



Project N°: **FP7-610582**
Project Acronym: **ENVISAGE**
Project Title: **Engineering Virtualized Services**
Instrument: **Collaborative Project**
Scheme: **Information & Communication Technologies**

Deliverable D4.3.2

Resource-aware Modeling the FRH Case Study

Date of document: T22



Start date of the project: **1st October 2013**

Duration: **36 months**

Organisation name of lead contractor for this deliverable: **FRH**

Final version

STREP Project supported by the 7th Framework Programme of the EC		
Dissemination level		
PU	Public	✓
PP	Restricted to other programme participants (including Commission Services)	
RE	Restricted to a group specified by the consortium (including Commission Services)	
CO	Confidential, only for members of the consortium (including Commission Services)	

Executive Summary:

Resource-aware Modeling the FRH Case Study

This document summarises deliverable D4.3.2 of project FP7-610582 (**Envisage**), a Collaborative Project supported by the 7th Framework Programme of the EC. within the Information & Communication Technologies scheme. Full information on this project is available online at <http://www.envisage-project.eu>.

This deliverable reports on the modeling of the resource-aware version of the FRH case study and deployment scenarios in the abstract behavioral specification language, using the **Envisage** modeling techniques. Based on the application of these techniques, we provide detailed feedback to T1.2, T1.3, T1.4, T2.2, T2.3.

List of Authors

Stijn de Gouw (FRH)
Behrooz Nobakht (FRH)
David Costa (FRH)

Contents

1	Introduction	4
2	Fredhopper Cloud Services	5
2.1	Building Blocks	5
2.1.1	Service Endpoints	5
2.1.2	Service Instances	6
2.1.3	Load Balancing Service	6
2.1.4	Platform Service	6
2.1.5	Deployment Service	6
2.1.6	Infrastructure Service	6
2.1.7	Monitoring and Alerting Service	7
2.2	Object Oriented Design	7
2.2.1	Resources and Virtual machines	7
2.2.2	Service Configuration	7
2.2.3	Service Endpoints and Instances	8
2.2.4	Service Architecture	12
2.2.5	Monitoring	16
2.3	Resource consumption and deployment	18
2.4	Summary	22
3	Experience and feedback to technical tasks	27
4	Summary	30
	Bibliography	30
	Glossary	32

Chapter 1

Introduction

FRH develops the Fredhopper Cloud Services to offer search and targeting facilities on a large product database to e-Commerce companies as services (SaaS) over the cloud computing infrastructure (IaaS). In Task 4.3 we conduct a case study on the Fredhopper Cloud Services, in which we aim to investigate the correspondence between user-level SLAs and lower level performance metrics. In particular, to fulfill user-level SLAs, our service deployment may offer SLA-aware services, evolve service implementation and configure cloud resource usage autonomously.

In the previous deliverable D4.3.1, we provided an initial model in the Abstract Behavioural Specification language (ABS) [1, 5] of the structural and functional aspects of the Fredhopper Cloud Services. This model forms the starting point for the resource modeling, formal specification and monitor generation. Structurally at the block level (i.e. the kinds of services running in the FRH cloud) which is described in Section 2.1, the model remains the same. Inside the blocks, the various services are extended with resource-awareness and deployment concerns; this is detailed in Sections 2.2 and 2.3.

Support for deployment of services that allow autonomous management of cloud resource usage - based on formalizations of SLA's - requires a *resource-aware* model of the Fredhopper Cloud Services. This deliverable D4.3.2 presents in Chapter 2 how the techniques developed in WP1 and WP2 were applied to obtain a resource-aware model. We show how resources are integrated through virtual machines and cost annotations - based on measurements from real-world log files - that specify resource consumption. Furthermore, we present the detailed modeling of deployment scenarios in a declarative manner. Based on these applications, in Chapter 3 we give detailed feedback to various ongoing technical tasks, pointing out desired extensions and refinements. In particular, we provide input for the final phase of T1.2, T1.3, T1.4, T2.2 and T2.3.

Chapter 2

Fredhopper Cloud Services

Support for deployment of services that allow autonomous management of cloud resource usage - based on formalizations of SLA's - requires a *resource-aware* model of the Fredhopper Cloud Services. In this chapter, we apply the **Envisage** framework to obtain such a model. In particular, we present the modeling of deployment scenarios, and show how resources are integrated through virtual machines and cost annotations that specify resource consumption. The initial ABS model of the Fredhopper Cloud Services, which was presented in the previous deliverable D4.3.1, forms the starting point.

2.1 Building Blocks

Figure 2.1 shows a block diagram of the Fredhopper Cloud Services, where the arrows indicate service consumption and service provision.

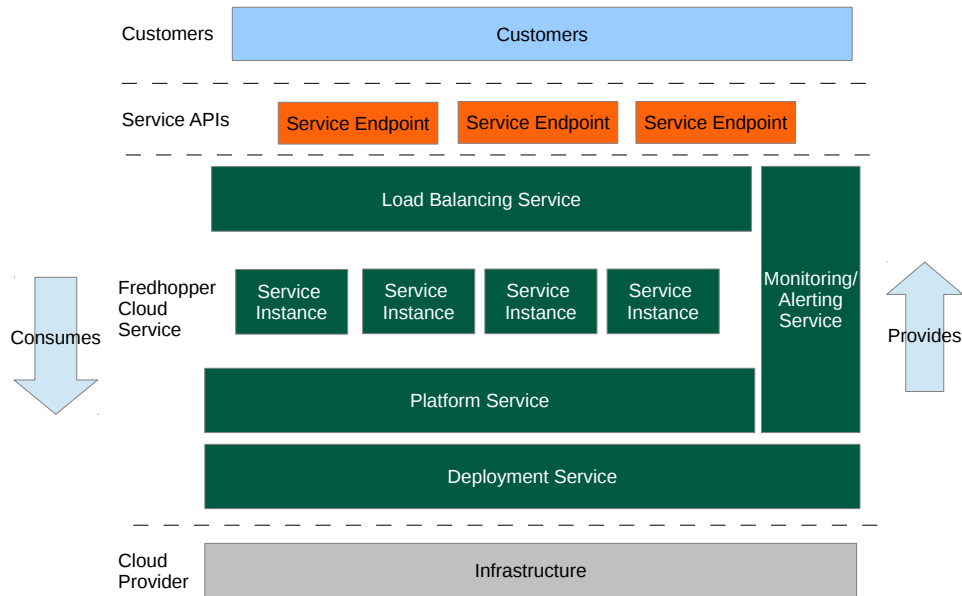


Figure 2.1: Block diagram of the Fredhopper Cloud Services

2.1.1 Service Endpoints

Fredhopper Cloud Services provides several SaaS offerings on the cloud. These services are exposed via endpoints. In practice these endpoints typically are implemented to be RESTful and accept communications over HTTP. For example, one of the services offered by these endpoints is the Fredhopper query service,

which allows users to query over their product catalogue via full text search¹ and faceted navigation². Service endpoints are exposed via the Load Balancing Service that distributes requests over multiple *service instances*.

2.1.2 Service Instances

The advantages of offering software as a service on the cloud over on-premise deployment include the following:

- to increase fault tolerance;
- to handle dynamic throughputs;
- to provide seamless service update;
- to increase service testability; and
- to improve the management of infrastructure.

To fully utilize the cloud computing paradigm, software must be designed to be *horizontally* scalable³. Typically, software services are deployed as *service instances*. Each instance offers the same service and is exposed via the Load Balancing Service, which in turn offers a service endpoint (Figure 2.1). Requests through the endpoint are then distributed over the instances. In the event of increasing/decreasing throughput, more/less instances may be deployed and be exposed through the same endpoint. Moreover, at any time, if an instance stops accepting requests, a new instance may be deployed in place.

2.1.3 Load Balancing Service

The Load Balancing Service is responsible for distributing requests from service endpoints to their corresponding instances. Currently at FRH, this service is implemented by HAProxy (www.haproxy.org), a TCP/HTTP load balancer.

2.1.4 Platform Service

The Platform Service provides an *interface* to the *Cloud Engineers* [4, Table 3.1] to deploy and manage service instances and to expose them through service endpoints. The Platform Service takes a service specification, which includes a *resource configuration* for the service [4, Section 3.1], and creates and deploys the specified service. A service specification from a customer determines which type of service is being offered, the number of service instances to be deployed initially and the amount of *virtualized resources* to be consumed by instance.

2.1.5 Deployment Service

The Deployment Service provides an API to the Platform Service to deploy service instances onto specified virtualized resources provided by the *Infrastructure Service*. The API also offers operations to control the lifecycle of the deployed service instances. The Deployment Service allows the Fredhopper Cloud Services to be independent of the specific infrastructure that underlies the service instances.

2.1.6 Infrastructure Service

The Infrastructure Service offers an API to the Deployment Service to acquire and release virtualized resources. At the time of writing the Fredhopper Cloud Services utilizes virtualized resources from the Amazon Web Services (aws.amazon.com), where processing and memory resources are exposed through Elastic Compute Cloud instances (<https://aws.amazon.com/ec2/instance-types/>).

¹en.wikipedia.org/wiki/Full_text_search

²en.wikipedia.org/wiki/Faceted_navigation

³en.wikipedia.org/wiki/Scalability#Horizontal_and_vertical_scaling

2.1.7 Monitoring and Alerting Service

The Monitoring and Alerting Service provides 24/7 monitoring services on the functional and non-functional properties of the services offered by the Fredhopper Cloud Services, the service instances deployed by the Platform Service, and the healthiness of the acquired virtualized resources.

If a monitored property is not satisfied, *Cloud Engineers* are alerted via emails and SMS messages and *Cloud Engineers* can react accordingly. For example, if the query throughput of a service instance is below a certain threshold, *Cloud Engineers* increase the amount of resources allocated to that service. For broken functional properties, such as a runtime error during service uptime, *Cloud Engineers* notify *Software Engineers* for further analysis.

2.2 Object Oriented Design

In order to apply the Envisage framework, to provide feedback to its ongoing development and to evaluate its effectiveness, we develop a resource-aware ABS model of the Fredhopper Cloud Services in Task 4.3. In this section we provide an overview of the resource-aware version of the ABS model of the Fredhopper Cloud Services.

2.2.1 Resources and Virtual machines

To support a fine-grained management of resource usage, we first model the kinds of virtual machines that are available, together with their associated resource properties and cost. Capturing the detailed resource properties of a virtual machine is a prerequisite to be able to make an informed decision which kind of virtual machine is appropriate to use when scaling. For instance, if the bandwidth is identified as a bottleneck using the monitoring framework, use virtual machines with enhanced networking.

As noted above, the Fredhopper Cloud Services currently utilizes Amazon AWS instances. Figure 2.2 shows a JSON file with several kinds of AWS instances, and their associated resources. “CPU” denotes the number of cores, “Memory” is the size of the memory (in MiB) and “IO” specifies the capacity of the storage device in GB.

The JSON file is processed and converted automatically into a representation of the virtual machines in the ABS. Figure 2.3 defines an ABS data type that enumerates the types of virtual machines.

The capacity of the resources associated to each type of virtual machine is stored in a map of type `Map<ResourceType, Rat>`. Figure 2.4 shows the map with the properties of each kind of virtual machine. The “CostPerInterval” property indicates the pricing of an instance per hour. The priced used are for *on-demand instances*: instances paid for by the hour, rather than “reserved” instances. Using a combination of on-demand instances *and* reserved instances for the same kind of virtual machine is possible by distinguishing two different virtual machine types (i.e., `C3_LARGE_RESERVED` and `C3_LARGE_ONDEMAND`) with the same resources but different cost per interval.

2.2.2 Service Configuration

Figure 2.5 shows the basic data types involved in a service configuration. A service configuration is modeled as a `Config` value that consists of the service type (`ServiceType`), the number of service instances and its resource requirement (`List< Map<ResourceType, Rat> >`). Given a configuration `c`, the value `instances(c)` is a list of resource descriptions `l` such that the `length(l)` is the number of service instances to be deployed and the `n`-th map in the list denotes the minimal amount of resources required for the `n`-th instance. The resource requirements are used to identify a suitable virtual machine. For instance, if the map for the `n`-th instance contains the pair `Pair(Memory, 3750)`, then the virtual machine on which the `n`-th service will be deployed should have at least 3750 MiB of memory.

```

"DC_description":
[ {
  "name" : "c3.large",
  "provide_resources" : {"IO" : 32, "CPU" : 2, "Memory" : 3750},
  "cost" : 105
},
{
  "name" : "c3.xlarge",
  "provide_resources" : {"IO" : 80, "CPU" : 4, "Memory" : 7500},
  "cost" : 210
},
{
  "name" : "c3.2xlarge",
  "provide_resources" : {"IO" : 160, "CPU" : 8, "Memory" : 15000},
  "cost" : 420
},
{
  "name" : "m3.medium",
  "provide_resources" : {"IO" : 4, "CPU" : 1, "Memory" : 3750},
  "cost" : 70
},
...
]

```

Figure 2.2: JSON file with virtual machine types

```

data VMType =
    C3_LARGE
  | C3_XLARGE
  | C3_2XLARGE
  | M3_MEDIUM
  | ...
;

```

Figure 2.3: Enumeration of virtual machine types for Amazon infrastructure provider

2.2.3 Service Endpoints and Instances

Figure 2.6 shows the static structure of an endpoint and its implementation and Figure 2.7 presents the corresponding ABS interface definition. The interface **EndPoint** models a service endpoint. It can be invoked (**invoke(Request)**) with a request of type **Request**. Currently **Request** is a type synonym to **Integer** to denote the *size* of the request. The method returns **Response** value to denote the corresponding response. Currently **Response** is a type synonym to **Boolean** to denote whether the request is successful. It is **True** if the invocation is successful, and **False** otherwise. In the production system of the Fredhopper Cloud Services, the resource utilization, may be dependent on the following two factors:

- the amount of data over which a request queries⁴; and
- the size of the corresponding (HTTP) response.

We use a type synonym to **Integer** as an argument to **invoke(Request)** for modeling these factors. The interface **EndPoint** is extended by **Service** and **LoadBalancerEndPoint**.

⁴In the context of the Query API, this is the size of the underlying product catalog


```

/*data Ressourcetype = CPU
    | Memory
    | Bandwidth
    | CostPerInterval
    | ...; // Defined in ABS.DC
*/

def Map<Ressourcetype, Rat> vmResources(VMTyp v) =
    case v {
        C3_LARGE => map[Pair(Memory,3750), Pair(CPU,2),
                           Pair(CostPerInterval, 105/1000)];
        C3_XLARGE => map[Pair(Memory,7500), Pair(CPU,4),
                           Pair(CostPerInterval, 210/1000)];
        C3_2XLARGE => map[Pair(Memory,15000), Pair(CPU,8),
                           Pair(CostPerInterval, 420/1000)];
        M3_MEDIUM => map[Pair(Memory,3750), Pair(CPU,1),
                           Pair(CostPerInterval, 67/1000)];
        ...
    };

```

Figure 2.4: Properties of virtual machine types

```

data ServiceType = ..;
data Config = Config(ServiceType serviceType, List< Map<Ressourcetype, Rat> > instances);

```

Figure 2.5: Service specification

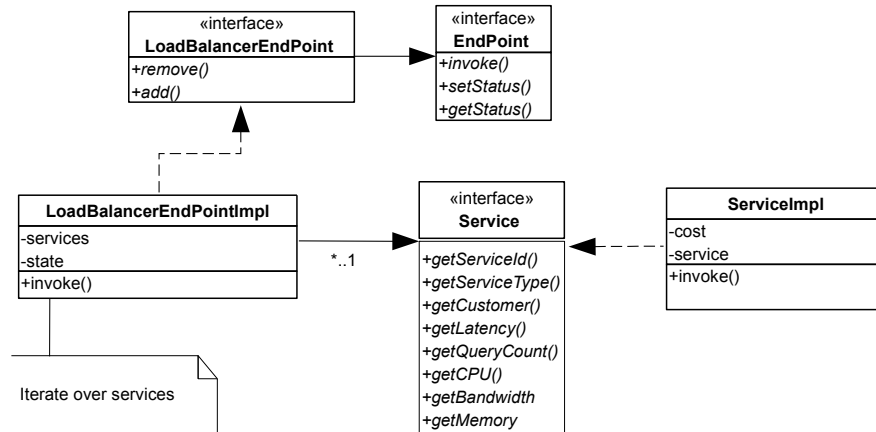


Figure 2.6: UML class diagram of the Fredhopper Cloud Services (1)

The interface **Service** models a service instance that performs the actual computation for the request received by its service endpoint.

Figure 2.8 shows the implementation of **ServiceImpl.invoke(Int)**. The class implementation takes as arguments at construction its Integer **id**, its service type **st**, the name of the customer to which the service is provided **c**, and the **cost** value, which denotes the amount of CPU resources consumed by the request when the size of the request is 1. Besides these two parameters, the exact latency of a request depends on the amount of resources available when performing the request: latency decreases if the virtual machine that

```

type Id = Int; def Id init() = 1; def Id incr(Id id) = id + 1;
type Request = Int; def Int cost(Request r) = r;
type Response = Bool; def Response success() = True; def Bool isSuccess(Response r) = r;

interface EndPoint {
  Response invoke(Request req);
  Unit setStatus(State status);
  State getStatus();
}

interface LoadBalancerEndPoint extends EndPoint {
  Bool remove(Service service);
  Bool add(Service service);
}

interface Service extends EndPoint {
  Id getServiceId();
  ServiceType getServiceType();
  Customer getCustomer();
  Int getLatency();
  Int getRequestCount();
  Rat getCPU();
  Rat getBandwidth();
  Rat getMemory();
}

```

Figure 2.7: ABS interface of the Fredhopper Cloud Services (1)

```

type Customer = String;
class ServiceImpl(Id id, ServiceType st, Customer c, Int cost) {
  Int latency = 0; Int log = 0;
  ..
  Int cost(Request request) {
    return max(1, cost(request)) * cost;
  }

  Response invoke(Request request) {
    Int cost = this.cost(request);
    Int time = currenttms();
    [Cost: cost] this.log = this.log + 1;
    time = currenttms() - time;
    this.latency = max(this.latency, time);
    return success();
  }
  ..
}

```

Figure 2.8: Implementation of ServiceImpl.invoke(Int)

processes the request has more CPU resources, and latency increases if **cost** and **Request** increase.

We use the type synonym **Customer** to model the customer's name. The function **currenttms()** is a built-in ABS function that returns the current clock cycle, and function **max(a, b)** returns the larger value of **a** and **b**. The method **invoke** uses the cost annotation **[Cost: cost]** to denote the required number of

CPU units to execute the annotated statement.

```

class LoadBalancerEndPointImpl(List<Service> services) implements LoadBalancerEndPoint {
  List<Service> current = services;
  { assert this.services != Nil; }
  Response invoke(Request request) {
    if (this.current == Nil) {
      this.current = this.services;
    }
    Service ser = head(this.current);
    this.current = tail(this.current);
    return await ser!invoke(request);
  }
  ..
}

```

Figure 2.9: Implementation of `LoadBalancerEndPointImpl.invoke(Request)`

```

class LoadBalanceCPU(List<Service> services) implements LoadBalancerEndPoint {
  { assert this.services != Nil; }
  Response invoke(Request request) {
    List<Service> remaining = services;

    Service best = head(remaining);
    Fut<Rat> fMostCPU = ser!getCPU();
    Rat mostCPU = fMostCPU.get;

    while(remaining != Nil) {
      remaining = tail(remaining);
      Service ser = head(remaining);
      Fut<Rat> fCPU = ser!getCPU();
      Rat cpu = fCPU.get;
      if(cpu < mostCPU) {
        best = ser;
        mostCPU = cpu;
      }
    }

    return await best!invoke(request);
  }
  ..
}

```

Figure 2.10: Implementation of `LoadBalancerEndPointImpl.invoke(Request)`

The interface `LoadBalancerEndPoint` extends interface `EndPoint` with the ability to dynamically associate service instances to a service endpoint, thereby allowing requests to the endpoint to be distributed. The class `LoadBalancerEndPointImpl` implements `LoadBalancerEndPoint` and its implementation of `invoke(Request)` is shown in Figure 2.10. This method implements a simple round-robin load balancing strategy to distribute requests. The class implementation `LoadBalancerEndPointImpl` is parametric to a non-empty list of unique `Service` references. The class `LoadBalancerOptimizeCPU` implements a resource-aware load balancer to distribute requests. Specifically, a request is directed to the service instance that currently has the most CPU resources available.

2.2.4 Service Architecture

Figure 2.11 shows the static structure of the Fredhopper Cloud Services, and Figure 2.12 shows the corresponding interfaces in ABS. We present the modeling of the Monitoring and Alerting Service in Section 2.2.5. The static structure shown in Figure 2.11 models dependencies between various services in the Fredhopper Cloud Services.

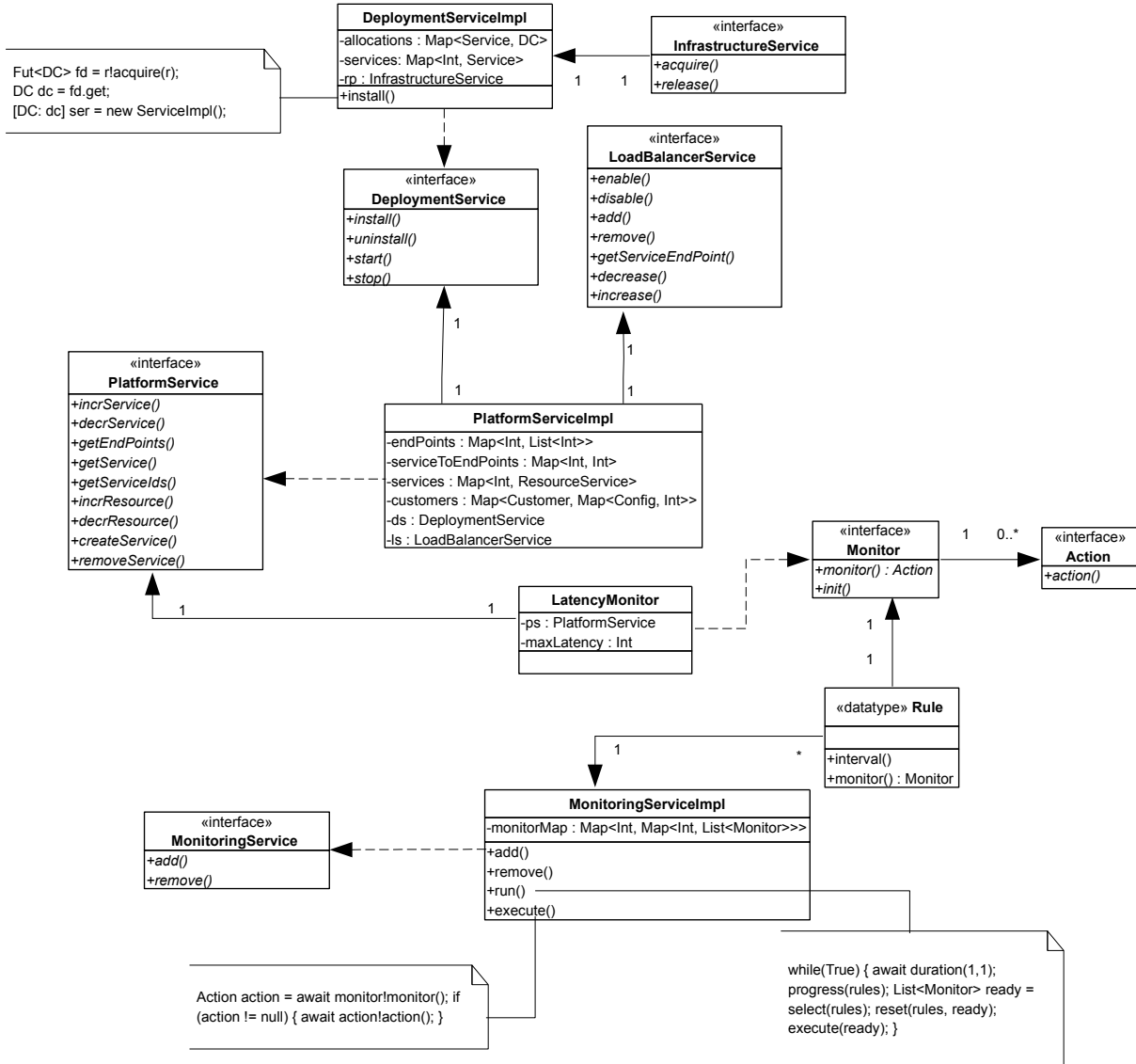


Figure 2.11: UML class diagram of the Fredhopper Cloud Services (2)

ABS models virtualized resources as a **DeploymentComponent**. A **DeploymentComponent** takes a value of the data type **Map<ResourceType, Rat>** as the resource specification. The abbreviation **DC** is a type synonym for **DeploymentComponent**.

The interface **InfrastructureService** is responsible for providing/managing virtual machines in the form of **DeploymentComponents**. This interface is implemented by the class **InfrastructureServiceImpl** shown in Figure 2.13. The figure shows the class's implementation of **acquire(Id, VMType)**. The method takes as input an **id** of the virtual machine to be acquired and its type, and returns an instance of the

```

interface InfrastructureService {
    DeploymentComponent acquire(Id id, VMType vmType);
    Unit release(DeploymentComponent component);
}

interface DeploymentService {
    Unit install(Customer c, ServiceType st, Id serviceId, VMType v);
    Unit uninstall(Id serviceId);
    Unit start(Id serviceId);
    Unit stop(Id serviceId);
}

interface LoadBalancerService {
    Bool enable(Id endPointId);
    Bool disable(Id endPointId);
    Bool add(List<Service> services, Id endPointId);
    Bool remove(Id endPointId);
    Maybe<EndPoint> getServiceEndPoint(Id endPointId);
    Bool decrease(Id endPointId, List<Service> services);
    Bool increase(Id endPointId, List<Service> services);
}

interface PlatformService {
    Unit incrService(Id endPoint, List< Map<Resourcetype, Rat> > instances);
    Unit decrService(Id endPoint, List<Id> serviceIds);
    List<Id> getEndPoints();
    Maybe<Service> getService(Id serviceId);
    List<Id> getServiceIds(Id endPoint);
    Unit alterResource(Id serviceId, Map<Resourcetype, Rat> r);
    Id createService(Config config, Customer customer);
    Unit removeService(Id endPoint);
}

```

Figure 2.12: ABS interface of the Fredhopper Cloud Services (2)

specified kind of machine. The method `acquire` either creates a new `DeploymentComponent` of the specified type, or reuses an existing `DeploymentComponent` if the id already exists. The capacity of the resources associated to the virtual machine are retrieved through the `Map<Resourcetypes, Rat> vmResources` given in Figure 2.4.

The interface `LoadBalancerService` in Figure 2.12 is responsible for binding requests to service endpoints to their constituent service instances. `DeploymentService` is responsible for allocating virtualized resources to service instances. This is implemented by the class `DeploymentServiceImpl`. Figure 2.14 shows the implementation of method `DeploymentServiceImpl.install(Customer, ServiceType, Id, VMType)`. This method instantiates a service of the specified type on a machine of the given type. The allocation of the virtualized resources to the new service instance is realized by the statement `[DC: vm] Service service = new ServiceImpl(serviceId, st, customer, 2);`, which indicates that the new service instance executes on the virtual machine `vm`.

`PlatformService` provides the interface to *Cloud Engineers* to add and to remove services. The interface also provides operations to the Monitoring and Alerting Service for adding and removing service instances to an endpoint and for adding and removing resources from a service instance. `PlatformService` is implemented by class `PlatformServiceImpl`, whose definition of method `Int createService(Config, Customer)` is shown in Figure 2.15. The method `createService` takes a service configuration and a customer identifier, deploys a corresponding service for that customer and returns the identifier of the service endpoint. Figure 2.25 shows how the services in the Fredhopper Cloud Services interact to deploy a service. Specifically,

```

class InfrastructureServiceImpl implements InfrastructureService {
  ...
  Map<Id, DeploymentComponent> inUse = EmptyMap;
  DC acquire(Id id, VMType vmType) {
    DC vm = null;
    Map<ResourceType, Rat> resourceConfig = vmResources(vmType);
    Maybe<DC> md = lookup(inUse, id);
    case md {
      Nothing => {
        //Allocate new instance of type vmType
        vm = new DeploymentComponent(intToString(id), resourceConfig);
        inUse = InsertAssoc(Pair(id, vm), inUse);
      }
      Just(d) => {
        //Use existing instance with the specified id
        vm = d;
      }
    }
    return vm;
  }
}

```

Figure 2.13: Definition of InfrastructureServiceImpl.acquire(Id, VMType)

```

class DeploymentServiceImpl(InfrastructureService rp) implements DeploymentService {
  Map<Service, DC> allocations = EmptyMap;
  Map<Id, Service> services = EmptyMap;

  Service install(Customer customer, ServiceType st, Id serviceId, VMType v) {
    assert lookup(services, serviceId) == Nothing;

    //acquire resource
    DC vm = await rp!acquire(serviceId, v);

    //instantiate service on vm
    [DC: vm] Service service = new ServiceImpl(serviceId, st, customer, 2);

    //update maps with resources (allocations) and service instances (services)
    allocations = InsertAssoc(Pair(service, dc), allocations);
    services = InsertAssoc(Pair(serviceId, service), services);
    return service;
  }
}

```

Figure 2.14: Definition of DeploymentServiceImpl.install

the method first iteratively creates the specified number of service instances, allocating each with a virtual machine that satisfies the resource requirements. Figure 2.25 shows the following:

1. the PlatformService interacts with the DeploymentService to install (install(Customer c, ServiceType st, Id serviceId, VMType v)) and start service instances (start(Id));
2. the DeploymentService interacts with the InfrastructureService to allocate (acquire(Id id, VMType vmType)) the required resources (DeploymentComponent) to the service instances;

3. after all service instances are deployed, the **PlatformService** interacts with the **LoadBalancerService** to bind (`add(List<Service>, Id)`) the instances to its service endpoint and to enable the endpoint (`enable(Id)`).

Figure 2.11 shows how the model of the Fredhopper Cloud Services respects the above dependencies. Specifically, **PlatformService** depends on **DeploymentService** and **LoadBalancerService** via the implementation **PlatformServiceImpl**, while **DeploymentService** depends on **InfrastructureService** via the implementation **DeploymentServiceImpl**. Dependencies are provided via dependency injection (i.e., passing the object that provides the service to the object that depends on it).

```

class PlatformServiceImpl(DeploymentService ds, LoadBalancerService ls) .. {
  Map<Id, ResourceService> services = EmptyMap;
  Map<Id, Id> serToEndPoint = EmptyMap; Map<Id, List<Id>> endPoints = EmptyMap;
  Map<Customer, Map<Config, Id>> customers = EmptyMap; Id serviceId = init();

  Id createService(Config config, Customer customer) {
    //this customer cannot already have the same service deployed
    ServiceType st = serviceType(config);
    assert lookupCustomerService(customers, customer, st) == Nothing;

    List<Int> instances = instances(config);
    //number of instances must be positive
    assert instances != Nil;

    //endpoint id
    Int endPoint = serviceId + 1;

    //create service instances
    List<Service> currentServices = Nil;
    List<Id> ids = Nil;

    while (instances != Nil) {
      Int res = head(instances);
      Service service = this.createServiceInstance(customer, st, res);
      Fut<Id> idf = service!getServiceId();
      Id id = idf.get;
      ids = Cons(id, ids);
      serviceToEndPoints = InsertAssoc(Pair(id, endPoint), serviceToEndPoints);
      currentServices = Cons(service, currentServices);
      instances = tail(instances);
    }

    //associate endpoint with service instances
    endPoints = InsertAssoc(Pair(endPoint, ids), endPoints);

    //update customer record
    customers = put(customers, customer,
      put(lookupDefault(customers, customer, EmptyMap), config, endPoint));

    //add services to load balancer
    await ls!add(currentServices, endPoint);

    //enable service
    await ls!enable(endPoint);

    return endPoint;
  }
}

```

Figure 2.15: Definition of PlatformServiceImpl.createService()

2.2.5 Monitoring

The static structure diagram of the Fredhopper Cloud Services shown in Figure 2.11 includes the Monitoring and Alerting Service. This service is modeled by the interface `MonitoringService`, which is implemented by the class `MonitoringServiceImpl` in ABS. The class `MonitoringServiceImpl`, shown in Figure 2.16,

has a `run` method that iteratively checks which monitors in the list of *scheduled* monitors (`monitorMap`) are ready in every clock cycle (`await duration(1, 1)`).

```
class MonitoringServiceImpl implements MonitoringService {
  Map<Int, Map<Int, List<Monitor>>> monitorMap = EmptyMap;
  ..
  Unit run() {
    while (True) {
      await duration(1, 1); // advance the clock
      this.monitorMap = decr(this.monitorMap); //decrement
      List<Monitor> toBeRun = lookupAllSecond(this.monitorMap, 0); //find all to be run
      this.monitorMap = reset(this.monitorMap); //reset
      //execute monitors
      while (toBeRun != Nil) {
        this!execute(head(toBeRun));
        toBeRun = tail(toBeRun);
      }
    }
  }

  Unit execute(Monitor m) {
    Action a = await m!monitor();
    if (a != null) {
      await a!action();
    }
  }
  ..
}
```

Figure 2.16: Definition of `MonitoringServiceImpl.run()`

The list of scheduled monitors are recorded as a two level map, where the first level key records the number of clock cycles between each execution of the lists of `Monitors` in the second level map, and the second level key records the number of remaining clock cycles until the next execution.

Given a `Monitor` `m`, the method invocation `m!monitor()` returns a possibly null `Action` object `a`. The returned object is null if no further action is required. Otherwise, the corresponding method `a!action()` executes the specified action.

Figures 2.26 – 2.28 depict the interactions between services to scale up the underlying resources of service instances suffering from high latency.

Figure 2.26 shows a sequence diagram of the `MonitoringServiceImpl` invoking a `Monitor` object to check the average latency of requests being served by all service instances (`monitor()`). The diagram shows that the `Monitor` collects latency reading (`Service.getLatency()`) from all service instances (`PlatformService.getServiceIds(Int)`) of all end points (`PlatformService.getEndpoints()`). If the latency of one or more service instances is too high, the `Monitor` object returns an `Action` for scaling up the virtualized resources underlying the service instances.

Figure 2.27 shows a sequence diagram of scaling up the virtualized resources (CPU unit per clock cycle) of a service instance. The diagram shows that said instance must be first removed from the load balancer (`LoadBalancerService.decrease(Int, List<Service>)`), and uninstalled (`uninstall(Int)`) from the existing resource via the `DeploymentService`. Note that the said service instance must no longer be serving requests before its removal from the load balancer. The instance is then installed onto a new virtual machine with the specified resource requirements and then added back to the load balancer (`LoadBalancerService.increase(Int, List<Service>)`).

Figure 2.28 shows a sequence diagram of scaling up the virtualized resources of a service instance when it is the only instance to its service endpoint. In order to keep the service running, before removing the

instance from the load balancer, a new service instance must first be installed onto the required resource and added to the load balancer. The old instance may then be removed and uninstalled.

2.3 Resource consumption and deployment

To offer services with a high QoS level, it is crucial to find an optimal deployment configuration: the number and kind of virtual machines used in a deployment must be sufficiently powerful and the cost of the virtual machines must be maintained at an acceptable level. The deployment configuration must also take into account several requirements. For example, some services should be co-located with other services: deploying an instance of the Query Service to a machine requires the presence of the Deployment Service on that same machine. Other service instances should be deployed on different machines, for example to increase fault tolerance (a resource failure will then not affect both service instances) or for security or business reasons, such as allocating a dedicated (per-customer) machine to services that manipulate sensitive private customer data (Figure 2.17).

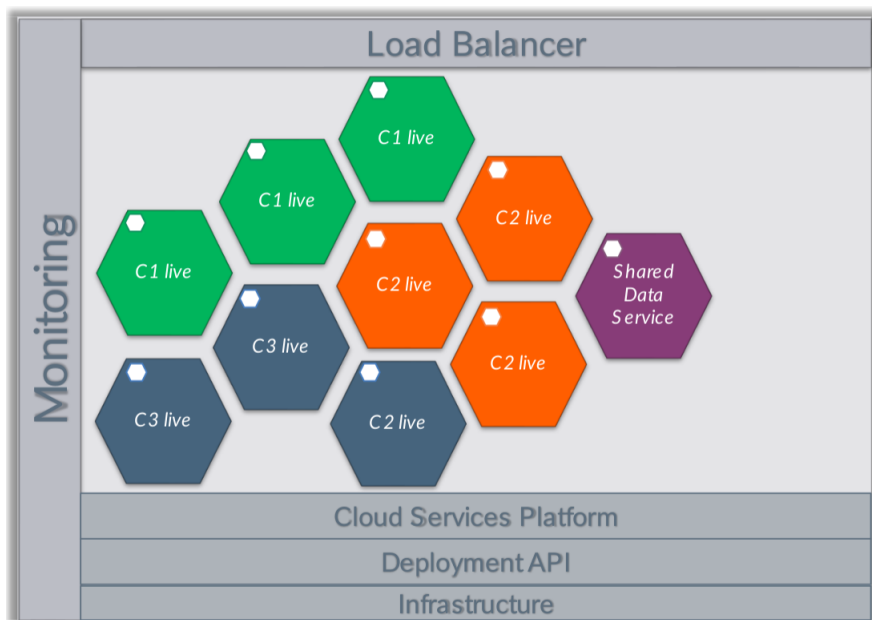


Figure 2.17: High-level view of a deployment

Finding an optimal deployment configuration that satisfies all requirements is a complex task that is currently done manually by an operations team. This requires domain-specific knowledge and is prone to human-error. Furthermore, the operations team takes conservative precautions to ensure customer quality, by overspending on the deployment configuration. In D1.3.1, and in a recently accepted publication [3], a tool-supported (“MODDE”) rigorous formal approach was developed that helps evaluating and automatically synthesizes better deployment configurations in an early phase (statically, in the design phase). Since the deployment configuration resulting from MODDE is known at an early stage, reserved instances of the virtual machines can be used, which are typically cheaper than on-demand instances. We show here how we have used these techniques to declaratively model deployment in the Fredhopper Cloud Services, using the following ingredients.

- Annotations on ABS classes that specify:
 - (a) the amount of resource consumed by instances of the class, and
 - (b) dependencies to other classes.

- A high-level declarative specification that captures requirements that should be satisfied of the desired deployment.
- The number of available virtual machines of each kind. The types of virtual machines are provided in a JSON file along the lines of Figure 2.2. An example JSON file showing the number of available instances of each type is given in Figure 2.18.

```
{
  "m1.large": 3,
  "m1.xlarge": 3,
  "c3.medium": 10,
  "c3.large": 5,
  "c3.2xlarge": 3
}
```

Figure 2.18: JSON file with the number of available instances of each type

Cost annotations specify the amount of resources that the annotated identity consumes, such as memory consumption, bandwidth, or CPU cycles. We have annotated the ABS model of the Fredhopper Cloud Services with cost annotations. The costs used in the annotations were extracted based on real-world log files of the in-production system. The resource consumption varies over time, and scenarios can be used to capture peaks and lows in the corresponding resource consumption. Figure 2.19 is a graph representation of one of these log files showing the number of Query requests per second and the roundtrip time (the total time of the HTTP request performing a Query Request as reported by the HTTP client handler). Figure 2.20 shows the size of the Query response over time (for the same customer, on the same date).

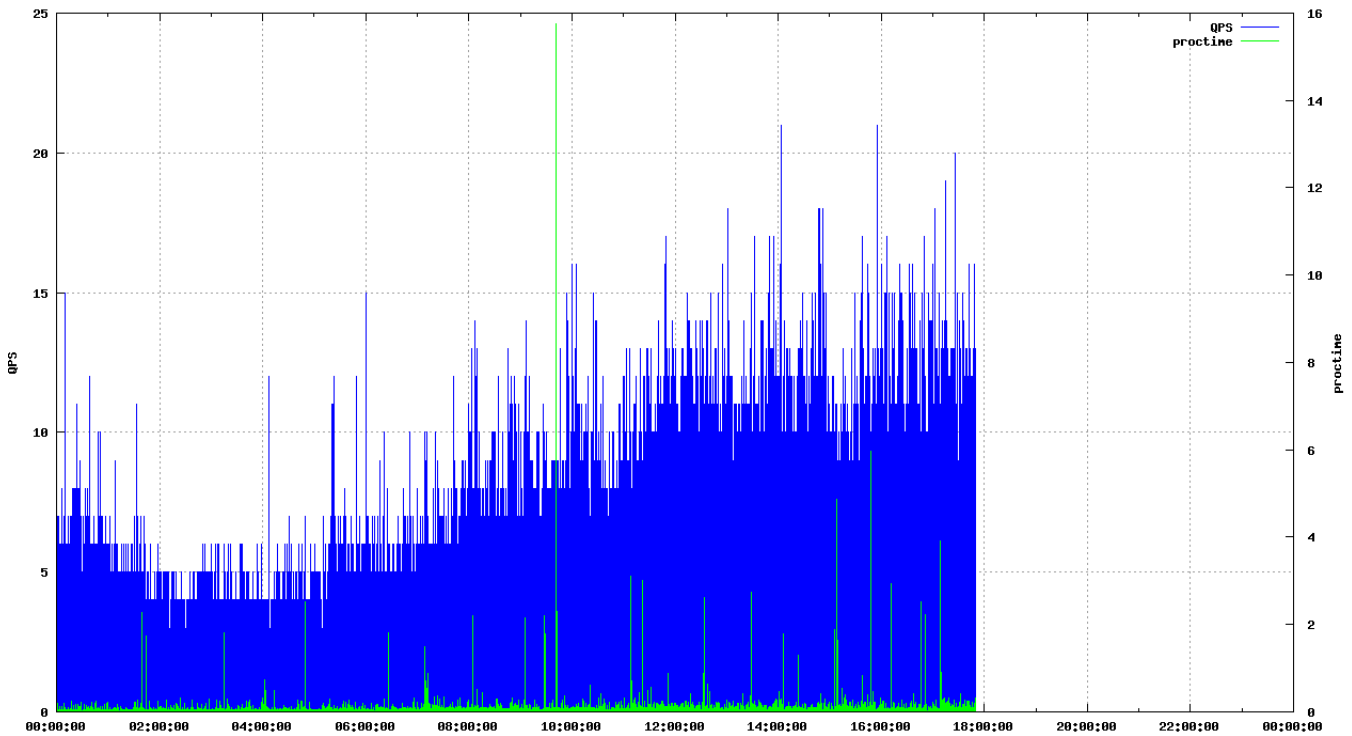


Figure 2.19: The number of queries per second (QPS) and roundtrip time (proctime) in seconds over time

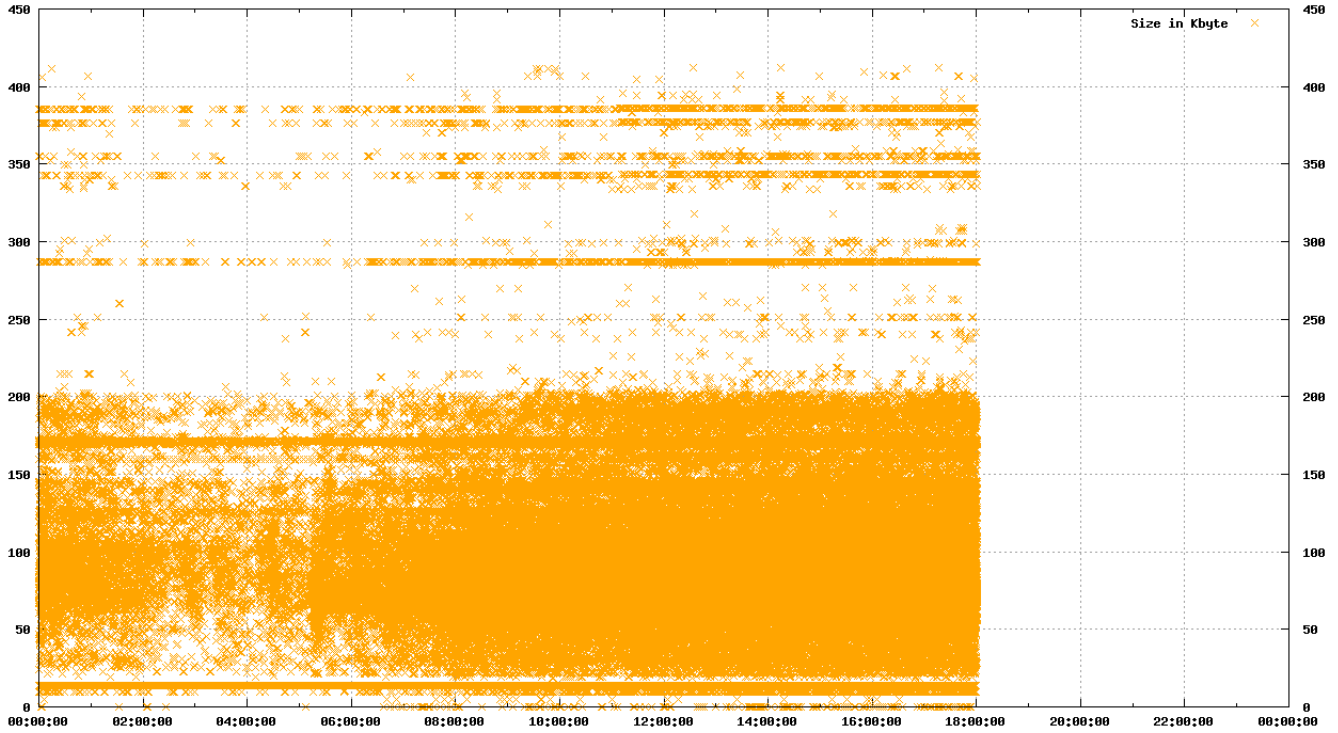


Figure 2.20: The response size (in kiB) of the Query Response over time

Figures 2.21 and 2.22 show annotations for the Query and Deployment services. The ABS annotation `Param("c", User)` indicates that the automated deployer leaves the parameter unspecified, and the user is expected to manually enter the right parameter instantiation. The annotation `Param("ds", Req)` means that the parameter is required to be defined by the automated deployer (by first creating an appropriate object of the desired type, and passing that as a parameter). Different scenarios (such as “DefaultUsage” and “HeavyUsage”) can be defined to capture variations in deployment requirements (for example, for different customers), and variations in resource consumption.

```
[Deploy: scenario[Name("DefaultUsage"), MaxUse(1), Cost("CPU", 1), Cost("Memory", 3000),
    Param("c", User), Param("ds", Req)] ]
[Deploy: scenario[Name("HeavyUsage"), MaxUse(1), Cost("CPU", 2), Cost("Memory", 4500),
    Param("c", User), Param("ds", Req)] ]
class QueryServiceImpl(DeploymentService ds, Customer c) implements Service
```

Figure 2.21: Cost annotation of Query service

```
[Deploy: scenario[MaxUse(2), Cost("CPU", 1), Cost("Memory", 800),
    Param("rp", Req)]]
class DeploymentServiceImpl(InfrastructureService rp) implements DeploymentService}
```

Figure 2.22: Cost annotation of the Deployment service

When a system deployment is automatically computed, a user expects to reach specific goals and could have some desiderata. For instance, in the considered Fredhopper Cloud Services use case, the goal is to deploy a given number of Query Services and a Platform Service, possibly located on different machines (e.g.,

to improve fault tolerance).

These goals and requirements are expressed in a *Declarative Deployment Language*: a language for stating the constraints that the final configuration should satisfy. For instance, the following constraint states that at least two `QueryService` instances and exactly one object of class `PlatformServiceImpl` should be deployed.

```
INTERFACE[IQueryService] >= 2 and CLASS[PlatformServiceImpl] = 1
```

More complex quantities involve constraints on the virtual machines used in deploying. For example, we can specify that no virtual machine with less than two CPUs should contain more than one object of class `QueryServiceImpl` as follows.

```
DC[ CPU <= 2 | CLASS[QueryServiceImpl] >= 2 ] = 0
```

Using such constraints it is also possible to express co-location or distribution requirements. For instance, for efficiency reasons it could be convenient to co-locate highly interacting objects or, for security or fault tolerance reasons, two objects should be required to be deployed separately. We can require that an object of class `QueryServiceImpl` must be always co-installed together with an object of class `DeploymentServiceImpl` with the following constraint.

```
DC[CLASS[QueryServiceImpl] > 0 and CLASS[DeploymentServiceImpl] = 0 ] = 0
```

Based on the above deployment requirements, the ABS class annotations specifying resource consumption and inter-class dependencies, and the available virtual machines in the JSON file, MODDE automatically synthesizes a deployment configuration with the least cost. Figure 2.23 shows the output of MODDE when 2 instances of the Query service are required for a customer.

```
DeploymentComponent m1_large_1 =
  new DeploymentComponent("m1.large_1", vmResouces(M1_LARGE));
DeploymentComponent m1_large_2 =
  new DeploymentComponent("m1.large_2", vmResouces(M1_LARGE));
DeploymentComponent m1_xlarge_1 =
  new DeploymentComponent("m1.xlarge_1", vmResouces(M1_LARGE));
DeploymentComponent m1_xlarge_2 =
  new DeploymentComponent("m1.xlarge_2", vmResouces(M1_LARGE));
DeploymentComponent amazon_internals =
  new DeploymentComponent("amazon_internals", map[]);

[DC: amazon_internals] InfrastructureService
  o1 = new InfrastructureServiceImpl();
[DC: m1.xlarge_1] LoadBalancerService o2 = new LoadBalancerServiceImpl();
[DC: m1.large_1] DeploymentService o3 = new DeploymentServiceImpl(o1);
[DC: m1.large_2] DeploymentService o4 = new DeploymentServiceImpl(o1);
[DC: m1.xlarge_2] MonitorPlatformService
  o5 = new PlatformServiceImpl(list[o3,o4], o2);
[DC: m1.large_2] IQueryService o6 = new QueryServiceImpl(o4, CustomerX);
[DC: m1.large_1] IQueryService o7 = new QueryServiceImpl(o3, CustomerX);
[DC: m1.xlarge_2] ServiceProvider o8 = new ServiceProviderImpl(o5, o2);
```

Figure 2.23: Initial optimal deployment configuration by MODDE

A graphical representation of the distribution of objects over the resources corresponding to this deployment configuration is shown in Figure 2.24. Virtual machines are depicted as boxes containing the objects, and an edge from an object *a* to an object *b* represents the use of *b* as a parameter for the creation of *a*.

The deployment configuration suggested by MODDE differs from the one used in-production, which uses only instances of type `c3.xlarge` (one for the Platform Service and the Service Provider, one for the Load Balancer, two for the two Query and Deployment Service pairs). This discrepancy is due to the fact that we allowed MODDE to use all the possible AWS instances. Currently, the machines of type `m1` have

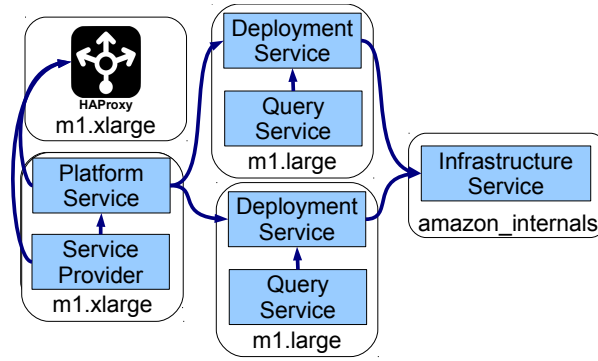


Figure 2.24: Deployment Configuration synthesized by MODDE

been deprecated and new **m1** machines could not be acquired any more. The optimal solution computed by MODDE can therefore be only used by customers that have already **m1** running machines. New customers have to rely instead on machines of type **m3** and **c3**.

If MODDE is executed taking into account just the new **m3** and **c3** AWS instances, the computed configuration obtained is exactly the one currently adopted by the operations team.

2.4 Summary

Code Metric	Value
Lines of code	1410
Functions	30
Classes	13
Interfaces	15
Data types and type synonyms	8

Table 2.1: Statistics

In this chapter we presented the resource-aware modeling of the Fredhopper Cloud Services and showed how we modeled deployment in a declarative manner. Table 2.1 shows some code metrics of the resource-aware ABS model of the Fredhopper Cloud Services.

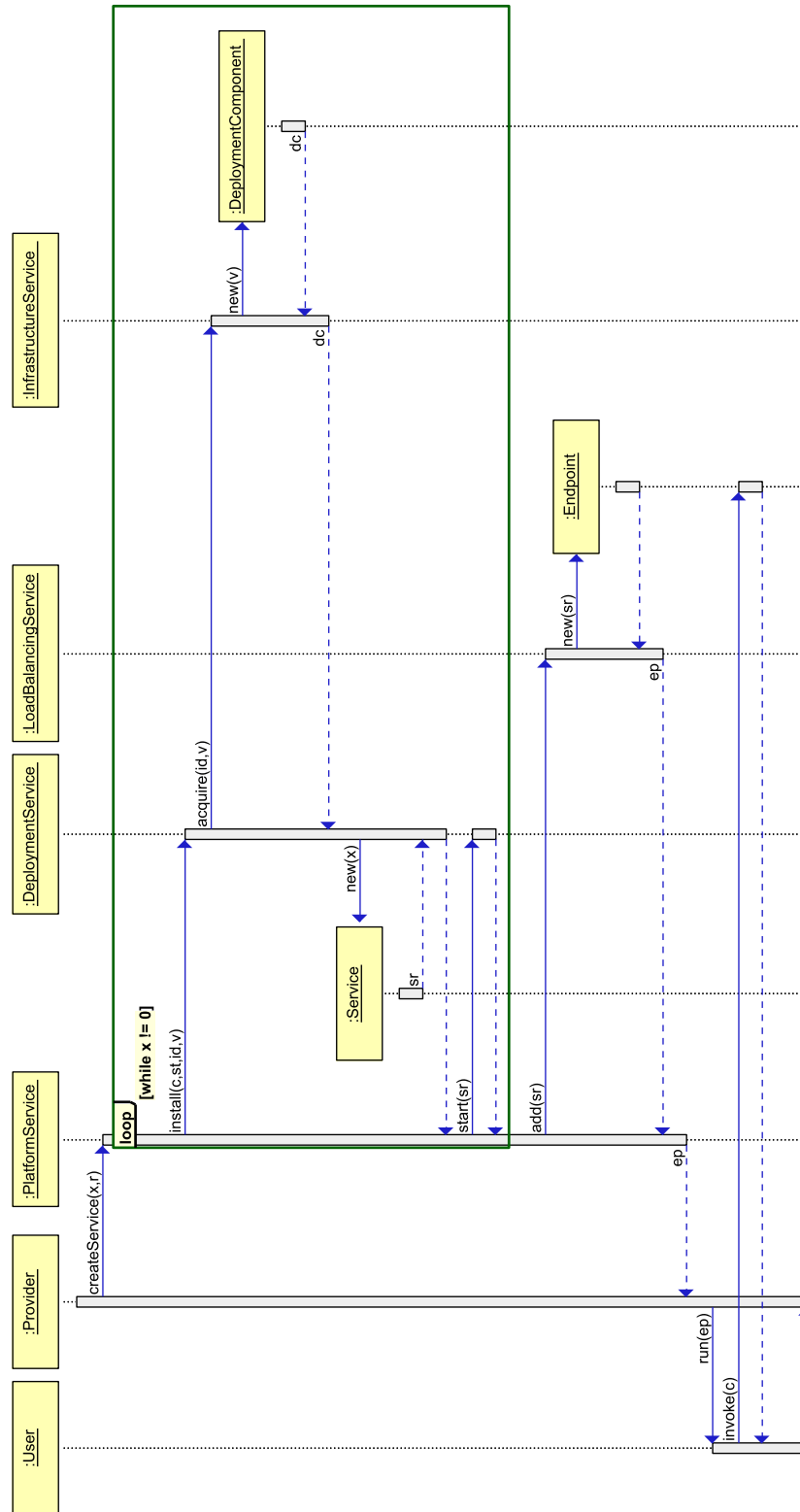


Figure 2.25: UML sequence diagram of creating a service using the Fredhopper Cloud Services

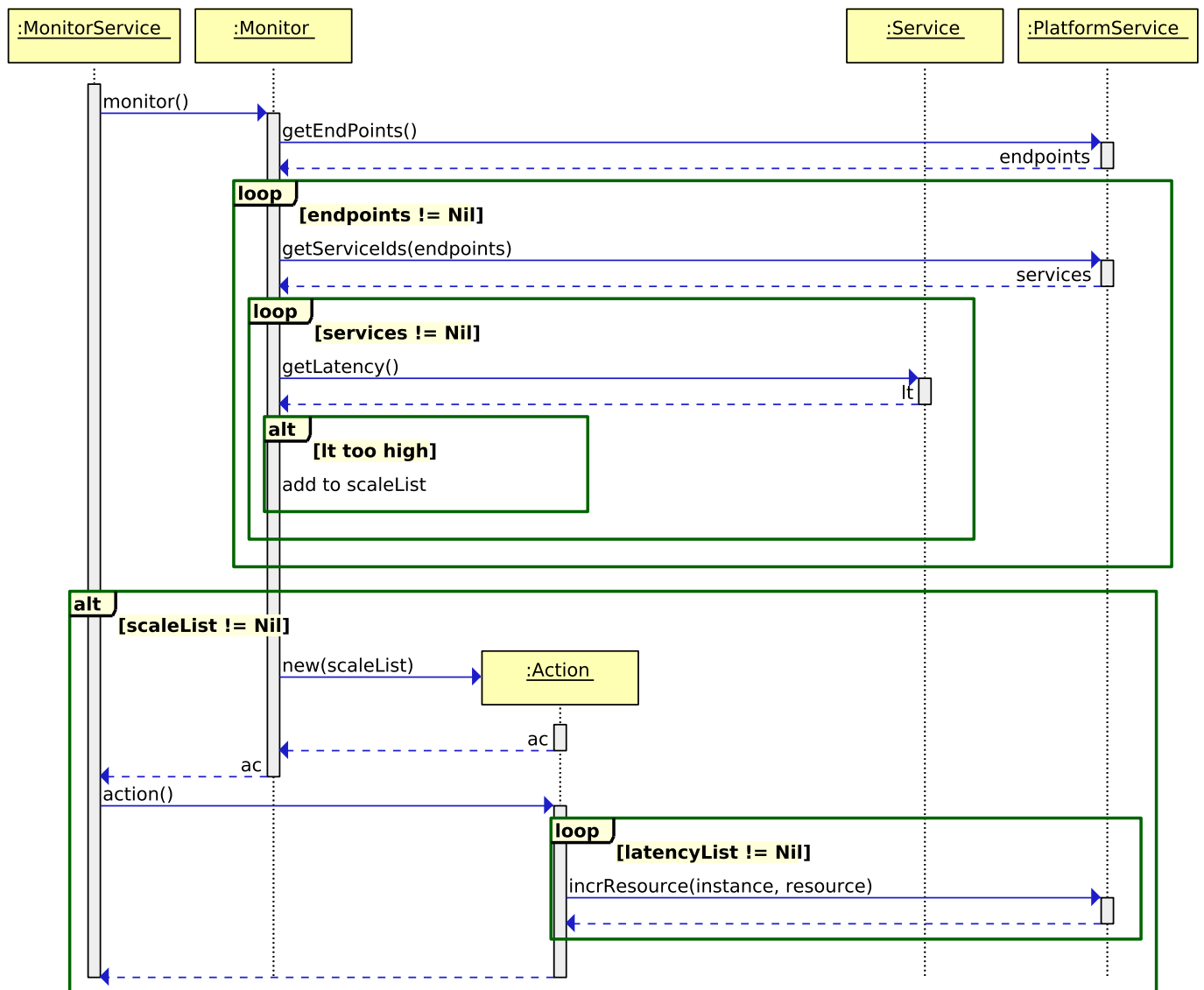


Figure 2.26: UML sequence diagram of monitoring service latency

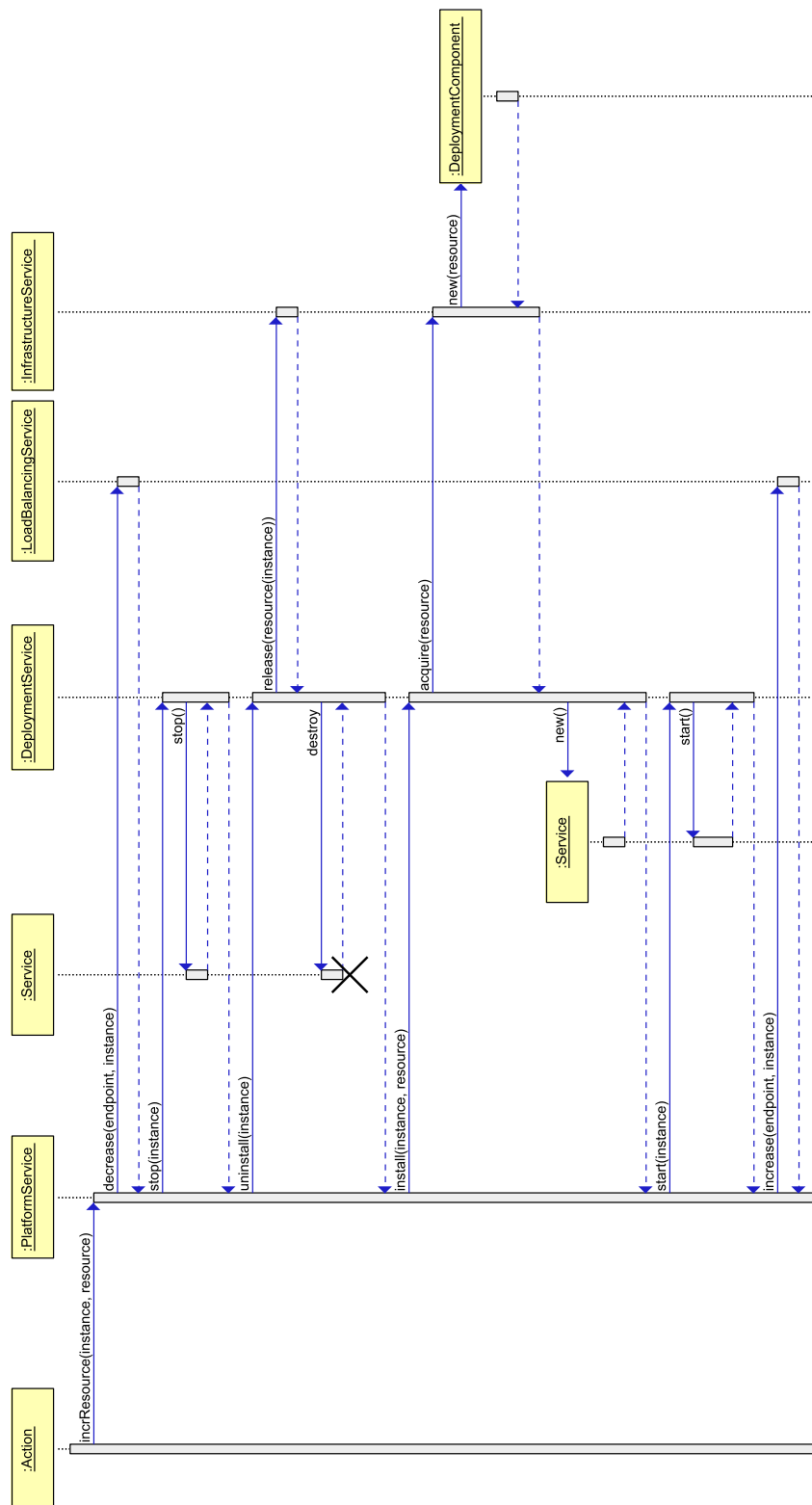


Figure 2.27: UML sequence diagram of scaling a service with more than one service instance

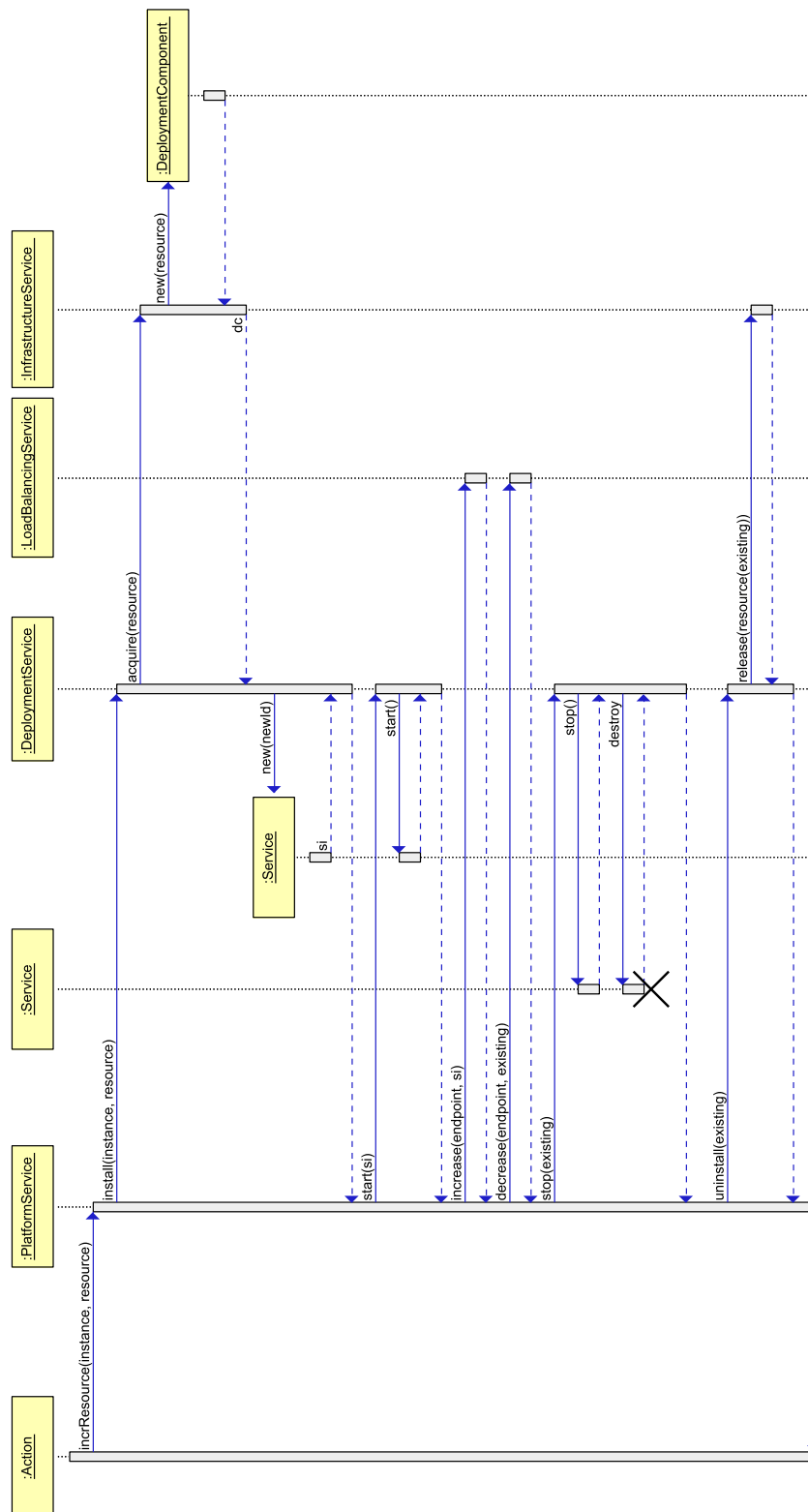


Figure 2.28: UML sequence diagram of scaling a service with only one service instance

Chapter 3

Experience and feedback to technical tasks

We successfully applied the **Envisage** framework to obtain a resource-aware model and capture deployment scenarios of the Fredhopper Cloud Services. Based on our experiences, in this chapter we give detailed feedback to various ongoing technical tasks pointing out desired extensions and refinements to increase accuracy and effectiveness. In particular, we provide input for the final reports of T1.2, T1.3, T1.4, T2.2 and T2.3.

Task T1.2: Modeling of Resources A virtual machine (VM) is modeled as a **DeploymentComponent**, and consists of several resource types: CPU, Bandwidth and Memory. Each resource type in a VM has a certain “capacity”, given by a (single) number. Modeling resources and deployment at design-time allowed for early analysis comparing different deployments, in particular to identify trade-offs between the QoS-level and the cost of the system. We suggest the following enhancements to further increase the accuracy of the resource modeling.

- A fundamental additional resource type not currently supported is *storage*. The speed (rather than the storage capacity) of the storage device has been a bottleneck in the in-production system at FRH, and has affected deployment decisions, in the sense that I/O-optimized machines are used for certain kinds of services.
- The accuracy of the modeling of the resource types can be enhanced by supporting multiple metrics as the capacity of the resource. More specifically, for memory, in addition to the total size of the memory, its speed can be considered. For the CPU resource type, the number of cores could be distinguished from the speed of the cores. At the moment, the capacity of each resource type present is given by a *single* number, forcing the user to choose between memory capacity *or* memory speed (and similarly for the other resource types).
- In practice, virtual machines and resources can fail (more on this below, in T1.3), and machines can be terminated. This should not change only the state of the machines (or **DeploymentComponents**), it also affects the objects running on these machines: in reality, these stop executing / fail. To capture this behavior accurately, the ABS semantics for objects should take failures of the underlying virtual machine on which it runs into account.

Task T1.3: Modeling of Deployment The declarative language for modeling deployment was intuitive to use and proved to be sufficiently expressive to capture the deployment scenarios of the in-production Fredhopper Cloud Services. We identify several desired extensions below.

- Virtual machines can fail, for instance due to an operating system kernel error, or a hardware failure, but currently, failures are not taken into account. Support for modeling such failures opens the door for reasoning about failures (both static *and* dynamic analyses, through monitoring and simulation), such as how failures *propagate*, and design of fault-tolerant systems.

- Virtual machines can be started, stopped, fail, and so on: they have a well-defined *state*. Throughout its lifetime, the instance moves from state to state. When an instance is started, it enters an initialization state. After it initializes successfully, it enters a “running” state. Instance states can also be triggered involuntarily, due to failures.

Adding states to virtual machines, and making it possible to retrieve this state for a specified machine, is important for load balancing, monitoring, cost management (i.e. charge only for machines in running state), etc. and allows implementing services with increased robustness. Figure 3.1 shows a possible virtual machine lifecycle.

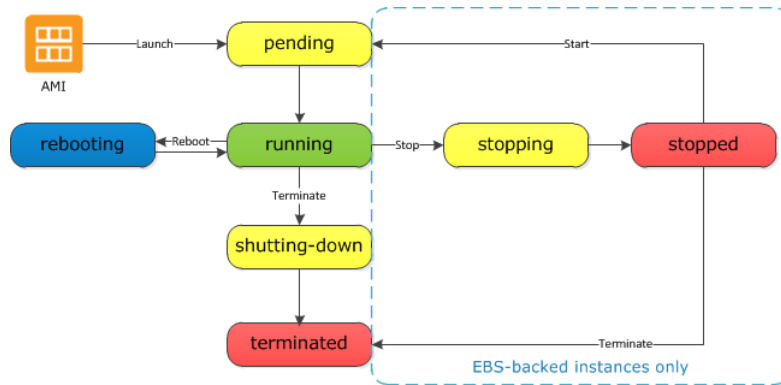


Figure 3.1: Instance lifecycle. Source: amazon.com

- The expressiveness of the cost annotations that the Model-Driven Deployment Engine (MODDE) (see D1.3.1, Chapter 2) supports could be enhanced: currently costs are constants. In practice, the exact cost is not necessarily fixed over time, and can depend on certain parameters, for instance: execution of a query by the FRH query service typically increases the larger the product catalogue is.

We worked around this problem by using a constant upper-bound observed from actual customer logs, and using scenarios to capture behavior varying over time. However, extracting a fixed constant upper-bound for all “reasonable” inputs may be difficult in certain cases, and lead to over-provisioning.

- The Cloud API (D1.3.1, Chapter 3) currently does not take into account which kind of virtual machines are offered by the infrastructure provider: it is possible to acquire a machine with *any* resource capacity, include resource configurations not offered by any existing machine (for example, a machine with one million CPU’s and 5 Bytes of memory). This could be addressed by making the API parametric with respect to the available virtual machine types (i.e. using the ADT in Figure 2.3 and the Map from Figure 2.4).

Task T1.4: Simulation The Maude back-end supports a powerful in-depth inspection of the system state for simulation and debugging purposes. Furthermore it is easily extensible for prototyping experimental features: simply add a new rewrite rule. We suggest to investigate the following idea’s.

- Simulation to observe the effect on the system while, for example, varying the number of received requests can be done by modeling (various kinds of) “the user” in ABS code: the user triggers the system by invoking requests to an end-point at a certain rate. Nonetheless, modeling the user in this manner is cumbersome. A “log-replay” tool that fires queries to the system according to a specified time and duration given in a log file would be a very useful addition for simulation purposes.
- An important enhancement to the Eclipse plug-in is support for navigation, such as “browse to declaration”, “display type hierarchy” and “display call hierarchy”.

- The Maude back-end supports inspection of the complete state of the system, which allows exploring detailed run-time information. However, this state can be unwieldy; sometimes a more abstract version of the information is already sufficient and allows faster identification of relevant data. In particular, support for visualizing the object graph, visualization of resource usage over time, and a visualization of the trace of messages between distributed COGs would be useful.

Task T2.2: Service Contracts and SLAs Behavioral interfaces (D2.2.1, Chapter 3) naturally capture properties of the behavior of a *single* instance of a given type, such as the response time guarantee example. Support for defining *aggregated* properties that involve multiple objects (or computation / communication traces) executing on different (distributed) virtualized resources would be useful. This would allow to capture properties of statistics - aggregations of a metric - such as “Total Number of Fredhopper Query Requests”, or the “Maximum response size” (in a given time window).

Task T2.3: Monitoring Add-ons

- Service metric functions, or statistics, aggregate a sequence of basic measurements (of a certain metric) to allow determining QoS levels. The question arises how such statistics can be defined in a systematic manner. A possible option to investigate is using attributes defined in an attribute grammar [6] for this purpose. Informally, an attribute is a function that assigns an aggregated value to a list of symbols (in our context, basic measurements). The definition of attributes can exploit structure present when different kinds of measurements should be aggregated. Attribute grammars were previously already integrated into ABS in the context of run-time checking [2].
- The system is initially executed in a declaratively specified deployment configuration using the techniques developed in T1.3. The monitors generated in T2.3 adapt the deployment configuration dynamically, for example due to usage peaks of a Service. Thus it is interesting to investigate the relationship between the monitoring service and possible dynamic re-deployment actions. In particular, do the monitors ensure that the evolved system preserves (a certain subset of) the deployment requirements specified initially? Could a monitor be generated automatically that checks at run-time whether the current deployment configuration respects the requirements?

Chapter 4

Summary

This deliverable reports on the detailed modeling of the different deployment scenarios of the FRH case study in the abstract behavioral specification language. Based on this application of the **Envisage** techniques, we give feedback to the technical tasks. In particular, we provide input for D1.2.2, D1.3.2, D1.4.2, D2.2.2 and D2.3.2.

Bibliography

- [1] Elvira Albert, Frank S. de Boer, Reiner Hähnle, Einar Broch Johnsen, Rudolf Schlatte, Silvia Lizeth Tapia Tarifa, and Peter Y. H. Wong. Formal modeling of resource management for cloud architectures: An industrial case study using Real-Time ABS. *Journal of Service-Oriented Computing and Applications*, 8(4):323–339, 2014.
- [2] Frank S. de Boer, Stijn de Gouw, and Peter Y. H. Wong. Run-time verification of coboxes. In *Software Engineering and Formal Methods - 11th International Conference, SEFM 2013, Madrid, Spain, September 25-27, 2013. Proceedings*, pages 259–273, 2013.
- [3] Stijn de Gouw, Michael Lienhardt, Jacopo Mauro, Behrooz Nobakht, and Gianluigi Zavattaro. On the integration of automatic deployment into the ABS modeling language. In *Service-Oriented and Cloud Computing - Second European Conference, ES OCC 2015*, 2015. To appear.
- [4] Initial User Requirements, January 2014. Deliverable D4.1 of project FP7-610582 (ENVISAGE), available at <http://www.envisage-project.eu>.
- [5] Einar Broch Johnsen, Reiner Hähnle, Jan Schäfer, Rudolf Schlatte, and Martin Steffen. ABS: A core language for abstract behavioral specification. In Bernhard Aichernig, Frank S. de Boer, and Marcello M. Bonsangue, editors, *Proc. 9th International Symposium on Formal Methods for Components and Objects (FMCO 2010)*, volume 6957 of *Lecture Notes in Computer Science*, pages 142–164. Springer-Verlag, 2011.
- [6] Donald E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145, 1968.

Glossary

Terms and Abbreviations

Cloud Engineer A *Cloud Engineer* handles the day-to-day operation of the Fredhopper Cloud Services. She deploys/updates services through PaaS and IaaS according to incomplete *service requirements* from *Consultants*, diagnoses issues at service-level and either resolves them at real time or informs the *Support Engineers* and/or the *Software Engineers*. She manages the up and down scaling of service resources according to alerts and metric visualizations provided by the monitoring system. She also performs any necessary infrastructural changes to the Fredhopper Cloud Services

Consultant A *Consultant* manages the technical setting that enables *Customer* to use the APIs offered by the Fredhopper Cloud Services. She provides *service requirements* to *Cloud Engineers*

Customer A *Customer* is a business entity that powers her online shop using the APIs provided by the Fredhopper Cloud Services

Faceted Navigation Faceted navigation is a technique for accessing information organized according to a faceted classification system, allowing users to explore a collection of information by applying multiple filters. Facets correspond to properties of the information elements

Fredhopper Cloud Services A set of services managed by FRH through cloud computing that allows the offering of search and targeting facilities on a large product database to e-Commerce companies

Full Text Search In text retrieval, full-text search refers to techniques for searching a single computer-stored document or a collection in a full text database

IaaS Infrastructure as a Service

Infrastructure as a Service A provision model in which an organisation outsources the equipment used to support IT operations, including storage, hardware, servers and networking components. The service provider owns the equipment and is responsible for housing, running and maintaining it. The client typically pays on a per-use basis

JSON JavaScript Object Notation. A data format that uses human-readable text to transmit data objects consisting of attribute–value pairs.

PaaS Platform as a Service

Platform as a Service A category of cloud service offerings that facilitates the deployment of applications without the cost and complexity of buying and managing the underlying hardware and software and provisioning hosting capabilities

QoS Quality of Service

Quality of Service Generic term encapsulating all the non-functional aspects of a service delivery

Resource Configuration A description of the number of service instances initially required for a service offered to a *Customer* and the virtualized resource to be allocated initially to those service instances

SaaS Software as a Service

Service Level Agreement A legal contract between a service provider and his customer. It records a common understanding about services, priorities, responsibilities, guarantees, and warranties

Service Requirement A service requirement consists of the agreed SLA and the *Customer's* specific configuration such as expected query throughput based on historical data in terms of monthly and peak page views

SLA Service Level Agreement

Software as a Service A software delivery model in which software and associated data are centrally hosted on the cloud. SaaS is typically accessed by users using a thin client via a web browser

Software Engineer A *Software Engineer* develops and maintains the Fredhopper Cloud Services. She provides technical support to *Cloud Engineers* and *Support Engineers*. She fixes bugs on the Fredhopper Cloud Services and continuously improves the Fredhopper Cloud Services by either adding new features or improving existing ones

Support Engineer A *Support Engineer* receives and coordinates issues identified either by *Customers* or *Cloud Engineers*. She receives questions from *Customer*. She interacts with *Customer*, and either addresses them directly or informs the *Software Engineers*