



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

Deliverable D4.3.1

Initial Modeling of the FRH Case Study

Date of document: T10



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:

Initial Modeling of the FRH Case Study

This document summarizes deliverable D4.3.1 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 initial modeling of the structural and functional aspects of the FRH case study, and details how the case study plans to cover the objectives O1-O5 of **Envisage**. This deliverable forms a part of the verification of **Envisage** project milestone M1.

List of Authors

Stijn de Gouw (FRH)
Peter Y. H. Wong (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	Service Configuration	7
2.2.2	Service Endpoints and Instances	7
2.2.3	Service Architecture	10
2.2.4	Monitoring	14
2.3	Summary	15
3	Planning and Summary	20
3.1	Planning	20
3.1.1	Objective O1: Foundations of Computation with Virtualized Resources	20
3.1.2	Objective O2: Behavioral Specification Language for Virtualized Resources	20
3.1.3	Objective O3: Design-by-Contract Methodology for Service Contracts	20
3.1.4	Objective O4: Model Conformance Demonstrator	21
3.1.5	Objective O5: Model Analysis Demonstrator	21
3.1.6	Objective O6: Demonstration of Impact	21
3.2	Summary	21
	Bibliography	22
	Glossary	23

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 this deliverable D4.3.1, we study the structural and the functional aspects of the Fredhopper Cloud Services and provide an initial model in the Abstract Behavioural Specification language (ABS) [1, 3]. This model forms the basis for the follow-up tasks within **Envisage** on resource modeling, formal specification and monitor generation. We discuss the initial model of the Fredhopper Cloud Services in Chapter 2. We highlight how the FRH case study covers the objectives of **Envisage** and summarize this deliverable in Chapter 3.

This deliverable forms a part of the verification of **Envisage** project milestone M1.

Chapter 2

Fredhopper Cloud Services

This chapter presents an initial model of the Fredhopper Cloud Services in the ABS language that models its structural and functional aspects¹.

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.

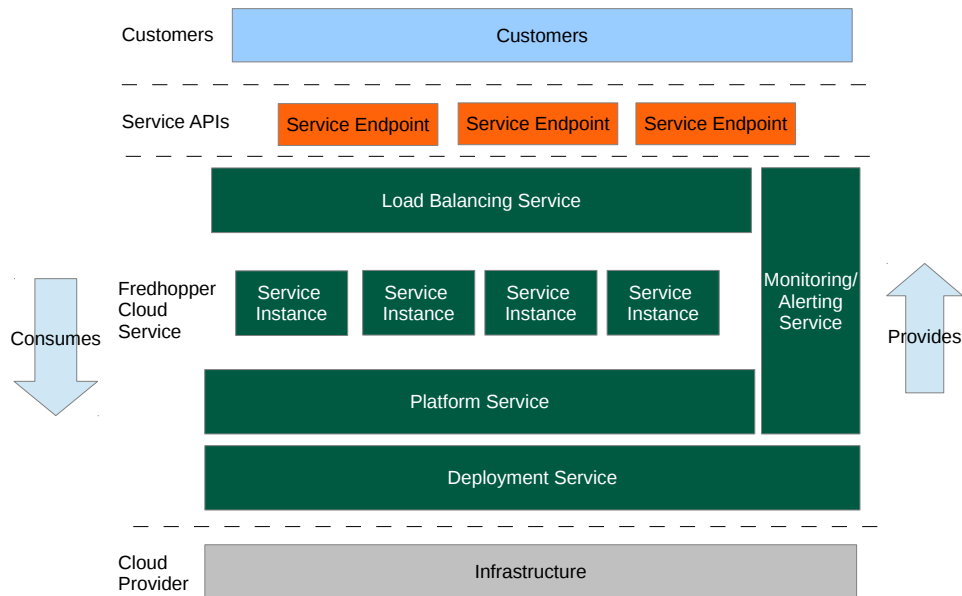


Figure 2.1: Block diagram of the Fredhopper Cloud Services

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,

¹The complete ABS model described in this chapter can be found at <https://envisage.ifi.uio.no:8080/redmine/projects/abstools/repository/revisions/master/raw/examples/T4.3/D4.3.1/FredhopperCloudServices.abs>.

which allows users to query over their product catalogue via full text search² and faceted navigation³. 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* scalable⁴. 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). HAProxy is a TCP/HTTP load balancer that also provides HTTP authentication.

2.1.4 Platform Service

The Platform Service provides an *interface* to the *Cloud Engineers* [2, 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 [2, 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/>).

²en.wikipedia.org/wiki/Full_text_search

³en.wikipedia.org/wiki/Faceted_navigation

⁴en.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 on which evaluation and experiments will be conducted. In this section we provide an overview of the initial version of the ABS model of the structural and the functional aspects of the Fredhopper Cloud Services.

2.2.1 Service Configuration

```
data ServiceType = ...;
data DCData = CPU(Int capacity) | InfCPU;
data Config = Config(ServiceType serviceType, List<DCData> instances);
```

Figure 2.2: Service specification

Figure 2.2 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**) and the number of service instances and its resource requirement (**List<DCData>**). 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 **nth DCData** value in the list denotes the amount of resources required (the number of CPU units per clock unit) for the **nth** instance. At the time of writing, ABS models resources as **DeploymentComponent**, where a **DeploymentComponent** takes a value of the data type **DCData** as the resource specification. The value **capacity(d)** on a *finite* **DCData** value **d** returns the number of CPU unit per clock unit [1].

2.2.2 Service Endpoints and Instances

Figure 2.3 shows the static structure of an endpoint and its implementation and Figure 2.4 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, and the latency of a request may be dependent on the following two factors:

- the amount of data over which a request queries; 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**.

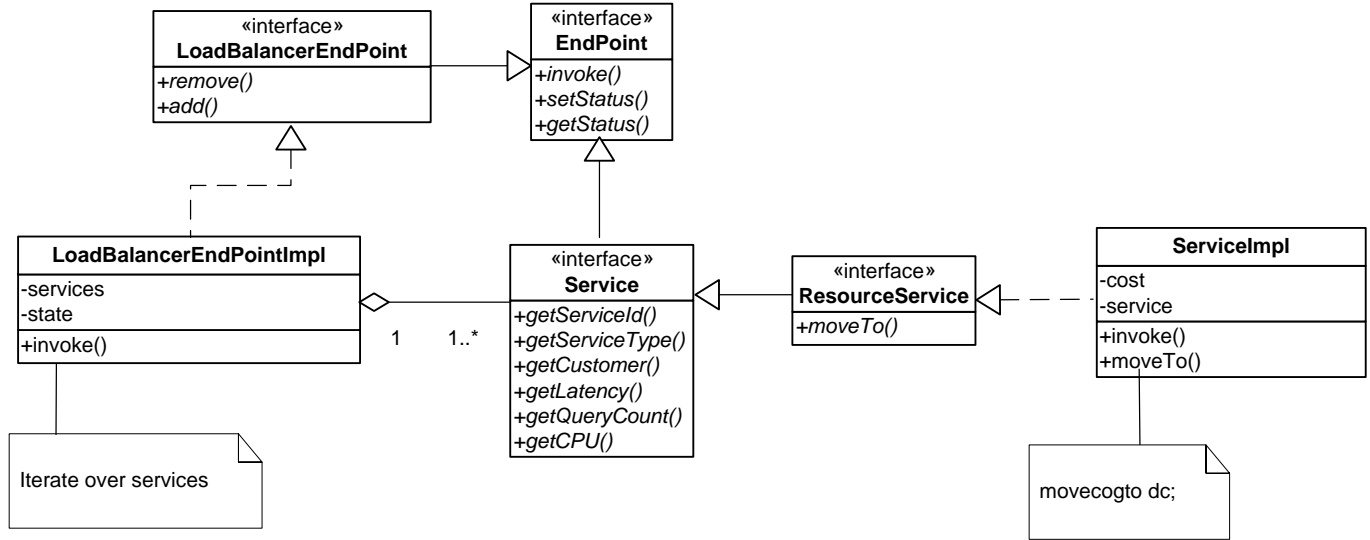


Figure 2.3: UML class diagram of the Fredhopper Cloud Services (1)

```

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();
  Int getCPU();
}

interface ResourceService extends Service {
  Unit moveTo(DeploymentComponent dc);
}

```

Figure 2.4: ABS interface 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. **Service** is further extended by **ResourceService** that exposes the ability of a service instance to be allocated with different resources. In ABS, there are two approaches to model re-allocation of resources:

1. execute `dc.incrementResources(n)` on `dc` where `dc` is the `DeploymentComponent` providing for the service instance to increase `dc`'s CPU unit per clock cycle by `n` unit
2. execute `movecogto dc` inside the concurrent object group of the service instance to move the service instance to the `DeploymentComponent dc`

In this case study, we adopt the second approach as it allows us to model how we might want to implement resource re-allocation in the production environment.

```

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 = currentms();
    [Cost: cost] this.log = this.log + 1;
    time = currentms() - time;
    this.latency = max(this.latency, time);
    return success();
  }
  ..
}

```

Figure 2.5: Implementation of `ServiceImpl.invoke(Int)`

Figure 2.5 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 that denotes the latency of a request when the size of the request is 1. We use the type synonym `Customer` to model the customer's name. The function `currentms()` 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 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 2.6: 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 dis-

tributed. The class `LoadBalancerEndPointImpl` implements `LoadBalancerEndPoint` and its implementation of `invoke(Request)` is shown in Figure 2.6. 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.

2.2.3 Service Architecture

Figure 2.7 shows the static structure of the Fredhopper Cloud Services, and Figure 2.8 shows the corresponding interfaces in ABS. We defer the presentation of the initial modeling of the Monitoring and Alerting Service to Section 2.2.4. The static structure shown in Figure 2.7 models dependencies between various services in the Fredhopper Cloud Services.

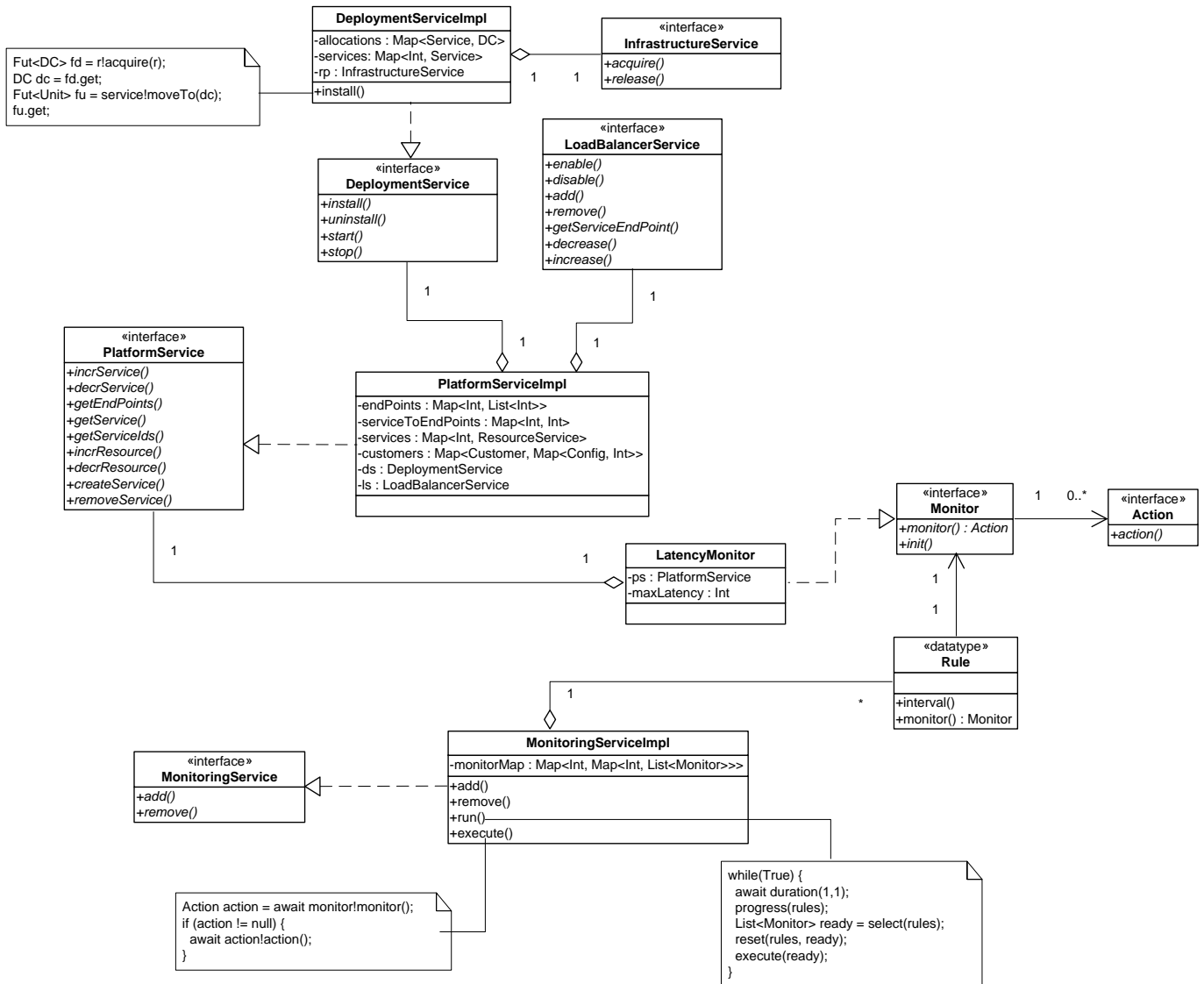


Figure 2.7: UML class diagram of the Fredhopper Cloud Services (2)

The interface `InfrastructureService` is responsible for providing/managing virtualized resources in the form of `DeploymentComponents`. This interface is implemented by the class `InfrastructureServiceImpl` shown in Figure 2.9. The figure shows the class's implementation of `acquire(Id, DCData)`. The method takes as input the `id` of the `DeploymentComponent` to be acquired and the required `amount` of resources from it. As mentioned earlier, currently virtualized resource in ABS is modeled as the number of CPU units per

```

interface InfrastructureService {
    DeploymentComponent acquire(Id id, DCData amount);
    Unit release(DeploymentComponent component);
}

interface DeploymentService {
    Unit install(ResourceService service, DCData res);
    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<DCData> 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, DCData r);
    Id createService(Config config, Customer customer);
    Unit removeService(Id endPoint);
}

```

Figure 2.8: ABS interface of the Fredhopper Cloud Services (2)

clock cycle. This method `acquire` either creates a new `DeploymentComponent` with the required resource, or reuses an existing `DeploymentComponent`. When reusing a `DeploymentComponent`, the method ensures it has the required number of CPU units per clock cycle.

The interface `LoadBalancerService` in Figure 2.8 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 2.10 shows the implementation of method `DeploymentServiceImpl.install(ResourceService, DCData)` that acquires the specified amount of virtualized resources via the `InfrastructureService` in the form of the `DeploymentComponent`. The allocation of `DeploymentComponent` `dc` to the service instance `service` is realized by the method invocation `service!moveTo(d)`.

`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 2.11. 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 2.13 shows how the services in the Fredhopper Cloud Services interact to deploy a service. Specifically, the method first iteratively creates the specified number of service instances, allocating each with the specified number of CPU units. Figure 2.13 shows the following:

```

class InfrastructureServiceImpl implements InfrastructureService {
  Int total = 0; Map<Id, DeploymentComponent> inUse = EmptyMap;
  DeploymentComponent acquire(Id id, DCData amount) {
    Maybe<DeploymentComponent> md = lookup(this.inUse, id);
    DeploymentComponent dc = null;
    case md {
      Nothing => {
        dc = new DeploymentComponent(intToString(id), amount);
        this.total = this.total + capacity(amount);
        this.inUse = InsertAssoc(Pair(id, dc), this.inUse); }
      Just(d) => {
        dc = d;
        Fut<DCData> cf = dc!available();
        DCData cpu = cf.get;
        Int cpu = capacity(cpu);
        Int amt = capacity(amount);
        if (cpu < amt) {
          Fut<Unit> inf = dc!incrementResources(amt - cpu); inf.get;
          this.total = this.total + amt;
        }}
    return dc;
  }
}

```

Figure 2.9: Definition of InfrastructureServiceImpl.acquire(Int, Int)

```

class DeploymentServiceImpl(InfrastructureService rp) implements DeploymentService {
  ..
  Unit install(ResourceService service, DCData res) {
    Id id = await service!getServiceId();
    DeploymentComponent dc = await rp!acquire(id, res);
    await service!moveTo(dc);
    ..
  }
  ..
}

```

Figure 2.10: Definition of DeploymentServiceImpl.install(ResourceService, Int)

1. the PlatformService interacts with the DeploymentService to install (install(ResourceService, DCData)) and start service instances (start(Id));
2. the DeploymentService interacts with the InfrastructureService to allocate (acquire(Id, DCData)) 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 2.7 shows how the initial 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.

```

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 f, Customer c) {
    ServiceType st = serviceType(f); List<DCData> instances = instances(f);

    //this customer cannot already have the same service deployed
    assert lookupCustomerService(this.customers, c, st) == Nothing;

    //number of instances must be positive
    assert instances != Nil;

    //endpoint id
    Id endPoint = incr(this.serviceId);

    //create service instances
    List<Service> currentServices = Nil; List<Id> ids = Nil;
    while (instances != Nil) {
      this.serviceId = incr(this.serviceId); ids = Cons(this.serviceId, ids);

      //base query costs at least 2 time unit
      ResourceService service = new ServiceImpl(this.serviceId, st, customer, 2);

      //register with deployment service with appropriate resource
      Fut<Unit> sf = this.ds!install(service, head(instances)); sf.get;

      //start instance
      Fut<Unit> uf = this.ds!start(this.serviceId); sf.get;

      //update record
      this.services = put(this.services, this.serviceId, service);
      this.serToEndPoint = InsertAssoc(Pair(this.serviceId, endPoint), this.serToEndPoint);
      currentServices = Cons(service, currentServices);
      instances = tail(instances);
    }

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

    //update customer record
    Map<Config, Id> existing = lookupDefault(this.customers, c, EmptyMap);
    this.customers = put(this.customers, c, put(existing, f, endPoint));

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

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

    return endPoint;
  }
}

```

Figure 2.11: Definition of PlatformServiceImpl.createService()

2.2.4 Monitoring

The static structure diagram of the Fredhopper Cloud Services shown in Figure 2.7 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 2.12, 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 2.12: 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 2.14 – 2.16 depict the interactions between services to scale up the underlying resources of service instances suffering from high latency.

Figure 2.14 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 2.15 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 resource **DeploymentComponent** with the specified CPU units per clock cycle (`install(ResourceService, DCData)`) and then added back to the load balancer (`LoadBalancerService.increase(Int, List<Service>)`).

Figure 2.16 shows a sequence diagram of scaling up the virtualized resources (CPU unit per clock cycle) 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 Summary

Properties	Values
Lines of code	1230
Functions	30
Classes	13
Interfaces	15
Data types and type synonyms	8

Table 2.1: Statistics

In this chapter we presented an initial model of the Fredhopper Cloud Services in the language of ABS that models its structural and functional aspects. Table 2.1 shows some statistics on the size of the initial model of the Fredhopper Cloud Services.

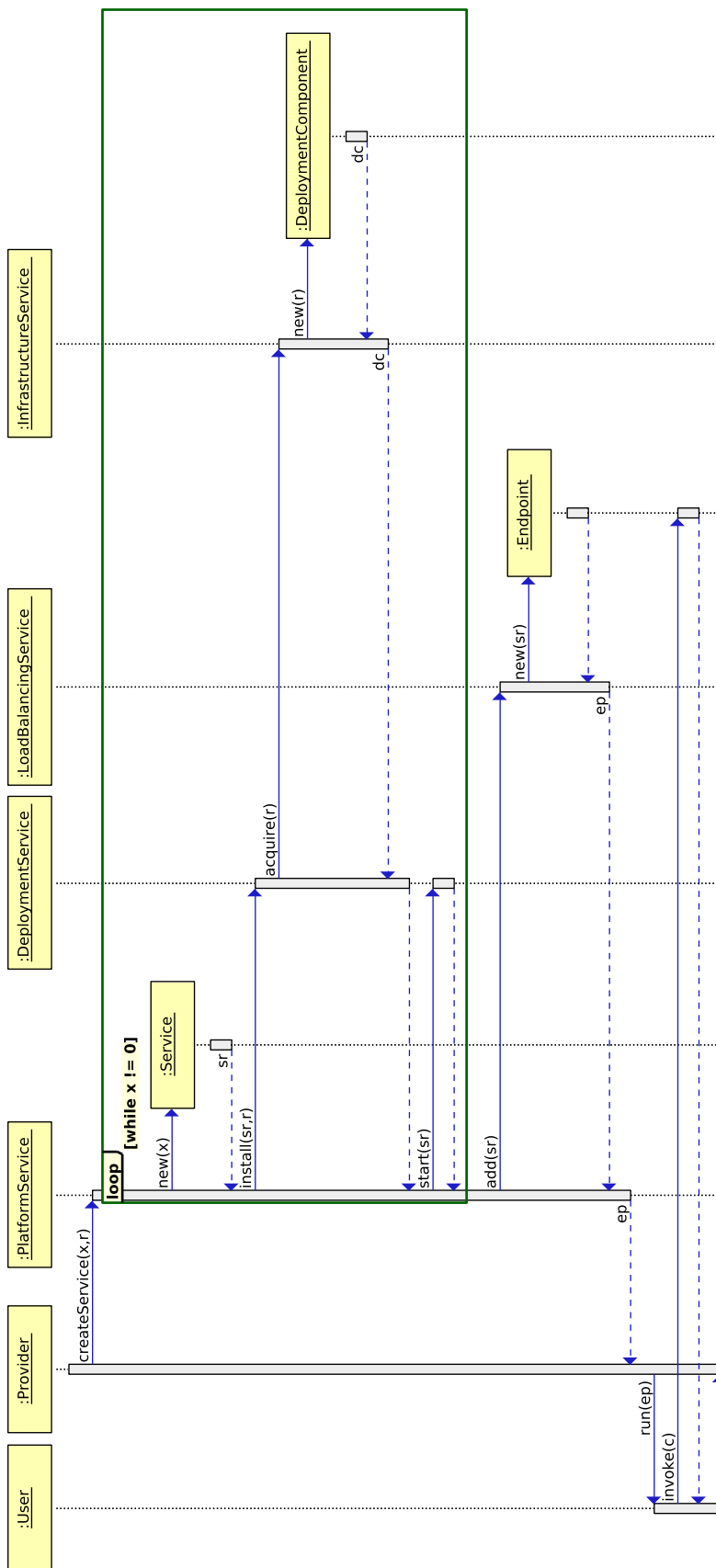


Figure 2.13: UML sequence diagram of creating a service using the Fredhopper Cloud Services

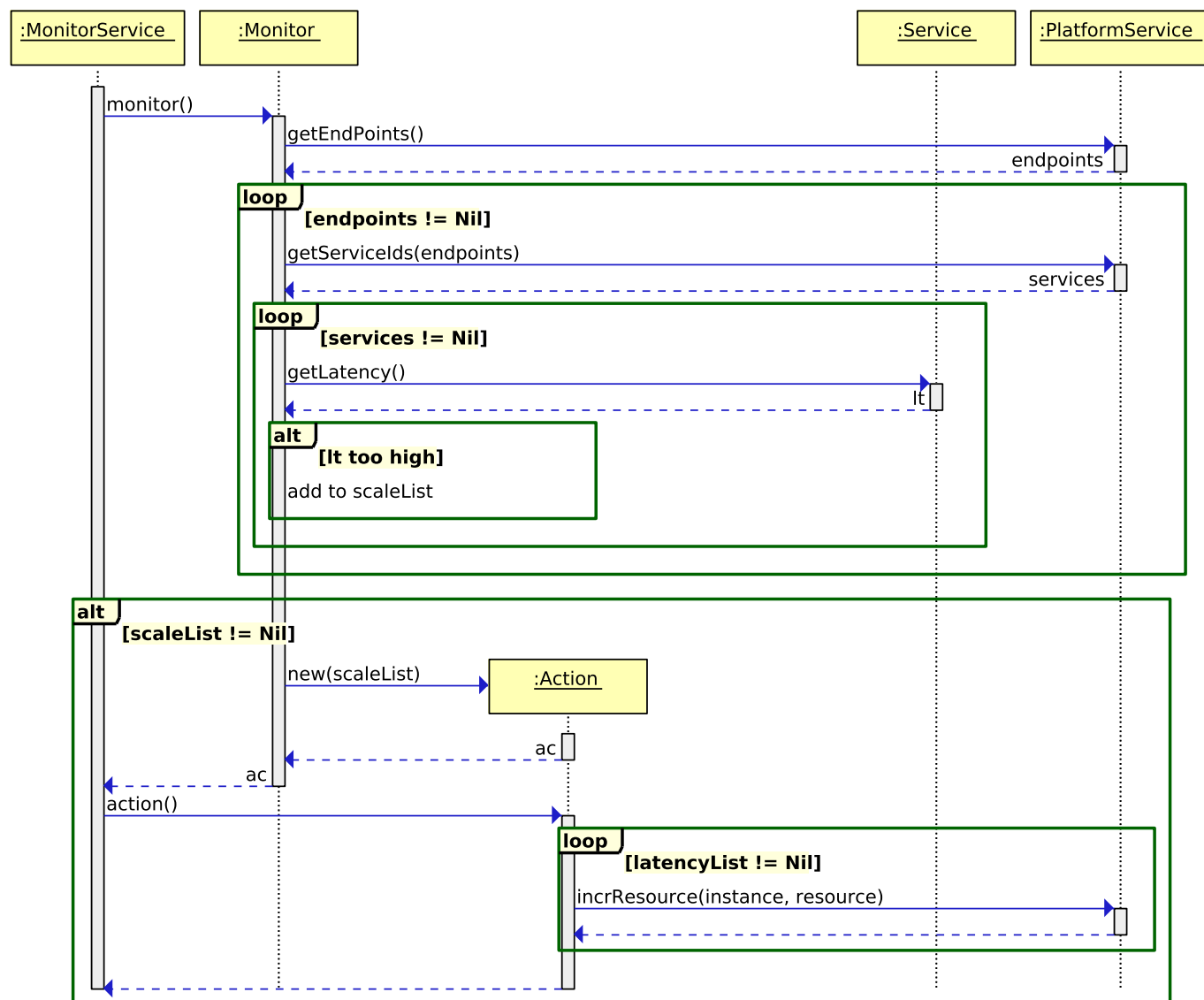


Figure 2.14: UML sequence diagram of monitoring service latency

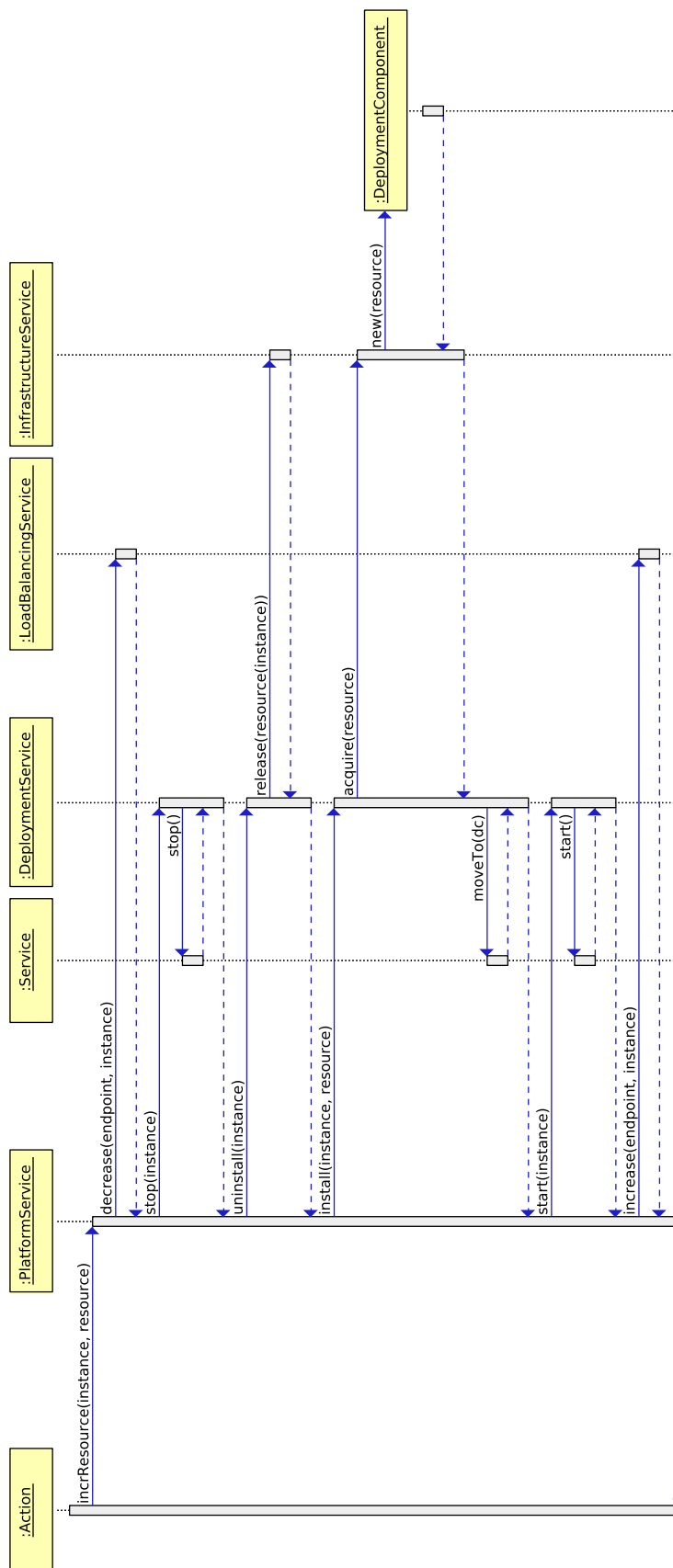


Figure 2.15: UML sequence diagram of scaling a service with more than one service instance

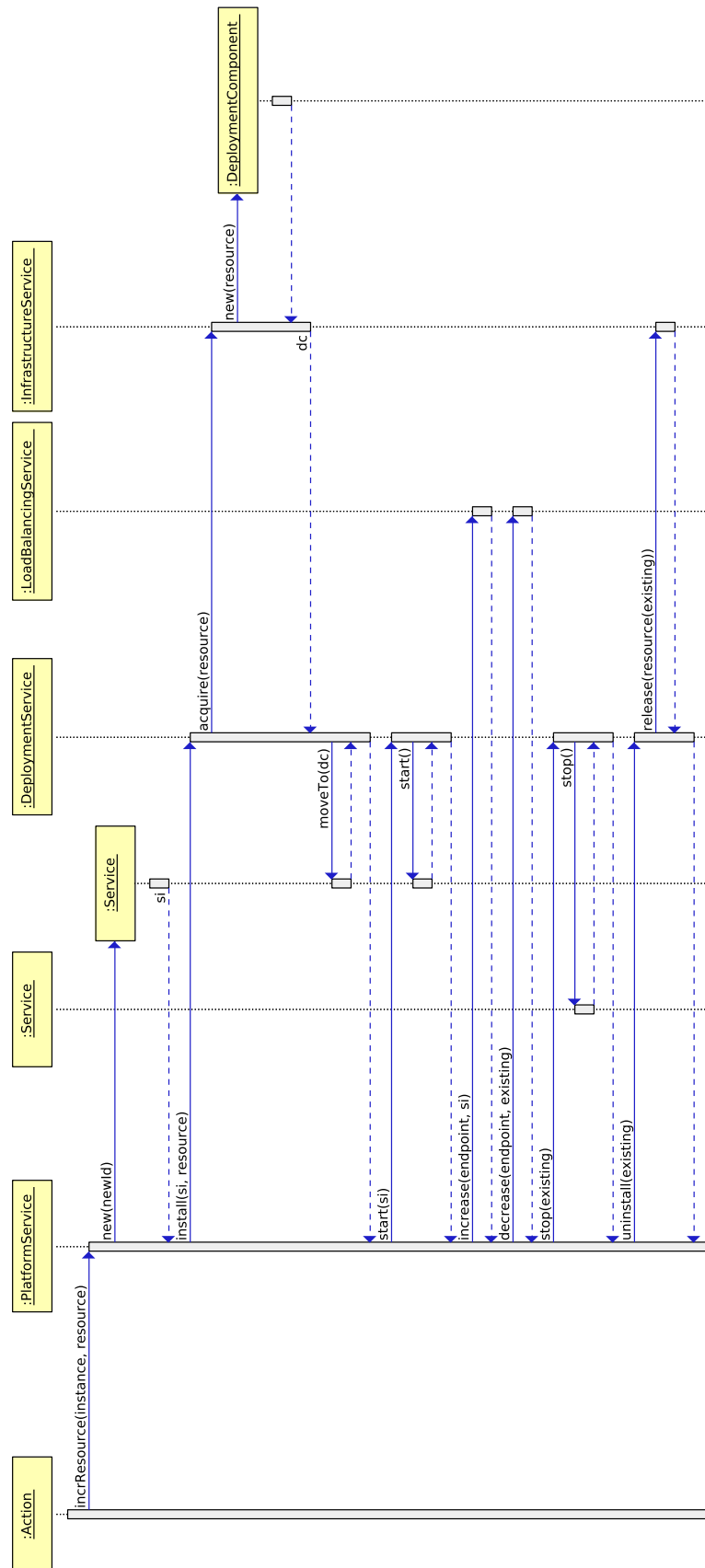


Figure 2.16: UML sequence diagram of scaling a service with only one service instance

Chapter 3

Planning and Summary

This chapter highlights how the FRH case study covers the objectives of *Envisage* and summarizes this deliverable.

3.1 Planning

Here we present the objectives according to the project's Description of Work and highlight how they may be covered by the FRH case study.

3.1.1 Objective O1: Foundations of Computation with Virtualized Resources

The outcome of this objective is a semantic framework for scalable architectures, infrastructures, and virtualized resources. The framework provides the means to model and to specify resource-related non-functional requirements that arise in the context of virtualized resources. This deliverable forms the verification of milestone M1. Specifically, using the ABS language, we were able to model the Fredhopper Cloud Services. This initial model focuses on the component structure of the Fredhopper Cloud Services and the functionalities of individual components, while abstracting away from the implementation of the services offered. We were also able to model (re-)allocation of virtualized resources to service instances, and automatic scaling of resources on instances via runtime monitoring.

3.1.2 Objective O2: Behavioral Specification Language for Virtualized Resources

The outcome of this objective is a resource-aware, abstract behavioral specification language and its prototype simulator. As this deliverable provides the initial model of the Fredhopper Cloud Services, we envisage that the model will be extended using that resource-aware, abstract behavioral specification language. We will conduct simulation exercises on difference deployment scenarios of services provided by the Fredhopper Cloud Services using the simulator from this objective's outcome. This will be part of deliverable D4.3.2.

3.1.3 Objective O3: Design-by-Contract Methodology for Service Contracts

The outcome of this objective is a formal specification language for service contracts that will include behavioral and QoS description, the definition of compliance of the service contract with the SLA, and a resource-aware generalization of design-by-contract methodology. Continuing from the initial model, we aim to first extend the model with resource information. Furthermore we will utilize the formal specification language from this objective to specify behavioral and QoS description of services as well as the Fredhopper Cloud Services. We aim to evaluate the practicality of the methodology delivered by this objective. This specification and evaluation work will be part of deliverable D4.3.2.

3.1.4 Objective O4: Model Conformance Demonstrator

The outcome of this objective is a demonstrator for the conformance of generated or legacy code to a given abstract model. Part of the FRH case study is to assess whether we could use the **Envisage** framework to automatically generate monitors for runtime checking and alerting. The requirements of this are that monitors are to be generated correctly with respect to the SLA of services managed by the Fredhopper Cloud Services [2, Requirements FRH-R003, FRH-R004]. In order to do this, we plan to first extend the initial model of the Fredhopper Cloud Services with resource-awareness, as well as behavioral and QoS specification of services according to the production Fredhopper Cloud Services. We will then use the extended model as a specification for generating runtime monitors and evaluate their correctness against the Fredhopper Cloud Services model. This exercise will be conducted as in Task 4.5 and will form part of deliverable D4.5.

3.1.5 Objective O5: Model Analysis Demonstrator

The main outcome of this objective is the runtime support for the resource analysis and validation with the SLA. As part of investigating FRH's user requirements [2, Requirements FRH-R002, FRH-R006, FRH-R008, FRH-R010], we plan to assess how the generated monitors (as part of Object O4) may be deployed on the production Fredhopper Cloud Services. Moreover, we plan to investigate how these monitors may be changed at runtime, in order to adapt changes to services, their load and their SLAs. This exercise will be conducted in Task 4.5 and will form part of deliverable D4.5.

3.1.6 Objective O6: Demonstration of Impact

The outcomes of this objective are case studies artifacts from Work Package 4 and their formal verification, the **Envisage** framework being made available as a service, and dissemination. The initial modeling of the Fredhopper Cloud Services provided in this deliverable, which is part of the verification of milestone M1, and the overall assessment of the **Envisage** framework in deliverable D4.5, which will be part of the verification of milestone M5, will be used to partly assess the outcome of this objective.

3.2 Summary

This deliverable presented the initial modeling of the structural and functional aspects of the FRH case study, and discussed how the case study plans to cover the objectives O1-O6 of **Envisage**.

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*, 2013.
- [2] Initial User Requirements, January 2014. Deliverable D4.1 of project FP7-610582 (ENVISAGE), available at <http://www.envisage-project.eu>.
- [3] 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.

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

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*