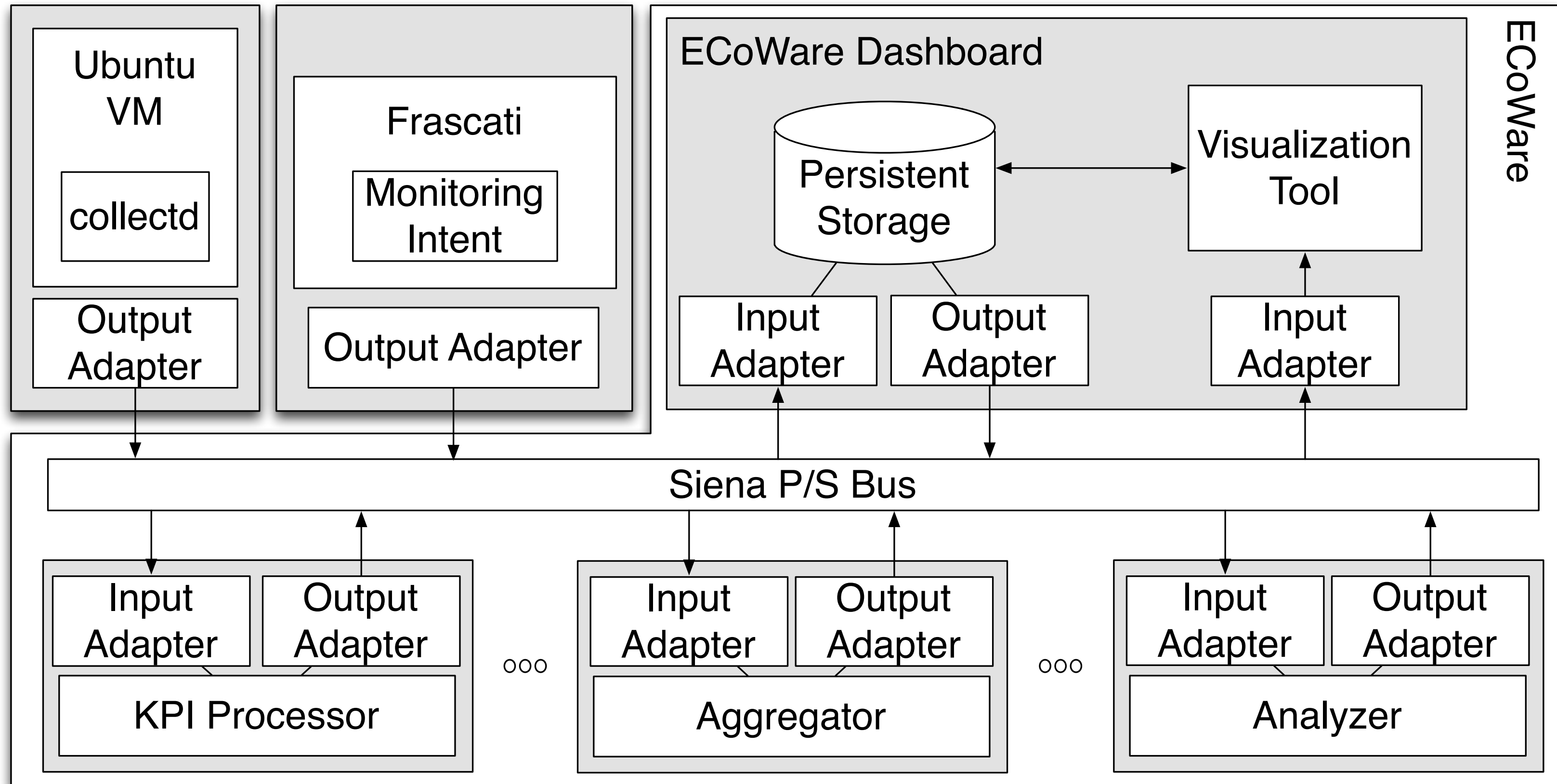
- When should I perform the data analysis?
  - We use primary again!

```
aliasName = evaluate(primary, expression);
```

```
violation = evaluate(avgrt, avgrt.get(value) > 5500);
```

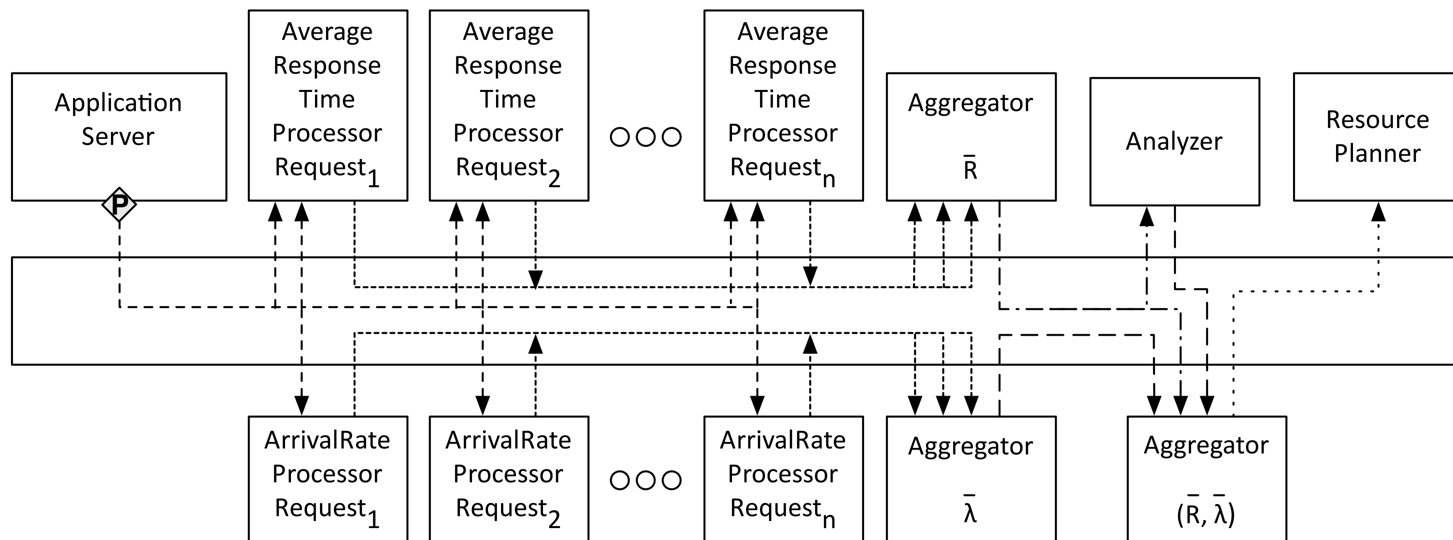


ECoWare



What about Monitoring and Analysis?

- Probes that tell us
  - Servlet execution time
  - Amount of time spent by a servlet communicating with the DB
  - Number of CPU / cache / RAM
- AvgRT monitoring + ArrivalRate Aggregation

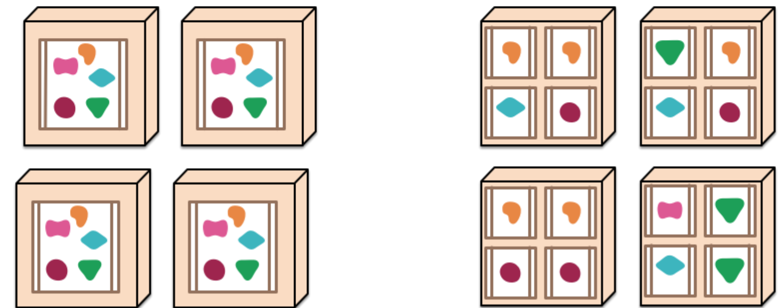


ADVANCED SELF-ADAPTATION OF CLOUD APPLICATIONS WITH CONTAINERIZATION AND CONTROL THEORY

Giovanni Quattrocchi, Alberto Leva,
Luciano Baresi, Sam Guinea

Cloud apps and Microservices

- Web application or service deployed and executed on a cloud (e.g., AWS)
- The cloud is a black-box (no access to the hypervisor)
- Many lightweight applications / microservices instead of a single monolithic application
- Shared infrastructure



Monolithic Architecture

Micro-service Architecture

Layers of cloud adaptation

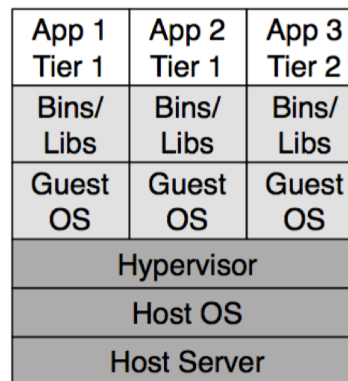
- Infrastructure adaptation: changing at runtime the number of computing resources (e.g., VMs) allocated for an app
- Platform adaptation: dynamically (re)configuring the middleware software that enable the execution of the application (e.g., number of workers of an application server)
- Feature adaptation / service degradation: remove/add heavy optional features (e.g., recommendation system) when the system is saturated/stable

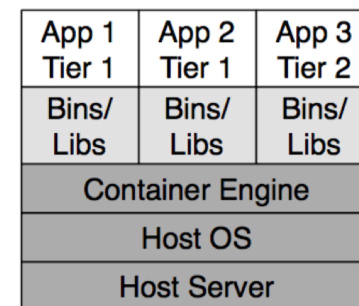
State of the art

- No self-adaptive system exploits all the aforementioned kinds of adaptation
- It’s not easy to create a generic system that takes into account different technologies and standards (in particular at the middleware and application layers)
- Current state of the art focus on infrastructure adaptation by means of Virtual Machines management

Containers

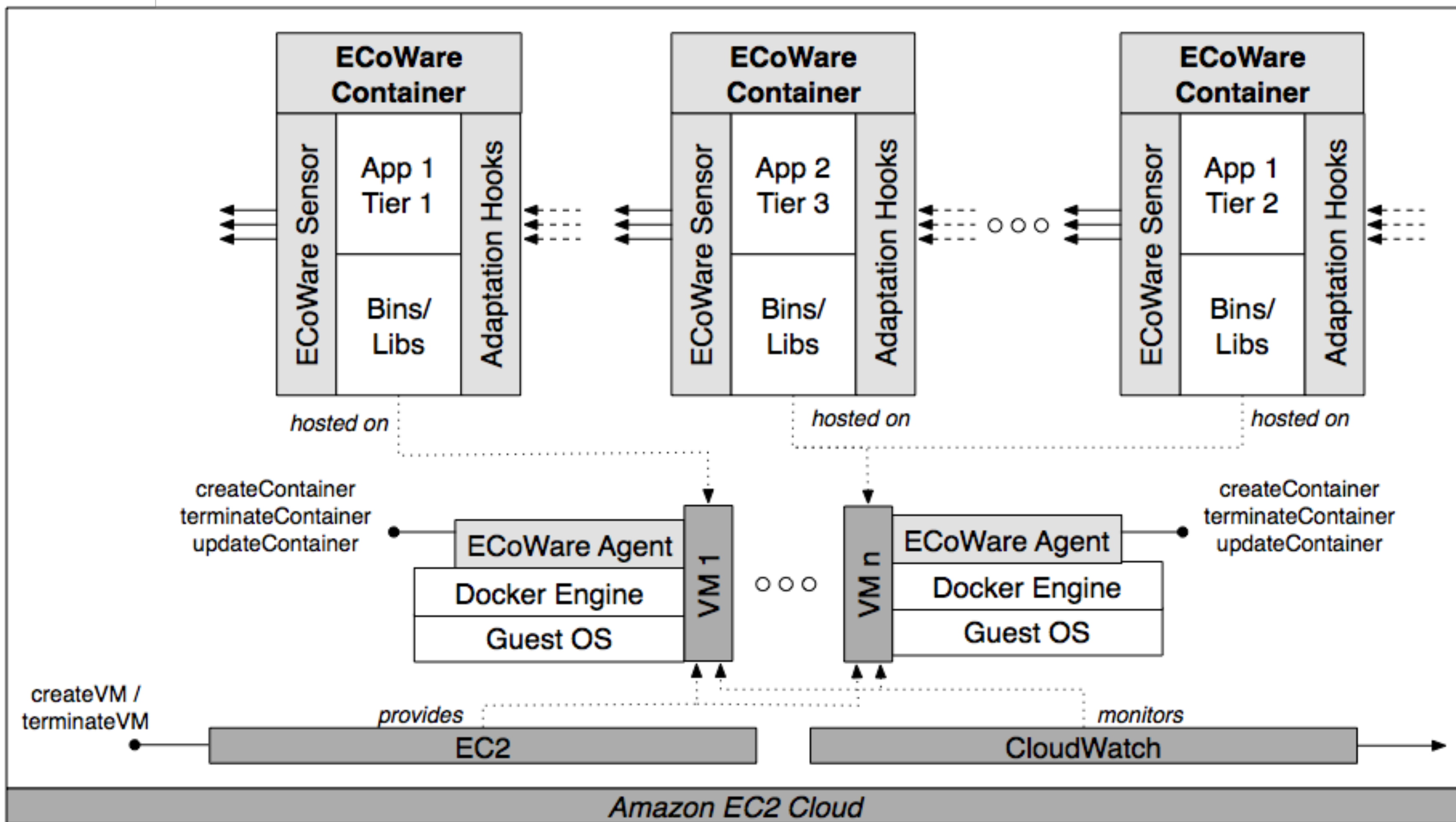
- OS layer virtualization technique
- Lightweight and faster than Virtual Machine
- One container = One process
- Each container runs sandboxed (isolated, no contention between processes)
- Sys admins can assign the adequate number of resources to each container (cpu, memory)
- Built-in the linux kernel (little overhead)
- Docker





Containers in ECoWare

- Containers enable adaptation at the Operating System layer
- Instant reaction to workload changes by dynamically starting, terminating or updating (vertical scaling) containers
- We deploy many containers inside each VM
- Finer granularity than working only with VMs
- Resources allocated to a middleware (e.g., MySQL, JBoss AS) can change during execution



ECoWare Container
ECoWare Sensor
App 1
Tier 1
Bins/
Libs
Adaptation Hooks
ECoWare Container
ECoWare Sensor
App 2
Tier 3
Bins/
Libs
Adaptation Hooks
ECoWare Container
ECoWare Sensor
App 1
Tier 2
Bins/
Libs
Adaptation Hooks
hosted on
hosted on
hosted on
createContainer
terminateContainer
updateContainer
ECoWare Agent
Docker Engine
Guest OS
VM 1
VM n
ECoWare Agent
Docker Engine
Guest OS
createContainer
terminateContainer
updateContainer
createVM /
terminateVM
provides
EC2
monitors
CloudWatch
Amazon EC2 Cloud
interface
monitoring data
incoming event

Planner

- Control-theoretical design, discrete-time
- Each application tier is endowed with a local controller devoted to maintaining a desired response time
- The controller computes the resources (i.e., CPU cores and memory) that need to be made available to that tier
- Five generic actions:
  - Container actions (create, update, terminate)
  - VM actions (create, terminate)

Control-theoretical design (I)

$$f\left(\frac{c(k)}{r(k)}\right) = c_1 + \frac{c_2}{1 + c_3 \frac{c(k)}{r(k)}}$$

- Grey-box approach
- The characteristic function is monotonically decreasing towards a possible lower horizontal asymptote (finite parallelism degree)
- Not linear, depends on c (core) and r (workload); c1,2,3 obtained through profiling
- Invertible in the signal range of interest

From resource allocation to actions

- Each controller emits the requested resource allocation for a tier
- These data are then passed to an ILP solver to be translated into actions (create VM, create container, …)
- The ILP formulation is a variation of the 2-dimensional bin packing problem: the bins are the VMs and we need to pack containers inside them
- Each of the five actions has a weigth w, such that containers actions are preferred over VM actions

Adaptation Hooks

- Scripts declared inside the Applications Description file and mounted into containers when launched
- Separated from the planner that we keep technology agnostic
- When some events happens we invoke the specific hook that manages the specific event
- Currently we support
  - on_node_scale: executed when the resources allocated to a container change
  - on_dependency_scale: executed when a dependent tier perform adaptation actions