ABS: Modeling and analysis with resource-sensitive actors

Silvia Lizeth Tapia Tarifa

University of Oslo, Norway
sltarifa@ifi.uio.no

Oslo, 13 November 2023

UNIVERSITY OF OSLO

SIRIUS Center for Scalable Data Access in the Oil and Gas Domain

sfi Centre for Research-based Innovation
The Research Council of Norway

http://www.sirius-labs.no

Modeling and Analysis

Main idea:

- Describe a system using a language.
- Describe the properties or analysis of the system that one wants to do.
- Use a systematic method (with tool support) to check the properties or do some other kind of analysis (e.g., correctness, reachability analysis, time-related analysis, resource analysis, what-if scenarios, optimization, planning, etc.).

Modeling and Analysis

Main idea:

- Describe a system using a language.
- Describe the properties or analysis of the system that one wants to do.
- Use a systematic method (with tool support) to check the properties or do some other kind of analysis (e.g., correctness, reachability analysis, time-related analysis, resource analysis, what-if scenarios, optimization, planning, etc.).

Main topic of the day:

- Modeling with an actor-based language.
- Elasticity/scalability of the deployment of a system: performance vs. cost
- Resource-usage predictions using simulations.

Plan for Today

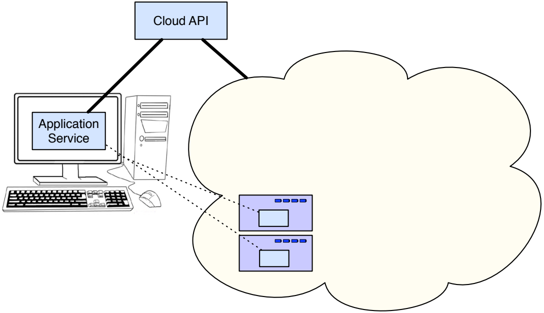
- **Question:** Can we use models to predict resource usage for applications running on the cloud?

Plan for Today

- **Question:** Can we use models to predict resource usage for applications running on the cloud?

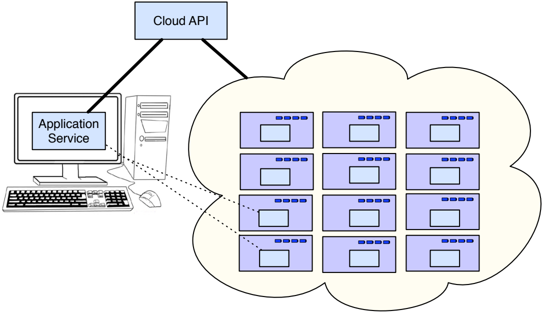
- Starting point: actors — a computation model which decouples execution from synchronization
- ABS: modeling language which adds resource-sensitive behavior to actors
- We look at how to enrich this model to make simulation of models of cloud-deployed services
- ABS has been used to model Hadoop clusters, industrial cloud applications, Kubernetes clusters, ...

We want to make effective use of cloud computing to meet service requirements



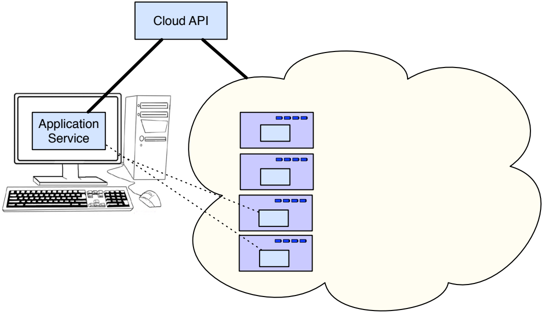
- **Virtualization** makes elastic amounts of resources available to application-level services
- **Metered resources**: Resources on the Cloud are pay-on-demand

We want to make effective use of cloud computing to meet service requirements



- **Virtualization** makes elastic amounts of resources available to application-level services
- **Metered resources**: Resources on the Cloud are pay-on-demand

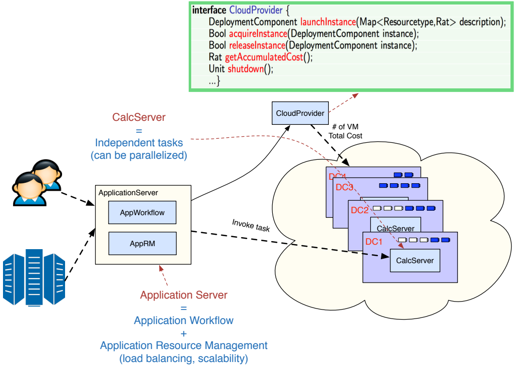
We want to make effective use of cloud computing to meet service requirements



- **Virtualization** makes elastic amounts of resources available to application-level services
- **Metered resources**: Resources on the Cloud are pay-on-demand

Resource-aware design

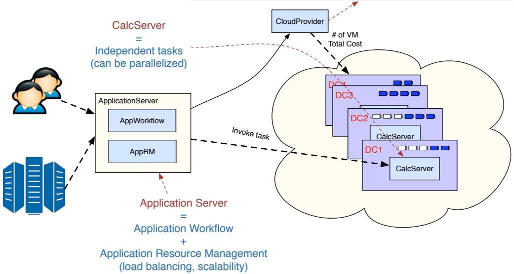
- **Goal:** Build software that can dynamically modify its own deployment to improve performance and/or reduce cost



Resource-aware design

- **Goal:** Build software that can dynamically modify its own deployment to improve performance and/or reduce cost

```
interface CloudProvider {
    DeploymentComponent launchInstance(Map<Resourcetype,Rat> description);
    Bool acquireInstance(DeploymentComponent instance);
    Bool releaseInstance(DeploymentComponent instance);
    Rat getAccumulatedCost();
    Unit shutdown();
    ...}
```



Discovering **bad design** after deployment on the Cloud can be a very costly (wasting both time and money!)

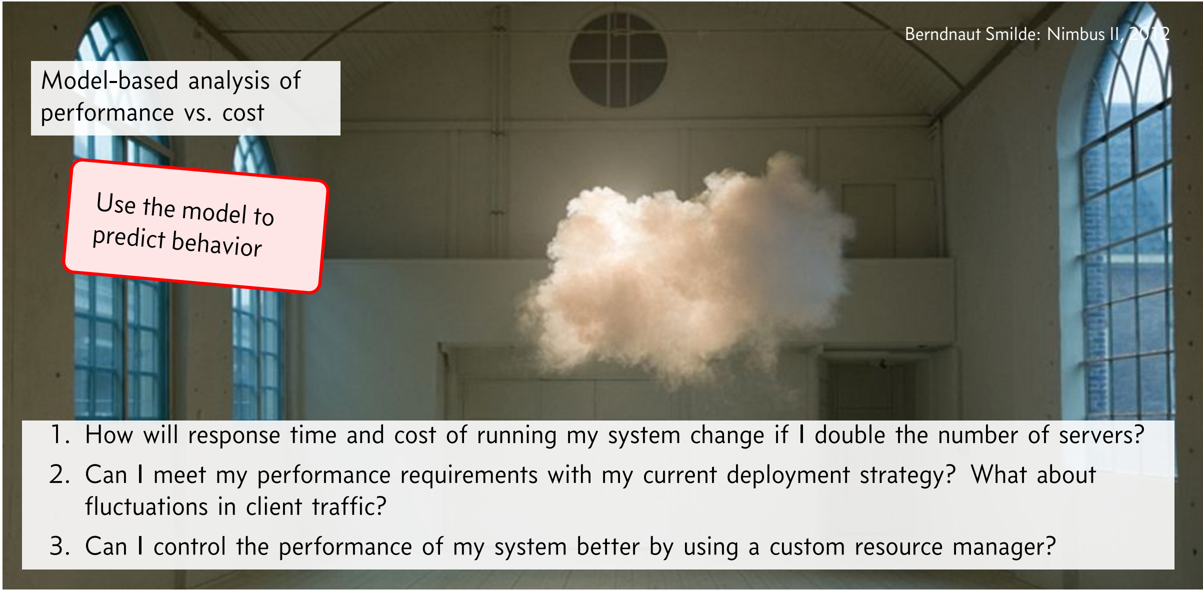
What kind of questions can we answer using models?

Berndnaut Smilde: Nimbus II, 2012

What kind of questions can we answer using models?

What kind of questions can we answer using models?

What kind of questions can we answer using models?



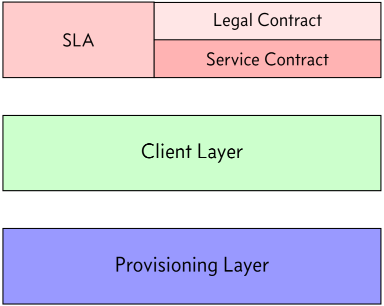
Berndnaut Smilde: Nimbus II, 2012

Model-based analysis of performance vs. cost

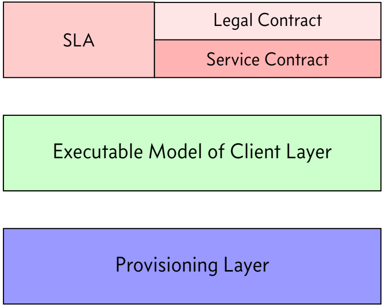
Use the model to predict behavior

1. How will response time and cost of running my system change if I double the number of servers?
2. Can I meet my performance requirements with my current deployment strategy? What about fluctuations in client traffic?
3. Can I control the performance of my system better by using a custom resource manager?

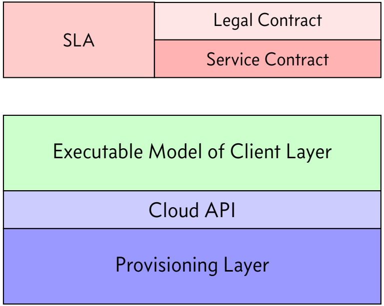
Conceptual Parts of a Deployed Cloud Service



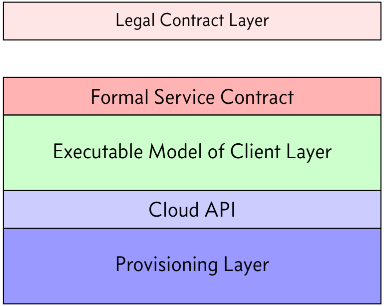
Conceptual Parts of a Deployed Cloud Service



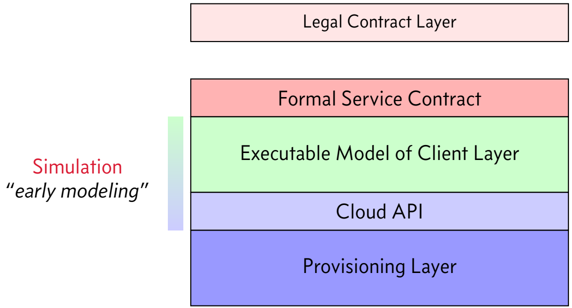
Conceptual Parts of a Deployed Cloud Service



Conceptual Parts of a Deployed Cloud Service



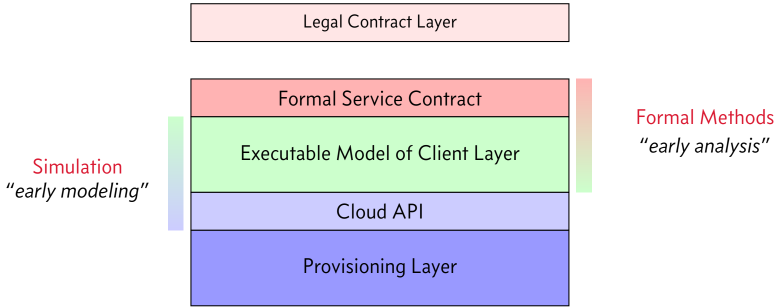
Conceptual Parts of a Deployed Cloud Service



Combine techniques based on abstract executable models

- Modeling using Abstract Behavioral Specifications (ABS)

Conceptual Parts of a Deployed Cloud Service



Combine techniques based on abstract executable models

- Modeling using Abstract Behavioral Specifications (ABS)
- Formal methods: Performance Analysis, Cost Analysis

Actors and ABS

Actors in Computer Science

Actors: Software for a concurrent world

- **Actor model** [Hewitt, Baker 77]: *actors* are the universal primitive of concurrent computation.

Actors in Computer Science

Actors: Software for a concurrent world

- **Actor model** [Hewitt, Baker 77]: *actors* are the universal primitive of concurrent computation.
- When receiving a message, an actor can react by making local decisions, creating actors, sending messages, and deciding how to respond to the next message.

Actors in Computer Science

Actors: Software for a concurrent world

- **Actor model** [Hewitt, Baker 77]: *actors* are the universal primitive of concurrent computation.
- When receiving a message, an actor can react by making local decisions, creating actors, sending messages, and deciding how to respond to the next message.
- Actors may modify their own private state, but can only affect each other indirectly through messaging.

Actors in Computer Science

Actors: Software for a concurrent world

- **Actor model** [Hewitt, Baker 77]: *actors* are the universal primitive of concurrent computation.
- When receiving a message, an actor can react by making local decisions, creating actors, sending messages, and deciding how to respond to the next message.
- Actors may modify their own private state, but can only affect each other indirectly through messaging.
- Messages need not arrive in the order in which they are sent to the actor.

Actors in Computer Science

Actors: ~~Software for~~ *Models of* a concurrent world

- **Actor model** [Hewitt, Baker 77]: *actors* are the universal primitive of concurrent computation.
- When receiving a message, an actor can react by making local decisions, creating actors, sending messages, and deciding how to respond to the next message.
- Actors may modify their own private state, but can only affect each other indirectly through messaging.
- Messages need not arrive in the order in which they are sent to the actor.

ABS: Abstract Behavioral Specifications

ABS explores the asynchronous features of loosely coupled actors in the setting of formal models

- **State-of-art programming language concepts**
  - ADTs + functions + objects + interfaces
  - Formal semantics, type-safety, data race-freeness by design

ABS: Abstract Behavioral Specifications

ABS explores the asynchronous features of loosely coupled actors in the setting of formal models

- **State-of-art programming language concepts**
  - ADTs + functions + objects + interfaces
  - Formal semantics, type-safety, data race-freeness by design
- **Layered concurrency model**

  Upper tier: asynchronous, no shared state, actor-based
  Lower tier: shared state, cooperative scheduling

ABS: Abstract Behavioral Specifications

ABS explores the asynchronous features of loosely coupled actors in the setting of formal models

- **State-of-art programming language concepts**
  - ADTs + functions + objects + interfaces
  - Formal semantics, type-safety, data race-freeness by design
- **Layered concurrency model**

  Upper tier: asynchronous, no shared state, actor-based

  Lower tier: shared state, cooperative scheduling
- **Modeling of variability/resources** with first-class language support
  - Variability: feature models, delta-oriented programming
  - Deployment: abstract resources and cost annotations

ABS: Abstract Behavioral Specifications

ABS explores the asynchronous features of loosely coupled actors in the setting of formal models

- **State-of-art programming language concepts**
  - ADTs + functions + objects + interfaces
  - Formal semantics, type-safety, data race-freeness by design
- **Layered concurrency model**

  Upper tier: asynchronous, no shared state, actor-based

  Lower tier: shared state, cooperative scheduling
- **Modeling of variability/resources** with first-class language support
  - Variability: feature models, delta-oriented programming
  - Deployment: abstract resources and cost annotations

ABS is designed with analysis/code generation tools in mind

ABS: Abstract Behavioral Specification

ABS is a language for executable designs

- Models follow the execution flow of OO programs, but abstract from implementation details using ADTs

ABS: Abstract Behavioral Specification

ABS is a language for executable designs

- Models follow the execution flow of OO programs,
  but abstract from implementation details using ADTs
- Imperative layer: concurrent objects communicating by asynchronous method calls
- Functional layer: user-defined (parametric) types and functions, pattern matching

ABS: Abstract Behavioral Specification

ABS is a language for executable designs

- Models follow the execution flow of OO programs,
  but abstract from implementation details using ADTs
- Imperative layer: concurrent objects communicating by asynchronous method calls
- Functional layer: user-defined (parametric) types and functions, pattern matching
- Java-like syntax: intuitive to the programmer

ABS: Abstract Behavioral Specification

ABS is a language for executable designs

- Models follow the execution flow of OO programs,
  but abstract from implementation details using ADTs
- Imperative layer: concurrent objects communicating by asynchronous method calls
- Functional layer: user-defined (parametric) types and functions, pattern matching
- Java-like syntax: intuitive to the programmer

ABS is a formal, tool-supported modeling language

- Formal language: operational semantics, type system
- Backends: compiles into Java COGs, Erlang Actors
- Maude interpreter formalizes semantics as a rewrite system

Asynchronous Method Calls

ABS decouples communication and synchronization

- ABS supports **asynchronous method calls**, using futures.

Asynchronous Method Calls

ABS decouples communication and synchronization

- ABS supports **asynchronous method calls**, using futures.

Futures are first-class values

```
f1=x!m(); f2=y!p(f1);
```

Asynchronous Method Calls

ABS decouples communication and synchronization

- ABS supports **asynchronous method calls**, using futures.

Futures are first-class values

f1=x!m(); f2=y!p(f1);

Flexible synchronization: *blocking* or *suspending* activities

- Blocking the object: **get-operation** on a future: v=f1.**get**;
- Suspending the process: **polling** a future: **await** f1?
- Polling as part of a (per object) guarded command: **await** b & f1? & f2?

Asynchronous Method Calls

ABS decouples communication and synchronization

- ABS supports **asynchronous method calls**, using futures.

Futures are first-class values

f1=x!m(); f2=y!p(f1);

Flexible synchronization: *blocking* or *suspending* activities

- Blocking the object: **get-operation** on a future: v=f1.**get**;
- Suspending the process: **polling** a future: **await** f1?
- Polling as part of a (per object) guarded command: **await** b & f1? & f2?

Cooperative scheduling of method activations

- Easy to **combine active** and **reactive behavior**

Cooperative Scheduling in ABS

Important Consequences

- Task suspension is a **syntactically explicit** decision of the modeller
- No preemptive scheduling $\Rightarrow$ **no data races**
  - *Scheduling cannot arbitrarily interfere with the computation*
- Non-deterministic scheduling otherwise
  - User-defined specification of schedulers via annotations
  - Analysis results for ABS programs are valid for any scheduler

Cooperative Scheduling in ABS

Important Consequences

- Task suspension is a **syntactically explicit** decision of the modeller
- No preemptive scheduling $\Rightarrow$ **no data races**
  - *Scheduling cannot arbitrarily interfere with the computation*
- Non-deterministic scheduling otherwise
  - User-defined specification of schedulers via annotations
  - Analysis results for ABS programs are valid for any scheduler

Reading Futures

- f.**get** — blocks execution until result is available, then reads future f
- Deadlocks possible (use static analyzer for detection)
- Programming idiom: use **await** f? to prevent blocking (safe access)

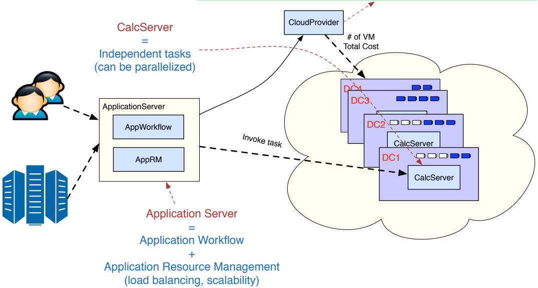
Fut<T> f = o!m(e);...; **await** f?; ...; r = f.**get**;

Modeling and Analysis of Deployment on the Cloud Using Real Time ABS

Services deployed on the cloud: Predicting behavior from models

- **Resource-aware design**: Build software that can dynamically modify its own deployment to improve performance and/or reduce cost

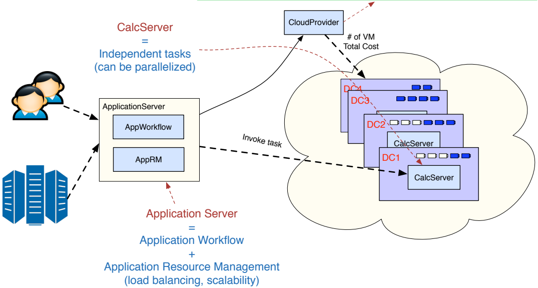
```
interface CloudProvider {
    DeploymentComponent launchInstance(Map<Resourcetype,Rat> description);
    Bool acquireInstance(DeploymentComponent instance);
    Bool releaseInstance(DeploymentComponent instance);
    Rat getAccumulatedCost();
    Unit shutdown();
    ...}
```



Services deployed on the cloud: Predicting behavior from models

- **Resource-aware design**: Build software that can dynamically modify its own deployment to improve performance and/or reduce cost

```
interface CloudProvider {
    DeploymentComponent launchInstance(Map<Resourcetype,Rat> description);
    Bool acquireInstance(DeploymentComponent instance);
    Bool releaseInstance(DeploymentComponent instance);
    Rat getAccumulatedCost();
    Unit shutdown();
    ...}
```



- **Model-based deployment decisions** at design time using service models
- **Formal semantics**: Architects and developers can simulate and analyze at design time how an application runs on the cloud

Modelling with Time

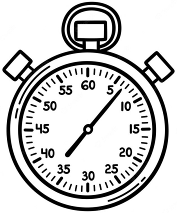
To express and compare resource usage, we need a notion of time in our models!

Modelling with Time

To express and compare resource usage, we need a notion of time in our models!

Timed semantics

- Let us assume that ABS programs execute under a **maximal progress** timed semantics
- Our model has a global clock
- **Assumption:** Execution is "infinitely fast" unless we say otherwise

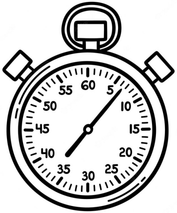


Modelling with Time

To express and compare resource usage, we need a notion of time in our models!

Timed semantics

- Let us assume that ABS programs execute under a **maximal progress** timed semantics
- Our model has a global clock
- **Assumption:** Execution is "infinitely fast" unless we say otherwise
- We have **explicit programming support** to express that execution takes time



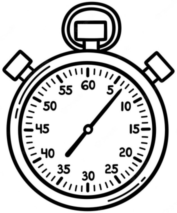
Modelling with Time

To express and compare resource usage, we need a notion of time in our models!

Timed semantics

- Let us assume that ABS programs execute under a **maximal progress** timed semantics
- Our model has a global clock
- **Assumption:** Execution is "infinitely fast" unless we say otherwise
- We have **explicit programming support** to express that execution takes time

Time x = **now**();

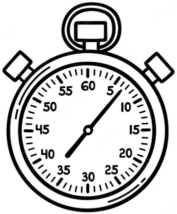


Modelling with Time

To express and compare resource usage, we need a notion of time in our models!

Timed semantics

- Let us assume that ABS programs execute under a **maximal progress** timed semantics
- Our model has a global clock
- **Assumption:** Execution is “infinitely fast” unless we say otherwise
- We have **explicit programming support** to express that execution takes time



Time x = **now**();

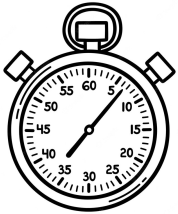
duration(x,y);

Modelling with Time

To express and compare resource usage, we need a notion of time in our models!

Timed semantics

- Let us assume that ABS programs execute under a **maximal progress** timed semantics
- Our model has a global clock
- **Assumption:** Execution is “infinitely fast” unless we say otherwise
- We have **explicit programming support** to express that execution takes time



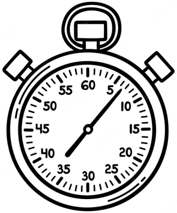
Time x = **now**();   **duration**(x,y);   **await duration**(x,y);

Modelling with Time

To express and compare resource usage, we need a notion of time in our models!

Timed semantics

- Let us assume that ABS programs execute under a **maximal progress** timed semantics
- Our model has a global clock
- **Assumption:** Execution is "infinitely fast" unless we say otherwise
- We have **explicit programming support** to express that execution takes time



Time x = **now**();

duration(x,y);

await duration(x,y);

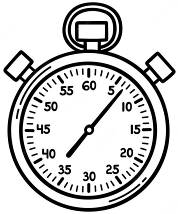
while (timeValue(**now**()) – x < 60) { **skip** } ;

Modelling with Time

To express and compare resource usage, we need a notion of time in our models!

Timed semantics

- Let us assume that ABS programs execute under a **maximal progress** timed semantics
- Our model has a global clock
- **Assumption:** Execution is "infinitely fast" unless we say otherwise
- We have **explicit programming support** to express that execution takes time



Time x = **now**(); **duration**(x,y); **await duration**(x,y);

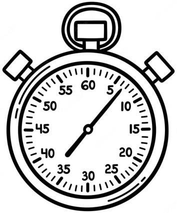
while (timeValue(**now**()) − x < 60) { **skip** } ; **duration**(60,60);

Modelling with Time

To express and compare resource usage, we need a notion of time in our models!

Timed semantics

- Let us assume that ABS programs execute under a **maximal progress** timed semantics
- Our model has a global clock
- **Assumption:** Execution is “infinitely fast” unless we say otherwise
- We have **explicit programming support** to express that execution takes time

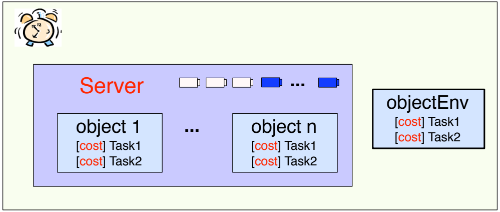


Time x = **now**(); **duration**(x,y); **await duration**(x,y);

while (timeValue(**now**()) − x < 60) { **skip** } ; **duration**(60,60); **await duration**(60,70);

Deployment Components

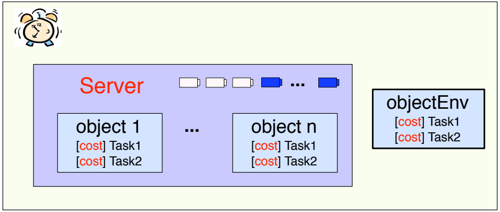
Deployment components are abstract "cost centers"



Deployment Components

Deployment components are abstract "cost centers"

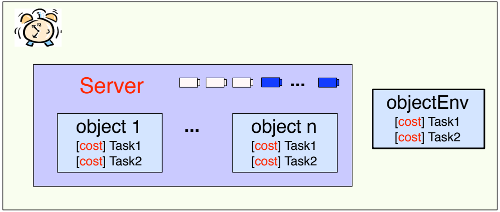
- Each deployment component has a given resource capacity
- Objects execute in the context of a deployment component



Deployment Components

Deployment components are abstract "cost centers"

- Each deployment component has a given resource capacity
- Objects execute in the context of a deployment component
- The *resources are shared* between the component's objects

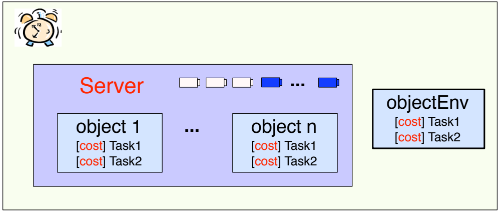


- Object execution *uses* resources in a deployment component (via Cost annotations)

Deployment Components

Deployment components are abstract "cost centers"

- Each deployment component has a given resource capacity
- Objects execute in the context of a deployment component
- The *resources are shared* between the component's objects

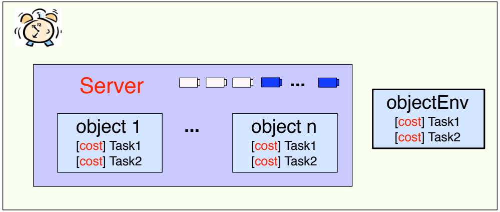


- Object execution *uses* resources in a deployment component (via Cost annotations)
- *How* resources are assigned and consumed, depends on the kind of resource

Deployment Components

Deployment components are abstract "cost centers"

- Each deployment component has a given resource capacity
- Objects execute in the context of a deployment component
- The *resources are shared* between the component's objects



- Object execution *uses* resources in a deployment component (via Cost annotations)
- *How* resources are assigned and consumed, depends on the kind of resource
- For cloud computing, let us focus on processing capacity

Example: Phone Services - Abstract Behavioral Model

Example: Phone Services - Abstract Behavioral Model

Telephone Service

```
interface TelephoneService {
    Unit call(Int calltime);
}
class TelephoneServer implements TelephoneService {
  Unit call(Int calltime){
    while (calltime > 0) { [Cost: 1] calltime = calltime − 1; await duration(1, 1); }
  }
}
```

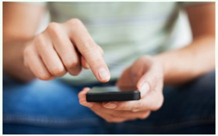
Example: Phone Services - Abstract Behavioral Model

Telephone Service

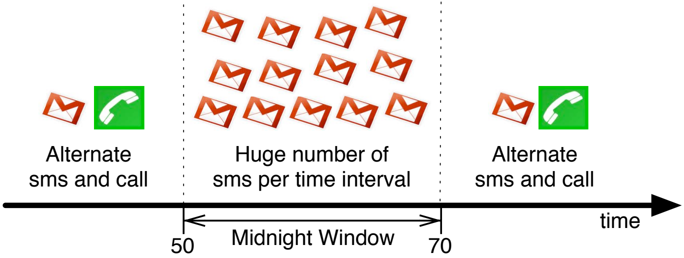
```
interface TelephoneService {
    Unit call(Int calltime);
}
class TelephoneServer implements TelephoneService {
  Unit call(Int calltime){
    while (calltime > 0) { [Cost: 1] calltime = calltime − 1; await duration(1, 1); }
  }
}
```

SMS Service

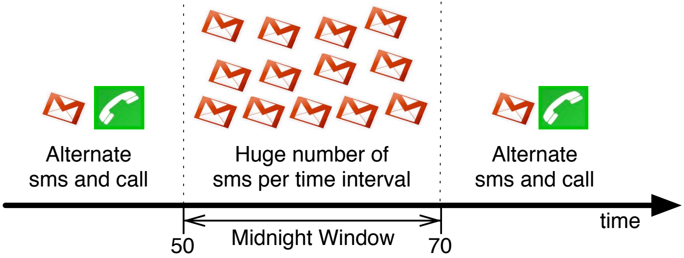
```
interface SMSService {
 Unit sendSMS();
}
class SMSServer implements SMSService {
  Unit sendSMS() {[Cost: 1] skip;}
}
```



Example: The New Year's Eve Client Behavior

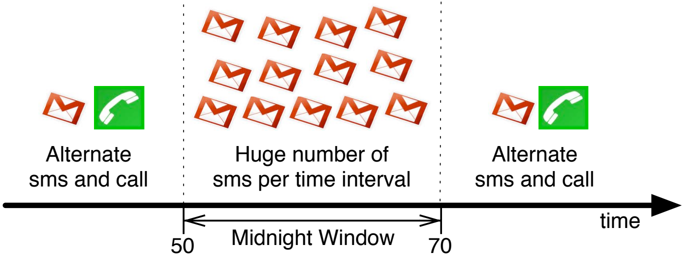


Example: The New Year's Eve Client Behavior




```
class NYEclient(Int frequency,TelephoneService ts,SMSService smss){
  Bool call=false;
  Unit normalBehavior(){ ... }
  Unit midnightWindow(){ ... } // Switch at appropriate time...
}
```

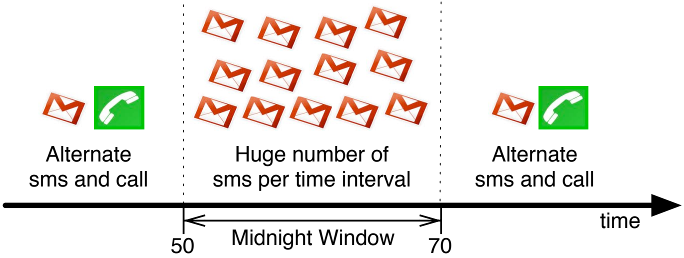
Example: The New Year's Eve Client Behavior




```
class NYEclient(Int frequency,TelephoneService ts,SMSService smss){
  Bool call=false;
  Unit normalBehavior(){ ... }
  Unit midnightWindow(){ ... } // Switch at appropriate time...
}
```

How to deploy the services?

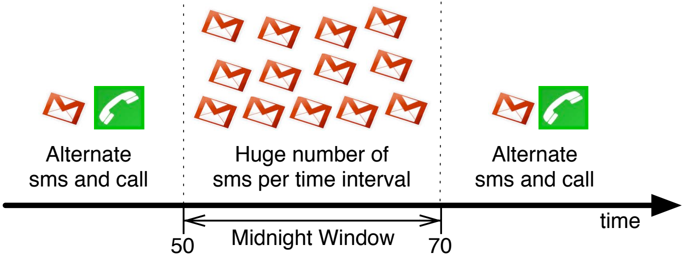
Example: The New Year's Eve Client Behavior




```
class NYEclient(Int frequency,TelephoneService ts,SMSService smss){
   Bool call=false;
   Unit normalBehavior(){ ... }
   Unit midnightWindow(){ ... } // Switch at appropriate time...
}
{// Main block:




}
```

How to deploy the services?

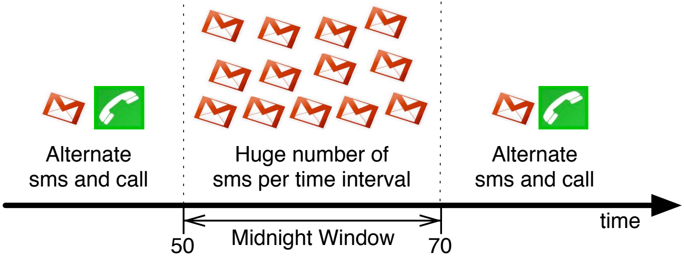
Example: The New Year's Eve Client Behavior




```
class NYEclient(Int frequency,TelephoneService ts,SMSService smss){
   Bool call=false;
   Unit normalBehavior(){ ... }
   Unit midnightWindow(){ ... } // Switch at appropriate time...
}
{// Main block:
      DC smscomp = new DeploymentComponent("smscomp", Speed(50));
      DC telcomp = new DeploymentComponent("telcomp", Speed(50));



}
```

How to deploy the services?

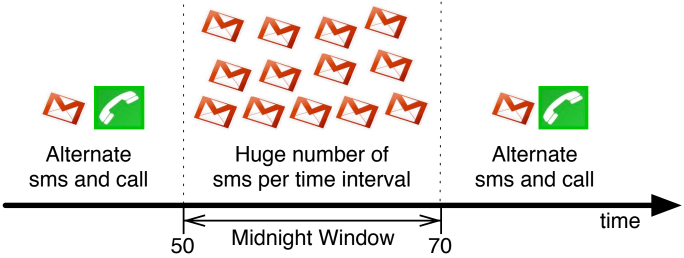
Example: The New Year's Eve Client Behavior




```
class NYEclient(Int frequency,TelephoneService ts,SMSService smss){
  Bool call=false;
  Unit normalBehavior(){ ... }
  Unit midnightWindow(){ ... } // Switch at appropriate time...
}
{// Main block:
    DC smscomp = new DeploymentComponent("smscomp", Speed(50));
    DC telcomp = new DeploymentComponent("telcomp", Speed(50));
    [DC: smscomp] SMSService sms = new SMSServer();
    [DC: telcomp] TelephoneService tel = new TelephoneServer();

}
```

How to deploy the services?

Example: The New Year's Eve Client Behavior



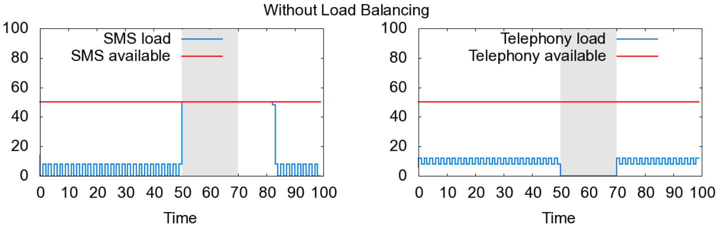

```
class NYEclient(Int frequency,TelephoneService ts,SMSService smss){
  Bool call=false;
  Unit normalBehavior(){ ... }
  Unit midnightWindow(){ ... } // Switch at appropriate time...
}
{// Main block:
    DC smscomp = new DeploymentComponent("smscomp", Speed(50));
    DC telcomp = new DeploymentComponent("telcomp", Speed(50));
    [DC: smscomp] SMSService sms = new SMSServer();
    [DC: telcomp] TelephoneService tel = new TelephoneServer();
    Client c = new NYEclient (1,tel,sms); ... // Clients
}
```

How to deploy the services?

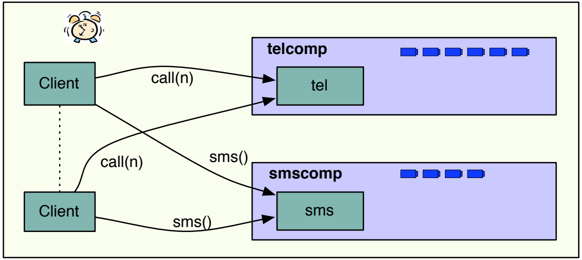
Example: NYEclient

```
class NYEclient (Int frequency, TelephoneServer ts, SMSServer smss) {
  Bool call = False;

  Unit normalBehavior() {
    if (timeValue(now()) > 50 && timeValue(now()) < 70) { this!midnightWindow(); }
    else {
      if (call) { await ts!call(1); } else { smss!sendSMS(); }
      call = ~ call;
      await duration(frequency,frequency); this!normalBehavior(); }
  }

  Unit midnightWindow() {
    if (timeValue(now()) >= 70) { this!normalBehavior(); }
    else {
      Int i = 0;
      while (i < 10) { smss!sendSMS(); i = i + 1; }
      await duration(1,1); this!midnightWindow(); }
  }

  Unit run(){ this!normalBehavior(); }
}
```

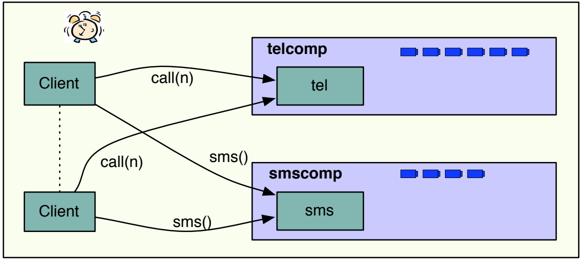
Example: Simulation Results



Load Balancing with Vertical Scaling



Load Balancing with Vertical Scaling

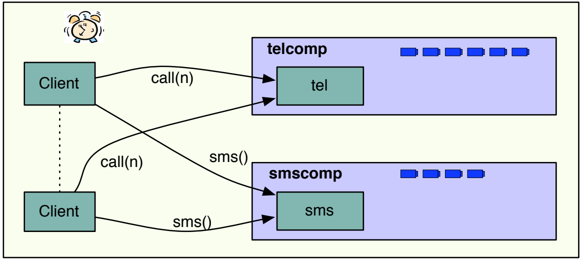


Resource awareness: monitor & react

- dc.**load**(rtype,e): average load on dc during the last e time intervals
- dc.**total**(rtype): currently allocated resources on dc
- dc.**transfer**(dc2, r, rtype): transfer r resources to dc2

- **thisDC**: reference to my deployment component

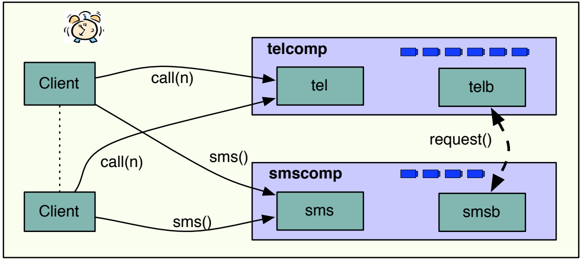
Load Balancing with Vertical Scaling



Resource awareness: monitor & react

- dc.**load**(rtype,e): average load on dc during the last e time intervals
- dc.**total**(rtype): currently allocated resources on dc
- dc.**transfer**(dc2, r, rtype): transfer r resources to dc2
- **thisDC**: reference to my deployment component
- **Reaction:** resource reallocation, object mobility, job distribution

Load Balancing with Vertical Scaling



Resource awareness: monitor & react

- dc.**load**(rtype,e): average load on dc during the last e time intervals
- dc.**total**(rtype): currently allocated resources on dc
- dc.**transfer**(dc2, r, rtype): transfer r resources to dc2
- **thisDC**: reference to my deployment component
- **Reaction:** resource reallocation, object mobility, job distribution

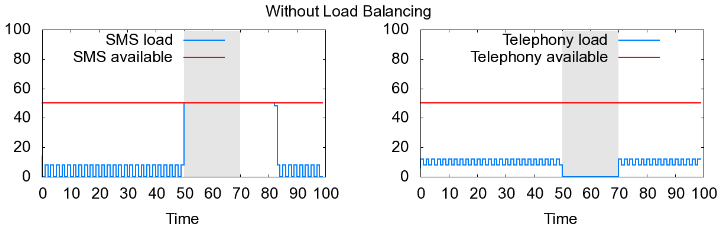
Load Balancing Strategy for the Phone Services

Reallocate $1/2 \times$total resources upon request from partner

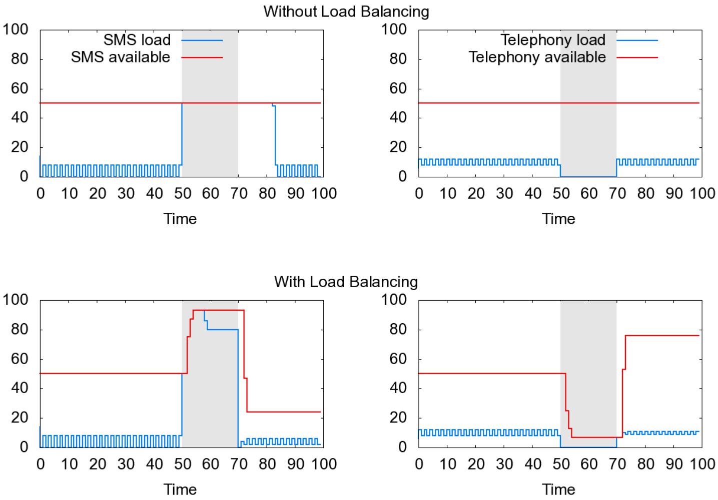
Exercise 4: Solution

```
class Balancer() implements Balancer {
  Balancer partner = null;

  Unit run() {
    await partner != null;
    while (True) {
        await duration(1, 1);
        Rat ld = await thisDC()!load(Speed, 1);
        if (ld > 90) { await partner!requestdc(thisDC()); }
    }
  }

  Unit requestdc(DC comp) {
    InfRat total = await thisDC()!total(Speed);
    Rat ld = await thisDC()!load(Speed, 1);
    if (ld < 50) { // we know total will not be InfRat
      thisDC()!transfer(comp, finvalue(total) / 2, Speed);
    }
  }

  Unit setPartner(Balancer p) { partner = p; }
}
```

Example: Simulation Results



Example: Simulation Results



Pause

BREAK: 15 minutes

An Interface to Cloud Providers

ABS modeling abstractions

- Dynamically created objects
- Very flexible synchronization:
  asynchronous method calls and futures
- Execution with time and resources
- Deployment components
  — can be dynamically created

An Interface to Cloud Providers

ABS modeling abstractions

- Dynamically created objects
- Very flexible synchronization: asynchronous method calls and futures
- Execution with time and resources
- Deployment components
  — can be dynamically created

All the building blocks we need to model an abstract **cloud provider** class

An Interface to Cloud Providers

ABS modeling abstractions

- Dynamically created objects
- Very flexible synchronization:
  asynchronous method calls and futures
- Execution with time and resources
- Deployment components
  — can be dynamically created

All the building blocks we need to model an abstract **cloud provider** class

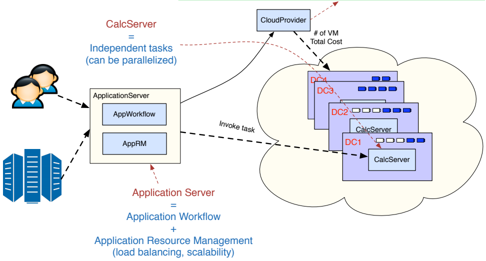
- DC launchInstance(Rat speed){...} *// create dc*
- Bool acquireInstance(DC instance){ ... } *// start dc*
- Bool releaseInstance(DC instance){ ... } *// stop dc*
- Rat getAccumulatedCost(){ ... }
- Unit shutdown(DC instance){ ... }

An Interface to Cloud Providers

ABS modeling abstractions

- Dynamically created objects
- Very flexible synchronization: asynchronous method calls and futures
- Execution with time and resources
- Deployment components
  — can be dynamically created

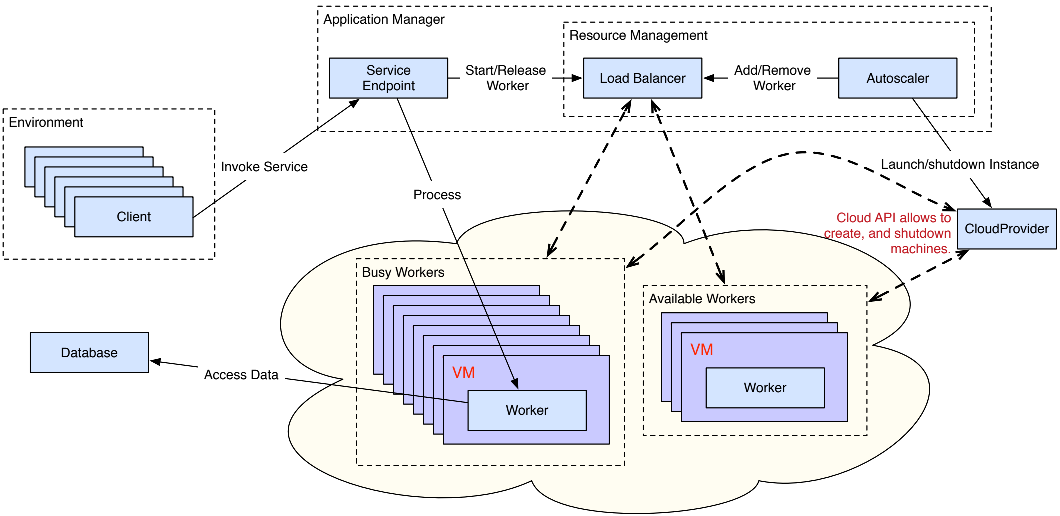
```
interface CloudProvider {
    DeploymentComponent launchInstance(Map<Resourcetype,Rat> description);
    Bool acquireInstance(DeploymentComponent instance);
    Bool releaseInstance(DeploymentComponent instance);
    Rat getAccumulatedCost();
    Unit shutdown();
    ...}
```



All the building blocks we need to model an abstract **cloud provider** class

- DC launchInstance(Rat speed){...} *// create dc*
- Bool acquireInstance(DC instance){ ... } *// start dc*
- Bool releaseInstance(DC instance){ ... } *// stop dc*
- Rat getAccumulatedCost(){ ... }
- Unit shutdown(DC instance){ ... }

Elastic Cloud Computing Architecture



Example: Load Balancing

Scenario

We assume given a class Worker. Define a Load Balancer class which implements round robin load balancing

- A field: List<Worker> available = list[];
- A field: List<Worker> inuse = list[];
- Implement a method getWorker(){...} which returns a Worker object
- Implement a method Unit releaseWorker(Worker w){...}
- Implement a method Unit addWorker(Worker w){...}

You can assume operations on lists: head, tail, isEmpty, appendright, without

A Round-Robin Load Balancer

```
class RoundRobinLoadBalancer() implements LoadBalancer {
  List<Worker> available = list[];
  List<Worker> inuse = list[];

  Worker getWorker(){
    await (!isEmpty(available));
    Worker w = head(available);
    available = tail(available);
    inuse = appendright(inuse,w);
    return w;
  }

  Unit releaseWorker(Worker w){ available = appendright(available,w); inuse = without(inuse,w); }

  Unit addWorker(Worker w){ available = appendright(available,w); }
}
```

Example: Workers

Let us define tasks and workflows

Example: Workers

Let us define tasks and workflows

```
type TaskID=Int;
data Workflow=Workflow(List<Task> tasks);
data Task=Task(TaskID tID, Rat cost, List<TaskID> dependencies);
```

Example: Workers

Let us define tasks and workflows

```
type TaskID=Int;
data Workflow=Workflow(List<Task> tasks);
data Task=Task(TaskID tID, Rat cost, List<TaskID> dependencies);
```

Scenario

Define a Worker class

- Implements a method processTask which receives a task
- The Worker has already been started,
- It should stop when execution is completed (i.e., returned to the load balancer)

Workers

Here, we assume given

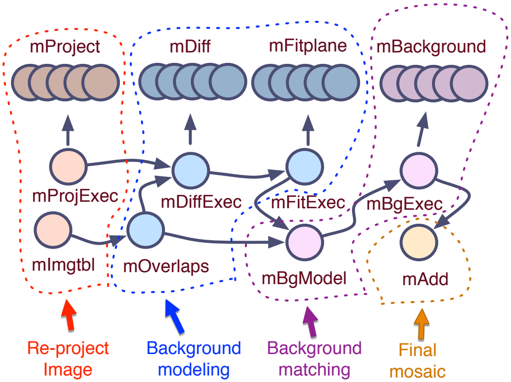
- a load balancer lb,
- futures for all tasks in the dependencies,
- a deadline for task execution, and
- a starting time.

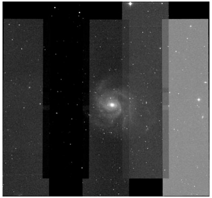
```
class WorkerObject(LoadBalancer lb) implements Worker {

  Bool processTask(Rat taskCost, Time started, Duration deadline, List<Fut<Bool>> dependencies){
  while(dependencies != Nil) {
    Fut<Bool> dep = head(dependencies);
    dependencies = tail(dependencies);
    await dep?;
  }
  [Cost: taskCost] skip; // This is the part we abstract from!
  Rat spentTime = timeDifference(now(),started);
  lb!releaseWorker(this);
  return (spentTime <= durationValue(deadline));
}
```

Validating the Models

Case Study: Montage (1)

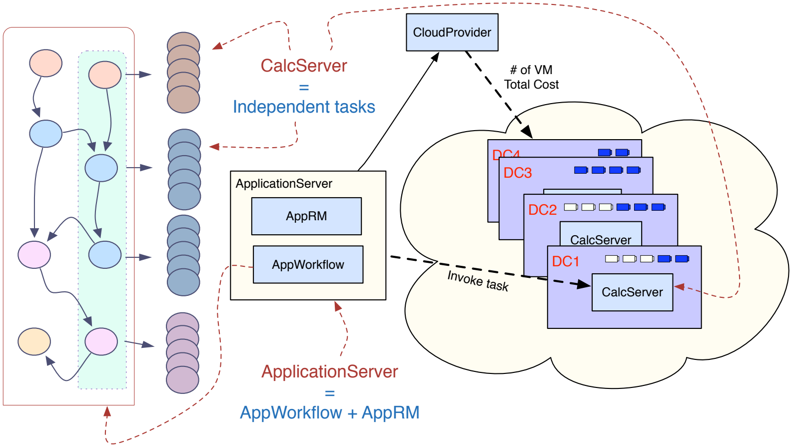
Montage is a toolkit for assembling astronomical images into customized mosaics
http://montage.ipac.caltech.edu/





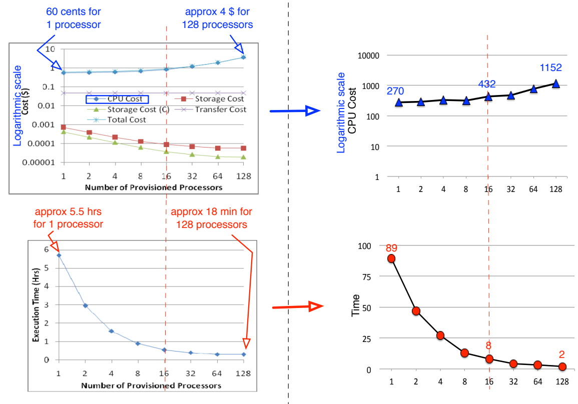
Partly ordered workflow and highly parallelizable tasks

Case Study: Montage (2)



Model the Montage toolkit using the **Cloud Provider API**, run simulations varying the deployment scenario, and compare the results.

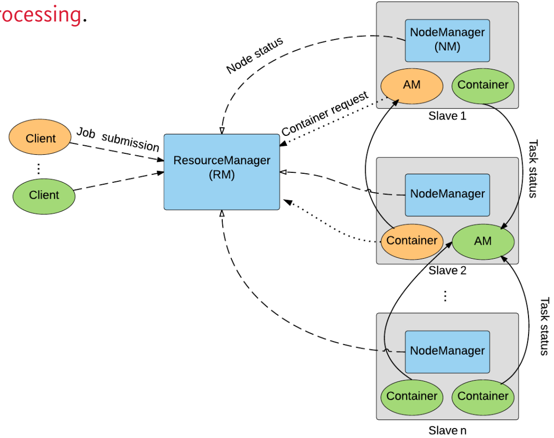
Cost vs. Time Tradeoff: can we reproduce the results of other informal cloud simulation tools (GridSim)?



The cost of doing science on the cloud: The Montage example.
E. Deelman, G. Singh, M. Livny, G. B. Berriman, and J. Good.
(SC'08), pages 1–12. IEEE/ACM, 2008.

Case Study: Big Data Processing Frameworks (1)

Hadoop YARN and SPARK Clusters:
Open-source software framework that implements a cluster management technology for distributed processing.

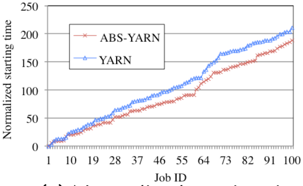


Popular cloud framework for big data processing:

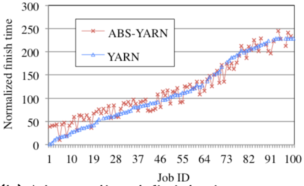
- Resource allocation
- Code distribution
- Distributed data processing
- Streaming data

Case Study: Big Data Processing Frameworks (2)

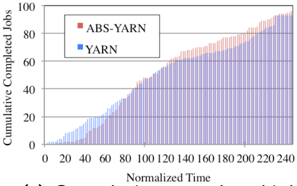
How does the ABS YARN compare to the actual YARN implementation?



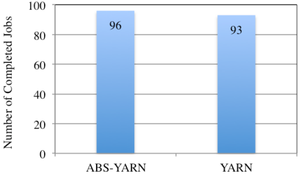
(a) Normalized starting time



(b) Normalized finish time



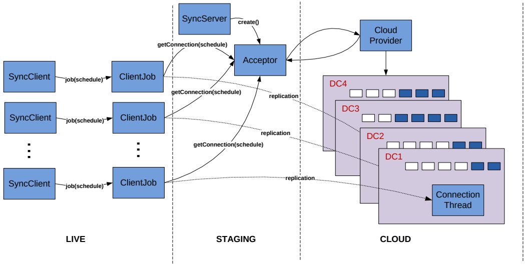
(c) Cumulative completed jobs



(d) Total number of completed jobs

Case Study: Fredhopper Replication Server (1)

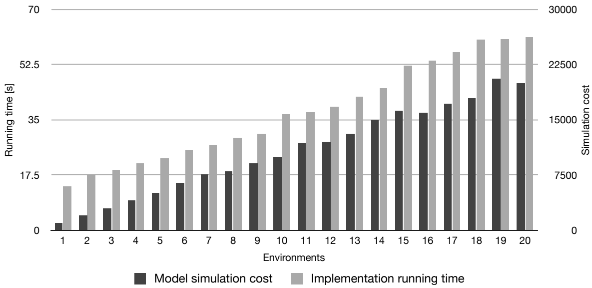
The Fredhopper Access Server (FAS) is a distributed, concurrent OO system providing search and merchandising services to e-Commerce companies.
The Replication Server is one part of FAS.



- **Very detailed model**: consists of 5000 lines of ABS
- **Model API:** use actual system logs to run simulations
- **SLA monitoring framework** for failure metrics at end-points

Case Study: Fredhopper Replication Server (2)

How does the accumulated cost in our model compare to the actual Java implementation?



Measured execution time of the implementation (left scale)
Accumulated cost of the simulation (right scale)

The deviation roughly seems to correspond to the start-up time of JVM

Kubernetes

Kubernetes

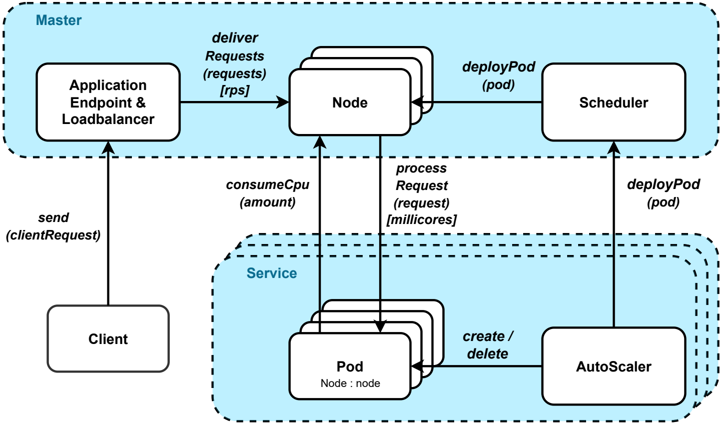
- *Service*: logical collection of load-balanced pods, with given service end-point & load balancer
- *Container*: units of deployment, corresponds to *Workers* in the previous slides
- *Pods*: scheduling unit, group of containers (one service container + sidecar and helper containers for health probing etc)
- *Nodes*: Explicit resource capabilities (CPU, memory, etc), contain pods
- *Scheduler* assigns pods to nodes based on resource requirements and capacities

Kubernetes

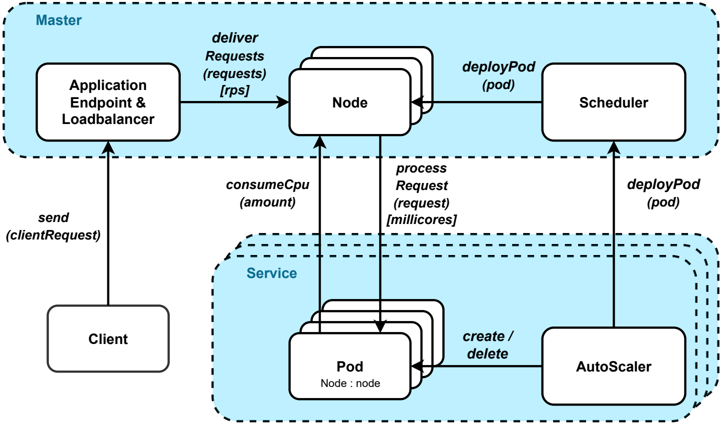
- *Service*: logical collection of load-balanced pods, with given service end-point & load balancer
- *Container*: units of deployment, corresponds to *Workers* in the previous slides
- *Pods*: scheduling unit, group of containers (one service container + sidecar and helper containers for health probing etc)
- *Nodes*: Explicit resource capabilities (CPU, memory, etc), contain pods
- *Scheduler* assigns pods to nodes based on resource requirements and capacities

- *Load balancing*: choice of loadbalancer crucial to the performance of a Kubernetes cluster (layer-4, layer-7)

Kubernetes Model

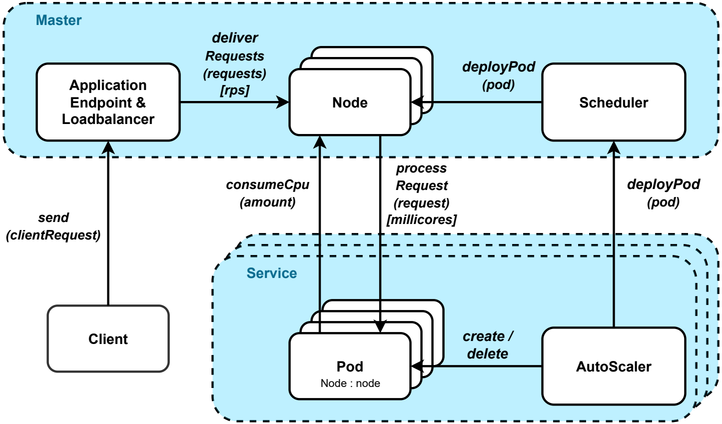


Kubernetes Model



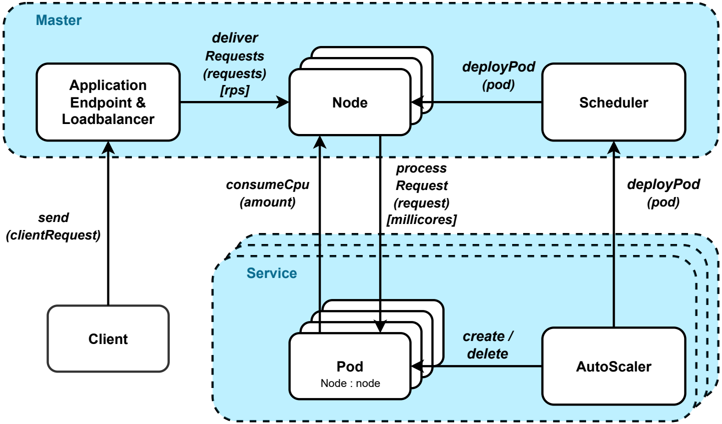
Challenge: A cluster can support many workflows.

Kubernetes Model



Challenge: A cluster can support many workflows. **Solution:** Costs depend on wf and node configuration

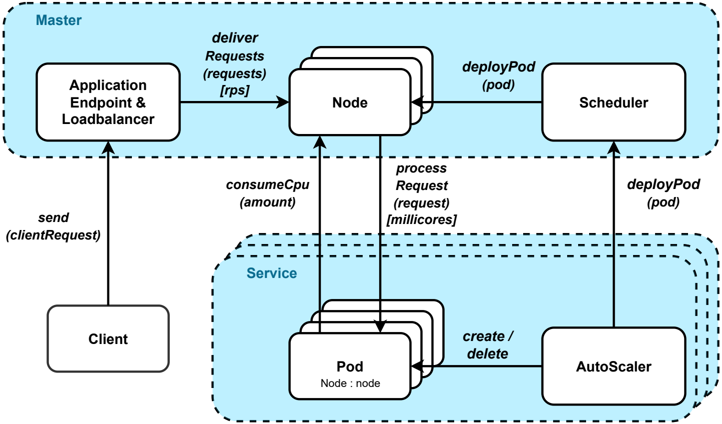
Kubernetes Model



Challenge: A cluster can support many workflows. **Solution:** Costs depend on wf and node configuration

Problem: Pods are not fully isolated.

Kubernetes Model



Challenge: A cluster can support many workflows. **Solution:** Costs depend on wf and node configuration

Problem: Pods are not fully isolated. **Solution:** Costs modelled in tables, depend on wf/node config

Building a Kubernetes Model from an Application

Methodology

1. Instrument the cluster
2. Identify suitable workflows
3. Identify node configurations
4. Define a sampling strategy for service loads to derive cost tables
5. Perform model-based predictions by means of simulations

Building a Kubernetes Model from an Application

Methodology

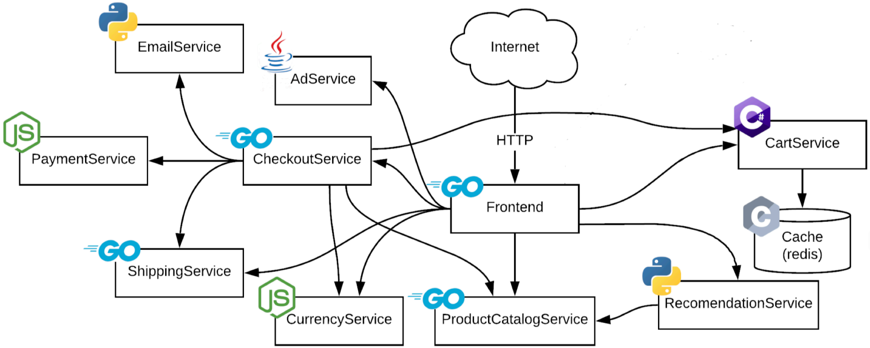
1. Instrument the cluster
2. Identify suitable workflows
3. Identify node configurations
4. Define a sampling strategy for service loads to derive cost tables
5. Perform model-based predictions by means of simulations

Cost tables

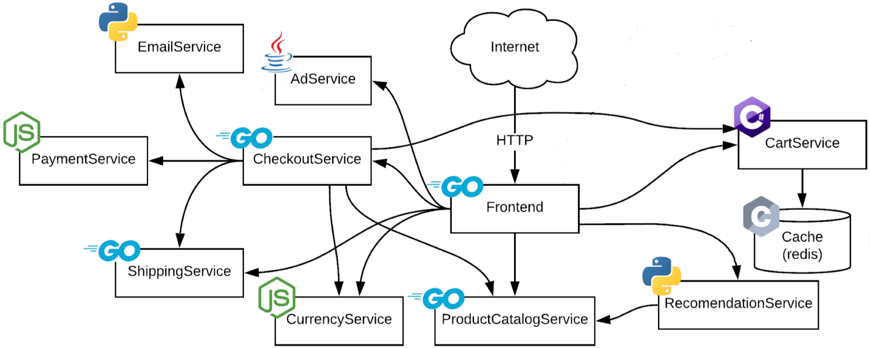
- The cost table captures resource consumption on a node for specific configurations.
- The cost table stores information about resource consumption for each workflow and service for different RPS entries

$$(\text{workflow}, \text{serviceName}, \text{RPS}) \mapsto \text{cost}$$

Example: Google's Online Boutique Microservices Demo



Example: Google's Online Boutique Microservices Demo



- Workflows: get index page, change currency, view random product, …
- Workflows like these invovle a surprising number of services!
- We obtain very good prediction models for static deployments!
- Source: Google microservices demo

Conclusions

Work with executable actor models!

- ABS models are **easy to understand** and **very efficient** for exploring designs and deployment decisions for distributed software
- Actors, asynchronous method calls: express flexible synchronization patterns
- Basic building blocks for resource-aware models: Cost annotations and deployment components
- Service end-points, load balancers, autoscaling etc etc easy to model
- Methodology for how to model of a real system: a concrete Kubernetes cluster with diverse workflows

Conclusions

Work with executable actor models!

- ABS models are **easy to understand** and **very efficient** for exploring designs and deployment decisions for distributed software
- Actors, asynchronous method calls: express flexible synchronization patterns
- Basic building blocks for resource-aware models: Cost annotations and deployment components
- Service end-points, load balancers, autoscaling etc etc easy to model
- Methodology for how to model of a real system: a concrete Kubernetes cluster with diverse workflows

- ABS is **open source** and **well documented**: http://www.abs-models.org

References

Einar Broch Johnsen, Rudolf Schlatte, Silvia Lizeth Tapia Tarifa: Modeling Resource-Aware Virtualized Applications for the Cloud in Real-Time ABS. ICFEM 2012: 71-86 [Preprint]

Einar Broch Johnsen, Rudolf Schlatte, Silvia Lizeth Tapia Tarifa: Integrating deployment architectures and resource consumption in timed object-oriented models. J. Log. Algebraic Methods Program. 84(1): 67-91 (2015) [Preprint]

Einar Broch Johnsen, Ka I Pun, Silvia Lizeth Tapia Tarifa: A formal model of cloud-deployed software and its application to workflow processing. SoftCOM 2017: 1-6 [Preprint]

Gianluca Turin, Andrea Borgarelli, Simone Donetti, Ferruccio Damiani, Einar Broch Johnsen, Silvia Lizeth Tapia Tarifa: Predicting resource consumption of Kubernetes container systems using resource models. J. Syst. Softw. 203: 111750 (2023) [Preprint]