Modelling Auto-scalable Services Using Real Time ABS

http://abs-models.org/autoscalable-services-tutorial/

Einar Broch Johnsen
Silvia Lizeth Tapia Tarifa

http://www.envisage-project.eu/

Introduction

In this tutorial, we are going to explain through an example how to model and deploy an auto-scalable service. The example is implemented on Real Time ABS and is using the ABS Cloud API.

Contents

1 General Overview of the Example

Let us consider an example which models a service that executes heavy and parallel computations on the cloud. Figure 1 depicts the architecture of this service. In this architecture *clients*, which want to access the service communicates with the application manager. To take advantage of the cloud elasticity, the *application manager* includes a *resource management* with a load balancer and an autoscaler. Clients send request to a service endpoint. The *service endpoint* communicates with a load balancer to get workers that will process the jobs requested by the clients. Each *worker* is able to process one call of the service at a time, and for that it needs to access a shared *database*. The *load balancer* keeps a list of workers which are currently processing jobs and a list of workers which are available for new jobs, and distributes jobs among the workers. The *autoscaler* will increase or decrease the number of workers as needed.

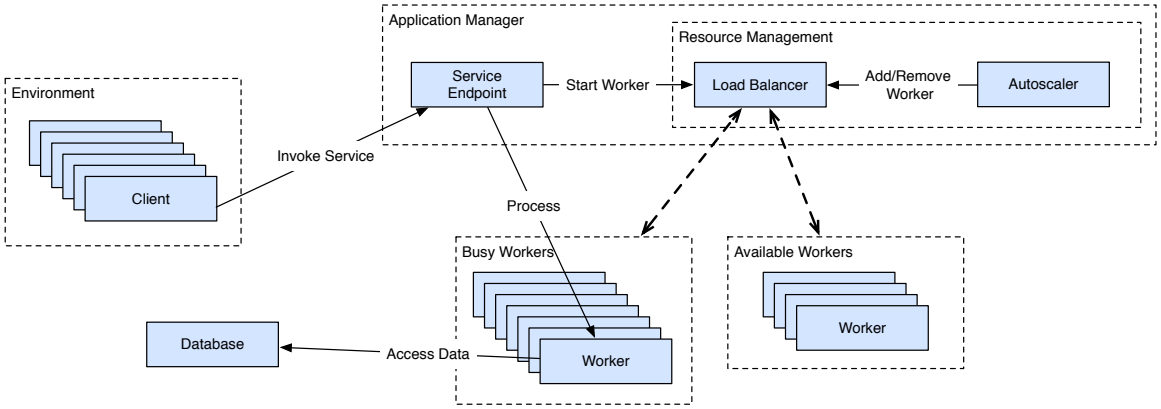

Figure 1: An architecture for an auto-scaling service

Let us now extend this architecture with its deployment on a cloud infrastructure. Figure 2 depicts this extension. For this particular scenario, we will deploy the application manager and a database in permanent servers outside the cloud, and each worker in their own dedicated virtual machines on the cloud.

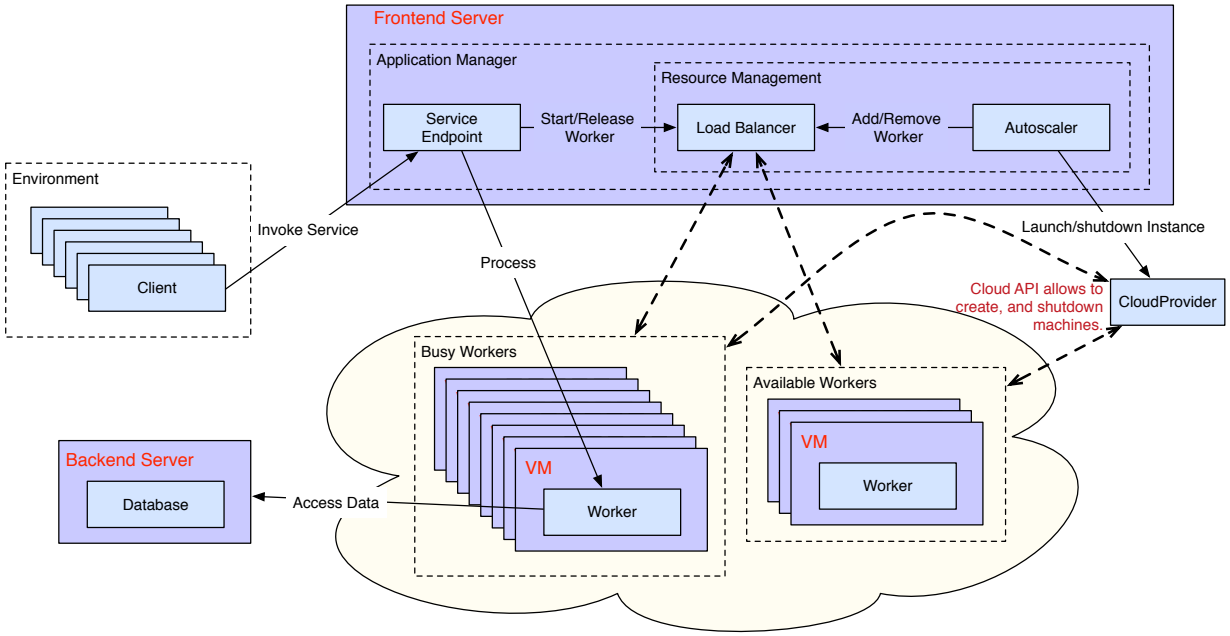

Figure 2: A deployment architecture for an auto-scaling service

```
1  interface Database {
2    Bool accessData(Duration deadline, Time calltime);
3  }
4  class Database (Int transactionCost) implements Database {
5
6    Bool accessData(Duration deadline, Time calltime) {
7      println("[Time: "+toString(timeValue(now()))+"] Database processing job, available time is "+toString(deadline));
8      Int cost = transactionCost;
9      while (cost>0) {
10     [Cost: 1 ] cost = cost −1; suspend;}
11     Rat remainingTime = timeDifference(calltime,now());
12     println("[Time: "+toString(timeValue(now()))+"] Database completed job");
13     if ((durationValue(deadline) − remainingTime) > 0){
14       println("[Time: "+toString(timeValue(now()))+"] Database access succeeded on time!");
15     } else {
16       println("[Time: "+toString(timeValue(now()))+"] Database access violated deadline!");
17     }
18     return (durationValue(deadline) − remainingTime) > 0;
19   }
20 }
```

Figure 3: ABS model of the database

2 Real Time ABS and The ABS Cloud API

Before we focus on the modelling of the example in ABS, let us have a brief introduction of Real Time ABS and the ABS Cloud API.

Real Time ABS extends ABS with the modelling and manipulation of explicit and implicit time. This allows to model and analyse the time-dependent behaviour of applications by representing execution time inside methods using explicit statements, called duration statements. For the complete documentation of ABS and its timed extension, see the reference manual.[1]

The ABS Cloud API provides an interface to the model of a cloud infrastructure in ABS. The ABS Cloud supports the dynamic acquisition/release of virtual machines with resources. Virtual machines in ABS are modelled through *deployment components*. A deployment component is a modelling abstraction of the ABS language which captures locations with resources. In particular for this example we will model computing resources, where the speed of the virtual machines is determined by the amount of computing resources. The speed in this case model the amount of computation that may occur in the objects deployed in these locations. The ABS Cloud API includes methods for launching and shutdown virtual machines. In the implementation, this is done by creating deployment components on which the client application can deploy objects. For the complete documentation of the ABS Cloud API and deployment components, see the resource modelling in timed ABS section of the reference manual. [2]

3 ABS Model of an Auto-scaling Service

Let us now focus on the modelling of the example in Real Time ABS. As a first iteration of our example we will focus on the modelling of the database and its deployment on the ABS Cloud. Later we will focus on the model of a worker. After that we will model the application manager with a service endpoint and a load balancer with a pool of workers. Then we will extend the model with the clients, and as a final extension we will model the autoscaler. Each of these models, corresponding to the different iterations, can be run online using the the Erlang simulation tool in ABS Collaboratory[3] (The code of the example is located at: collaboratory/examples/DeploymentComponent/ElasticDeployment/).

[1] http://abs-models.org/documentation/manual/

[2] http://abs-models.org/documentation/manual/#-resource-modeling-in-timed-abs

[3] http://ei.abs-models.org:8082/clients/web/index.html?file=/collaboratory/examples/DeploymentComponent/ElasticDeplyment&app=erlangbackend

```
1  {// Model a server
2    DC backend = new DeploymentComponent("Backend Server", map[Pair(Speed, 20)]);
3    println("[Time: "+toString(timeValue(now()))+"] Server operational!");
4
5    // Deploy the database on the server
6    [DC: backend] Database db = new Database(2);
7    println("[Time: "+toString(timeValue(now()))+"] Database operational!");
8
9    await duration(1,1);
10   // Testing the access to the database
11   Duration deadline = Duration(5);
12   await db!accessData(deadline,now());
13 }
```

Figure 4: A main block deploying the database on a server

3.1 Modelling the Database

Figure 3 contains the ABS model of the database. The *database* of our example has only one method accessData and is modelled by the class Database. The class Database has a parameter transactionCost which captures the amount of computing resources required for executing a transaction in the database. The method accessData has two parameters, deadline which captures the maximum amount of time that a transaction should take in order to keep a good QoS, and calltime which captures the time when call was made. The implementation of the method abstracts away all the functional behaviour by adding a cost annotation. Note that this implementation can be refined with more explicit functionality and more fine-grained cost annotations which can be obtained using one of the associated ABS tools for cost analysis such SACO[4]. During execution, the cost annotations will consume computing resources from the virtual machine where the object is deployed. The rest of the method implementation checks whether or not the method call has managed to keep its deadline and prints such information on the command line. Here the instruction **now**() returns the current time of the global clock. Note that in this model we are interested to check the QoS and therefore we model hard deadlines instead of soft deadlines.

The main block. Figure 4 contains a main block in ABS which deploys the database on a server. Here are the steps to model the deployment on a server:

1. Create a machines (deployment component) with the desired resources, using the instruction `new DeploymentComponent("Name", map[Pair(kindOfRsc,amountRsc)])`.

2. Deploy objects on the machine (Using the [DC: DCid] annotation).

In particular for this deployment scenario we are creating one machine called backend with computing resources. Afterwards we are deploying a database in this machine, where each transaction consumes 2 computing resources. In addition the main block contains some instructions to test the access to the database. In this simple test we are defining a deadline of duration 5 (measured in time units), and we are testing the service by making an asynchronous call to the method accessData with the defined deadline and the time in which the call was made.

3.2 Modelling the Worker

Figure 5 contains the ABS model of the worker. The *worker* of our example has two methods process and getDC. It is modelled by the class WorkerObject. The class WorkerObject has as a parameter a connection to the database db. The method process has as parameters the amount of resources needed to process a task, the time when the call was made and the deadline of the method call. As before, the implementation of the method process abstracts away all the functional behaviour by adding a cost annotation, but captures the fact that the task needs a transaction in the database, it also takes into account the time of making such

[4] `http://abs-models.org/saco-tutorial/`

```
1  interface Worker {
2      Bool process(Rat taskCost, Time started, Duration deadline);
3      DC getDC();
4  }
5  class WorkerObject(Database db) implements Worker {
6
7    Bool process(Rat taskCost, Time started, Duration deadline) {
8      println("[Time: "+toString(timeValue(now()))+"] Worker start processing job");
9      [Cost: taskCost] skip;
10     Duration remainingTime = subtractFromDuration(deadline, timeDifference(started,now()));
11     println("[Time: "+toString(timeValue(now()))+"] Remaining time until deadline is "+toString(remainingTime));
12     // When we receive more jobs, this becomes a bottleneck
13     Bool success = await db!accessData(remainingTime,now());
14     return success;
15   }
16
17   DC getDC(){ return thisDC();}
18 }
```

Figure 5: ABS model of the worker

```
1  {
2    //parameters
3    Duration respTime = Duration(5);
4    Rat taskCost = 81;
5    Int nResources = 25;
6
7    // Create a server
8    DC backendServer = new DeploymentComponent("Backend Server",map[Pair(Speed, 20)]);
9    println("[Time: "+toString(timeValue(now()))+"] Server operational!");
10
11   // Create the cloud provider
12   CloudProvider cloud = new CloudProvider("cloud");
13   println("[Time: "+toString(timeValue(now()))+"] Created the CloudProvider");
14
15   // Create a virtual machine on the cloud
16   DC virtualServer = await cloud!launchInstance(map[Pair(Speed, nResources)]);
17   println("[Time: "+toString(timeValue(now()))+"] Virtual machine operational!");
18
19   // Deploy database on a server
20   [DC: backendServer] Database db = new Database(2);
21   println("[Time: "+toString(timeValue(now()))+"] Database operational!");
22
23   // Deploy worker on the cloud
24   [DC: virtualServer] Worker w1 = new WorkerObject(db);
25   println("[Time: "+toString(timeValue(now()))+"] Worker operational!");
26
27   await duration (1,1);
28   // Test the deployment
29   Bool success = await w1!process(taskCost, now(), respTime);
30   if (success){
31     println("[Time: "+toString(timeValue(now()))+"] Worker succeeded on time!");
32   } else {
33     println("[Time: "+toString(timeValue(now()))+"] Worker violated deadline!");}
34 }
```

Figure 6: ABS model of the worker

transaction. The rest of the method implementation checks whether or not the method call has managed to keep its deadline. The method getDC returns the id of the virtual machine where the worker is deployed by using the instruction **thisDC**().

```
1  interface LoadBalancer {
2    Worker getWorker();
3    Unit releaseWorker(Worker w);
4    Unit addWorker(Worker w);
5    Worker firingWorker();
6    Int getNbrAvailableW();
7    Int getNbrInuseW();
8  }
9
10 class RoundRobinLoadBalancer()
11 implements LoadBalancer {
12   List<Worker> available = Nil;
13   List<Worker> inuse = Nil;
14
15   Unit run(){
16       [Cost: 1] skip;
17       await duration(1, 1);
18       Int naval = await this!getNbrAvailableW();
19       println("[Time: "+toString(timeValue(now()))+"] Available workers:"+toString(naval));
20       Int ninuse = await this!getNbrInuseW();
21       println("[Time: "+toString(timeValue(now()))+"] In use workers:"+toString(ninuse));
22       println("*******************************************************************************");
23       this!run();
24     }
25
26     Worker getWorker(){
27         [Cost: 1] skip;
28         await (available != Nil);
29         Worker w = head(available);
30         available = tail(available);
31         inuse = appendright(inuse,w);
32         return w;}
33
34     Unit releaseWorker(Worker w){
35         [Cost: 1] skip;
36         available = appendright(available,w);
37         inuse = without(inuse,w);}
38
39     Unit addWorker(Worker w){
40         [Cost: 1] skip;
41         available = appendright(available,w);}
42
43     Worker firingWorker(){
44       [Cost: 1] skip;
45       await (available != Nil);
46       Worker w = head(reverse(available));
47       available = without(available,w);
48       return w;}
49
50     Int getNbrAvailableW() {
51       [Cost: 1] skip;
52       Int a =length(available);
53       return a;}
54
55     Int getNbrInuseW() {
56       [Cost: 1] skip;
57       Int i = length(inuse);
58       return i;}
59 }
```

Figure 7: ABS model of the load balancer

```
1  interface SE {
2      Bool invokeService(Rat cost);
3  }
4
5  class ServiceEndpoint(LoadBalancer lb, Duration responseTime) implements SE {
6      Bool invokeService(Rat cost){
7          Time started = now();
8          [Cost: 1] skip;
9          Worker w = await lb!getWorker();
10         Bool success = await w!process(cost,started,responseTime);
11         await lb!releaseWorker(w);
12         return success;
13     }
14  }
```

Figure 8: ABS model of the service endpoint

```
1  { // Main block
2    CloudProvider cloud = new CloudProvider("cloud");
3    println("[Time: "+toString(timeValue(now()))+"] Created the CloudProvider");
4
5    // Parameters
6    Int nResources = 25;
7    Int nWorkers=15;
8    Duration respTime = Duration(5);
9    // Create server machines
10   DC frontendServer = await cloud!launchInstance(map[Pair(Speed, 35)]);
11   DC backendServer = await cloud!launchInstance(map[Pair(Speed, 20)]);
12   // Deploy the service
13   [DC: backendServer] Database db = new Database(2);
14   [DC: frontendServer] LoadBalancer lb = new RoundRobinLoadBalancer();
15   [DC: frontendServer] SE se = new ServiceEndpoint(lb, respTime);
16   //start workers
17   Int ctr = 0;
18   while (ctr<nWorkers) {
19         Fut<DC> fs = cloud!launchInstance(map[Pair(Speed, nResources)]); DC vm = fs.get;
20         [DC: vm] Worker w = new WorkerObject(db); lb!addWorker(w); ctr=ctr+1;}
21   await duration(1,1);
22   Bool success = await se!invokeService(20);
23   if (success){
24     println("[Time: "+toString(timeValue(now()))+"] Worker succeeded on time!");
25   } else {
26     println("[Time: "+toString(timeValue(now()))+"] Worker violated deadline!");}
27 }
```

Figure 9: A main block deploying the application manager with a service endpoint and a load balancer.

The main block. As in the previous subsection, it is possible to setup a deployment scenario in the main block. Figure 6 contains this deployment. In this case we are creating a cloud and deploying a worker on a virtual machine. The speed of the virtual machine is setup in the variable nResources. Here are the steps to setup a cloud in ABS:

1. Create a new cloud using the instruction **new** CloudProvider("description name").
2. Create a number of virtual machines (deployment components) with the desired resources, using the instruction `await cloud!launchInstance(map[Pair(kindOfRsc,amountRsc)])`.
3. Deploy objects on the virtual machines (Using the [DC: DCid] annotation).

```
1  interface Client {}
2
3  // Client with synchronous calls (i.e., does not flood the system)
4  class ClosedClient (SE ep, Int cycle, Rat cost, Int nbrOfJobs, Counter c) implements Client {
5    Int jobcount = 0;
6    Unit run() {
7      await duration(cycle, cycle);
8      Bool result = await ep!invokeService(cost);
9      if (result == True) {await c!addSuccesses(1);} else {await c!addFails(1);}
10     jobcount = jobcount + 1;
11     if (jobcount < nbrOfJobs) { this!run(); }
12   }
13 }
```

Figure 10: An ABS model of a synchronous client, which waits for a reply before it sends a new invocation

```
1   // Client with asynchronous calls (i.e., floods the system)
2   class OpenClient (SE ep, Int cycle, Rat cost, Int nbrOfJobs, Counter c) implements Client {
3    Int jobcount = 0;
4      Unit run() {
5          Fut<Bool> fresult = ep!invokeService(cost);
6          jobcount = jobcount + 1;
7          await duration(cycle, cycle);
8          if (jobcount < nbrOfJobs) { this!run(); }
9          await fresult?;
10         Bool result = fresult.get;
11         if (result == True) {await c!addSuccesses(1);} else {await c!addFails(1);}
12       }
13 }
```

Figure 11: An ABS model of a periodic client, which sends a new invocation after a fixed period of time

```
1  type MachineUseLog = Map<DeploymentComponent,Pair<Time,Maybe<Time>>>;
2
3  def Pair<A, B> mapHead<A, B>(Map<A, B> map, Pair<A,B> default) = // remove the head of the map
4        case map { EmptyMap => default;
5                   InsertAssoc(head, tail) => head;};
6  def Map<A, B> mapTail<A, B>(Map<A, B> map) = // remove the tail of the map
7        case map { EmptyMap => map;
8                   InsertAssoc(head, tail) => tail;};
9  def Rat calcCost( MachineUseLog ml, Int cost, Int interval, Time nw) =
10       case ml{ InsertAssoc(Pair(id, Pair(s,e)), mltl) => (costMachine(s,e,nw)/interval)*cost + calcCost(mltl, cost, interval,nw);
11                EmptyMap => 0;};
12 def Rat costMachine(Time start ,Maybe<Time> end ,Time nw) =
13       case end { Nothing => timeDifference(start, nw);
14                  Just(e) => timeDifference(start, e);};
```

Figure 12: ABS data types and functions to calculate resource cost on the cloud

3.3 Modelling the Load Balancer and the Service Endpoint

Let us now model the application manager with a service endpoint and a load balancer with a pool of workers connected to a shared database.

The load balancer. Figure 7 contains the ABS model of a load balancer with a round robin scheduling policy. The RoundRobinLoadBalancer implements the interface LoadBalancer. This class keeps two lists, one with the identifiers of workers which are in use (or busy) and one with the identifiers of workers which are available. The class also has methods to add and remove workers to the lists, and to move workers from one list to the other. Method addWorker adds the identifiers of new workers to the list of available workers. In

```
1  interface Counter {
2    Unit addSuccesses(Int amount);
3    Unit addFails(Int amount);
4    Unit addMachine(DeploymentComponent id, Time startup);
5    Unit addShutdown(DeploymentComponent id, Time shutdown);
6    Unit printSuccess();
7    Unit printFail();
8    Unit calculateCost(Int cost, Int interval,Time until);
9    Unit printMachineUseLog();
10 }
11
12 class Counter() implements Counter {
13   Int success = 0;
14   Int fail = 0;
15   Int cost = 0;
16   MachineUseLog machines = EmptyMap;
17
18   Unit addSuccesses(Int amount) {success = success+amount;}
19   Unit addFails(Int amount){fail = fail+amount;}
20   Unit printSuccess(){println("[Time: "+toString(timeValue(now()))+"] *******Total successes:"+ toString(success));}
21   Unit printFail(){println("[Time: "+toString(timeValue(now()))+"] ********Total Fails:"+ toString(fail));}
22   Unit addMachine(DeploymentComponent id, Time startup){machines = InsertAssoc(Pair(id,Pair(startup,Nothing)), machines);}
23   Unit addShutdown(DeploymentComponent id, Time shutdown){
24         Pair<Time,Maybe<Time>> tmp = lookupDefault(machines, id,Pair(Time(−1),Nothing));
25         machines = InsertAssoc(Pair(id, Pair(fst(tmp),Just(shutdown))), removeKey(machines, id));}
26   Unit calculateCost(Int cost, Int interval,Time until){
27         Rat calculatedCost = calcCost(machines, cost, interval, until);
28         println("[Time: "+toString(timeValue(now()))+"] Cost (until "+toString(timeValue(until))+ "):"+ toString(calculatedCost));
29   }
30   Unit printMachineUseLog(){
31        MachineUseLog tmp = machines;
32        DeploymentComponent tdc = null;
33        while (tmp != EmptyMap) {
34          Pair<DeploymentComponent,Pair<Time,Maybe<Time>>> head = mapHead(tmp,Pair(tdc,Pair(Time(−1),Nothing)));
35          DeploymentComponent dc = fst(head);
36          String name = await dc!getName();
37          Pair<Time,Maybe<Time>> tmptime = snd(head);
38          Time frmt = fst(tmptime);
39          Maybe<Time> tot = snd(tmptime);
40           if (tot != Nothing){println("[Time: "+toString(timeValue(now()))+"]"+name+"−>
41                                        ("+toString(timeValue(frmt))+","+toString(timeValue(fromJust(tot)))+")");}
42           else {println("[Time: "+toString(timeValue(now()))+"]"+name+"−>("+toString(timeValue(frmt))+",−)");}
43           tmp = mapTail(tmp);
44         }
45    }
46 }
```

Figure 13: An ABS class for monitoring the number of successes, fails and resource cost of the service

this version of the system, this method is called by the main block to configure a static deployment scenario with a fixed number of workers. Methods getWorker and releaseWorker move workers from the lists available to inuse and vice versa. These methods are called by the service endpoint to assign processing jobs to the workers. Method firingWorker permanently removes a worker from the list of available workers. Methods getNbrAvailableW and getNbrInuseW calculates the number of available workers and the number of workers that are in use, respectively. Note that we are not using the last three methods in this version of the system, they are used later in Section 3.5.

The service endpoint. Figure 8 contains the ABS model of the service endpoint which is in charge of receiving the calls from the clients and forward it to one of the workers (that the load balancer choses). The ServiceEndpoint class has two parameters. The first parameter is a connection to the load balancer lb, and the second parameter is the expected response time (or deadline) per call. This class has one methods

```
1  interface Autoscaler {}
2  class Autoscaler(CloudProvider cloud, LoadBalancer lb, Int nbrOfWorkers, Int maxWorkers,
3                   Int nResources,Database db, Int cycle, Counter c)
4  implements Autoscaler {
5    Unit run(){
6        [Cost: 1] skip;
7        Int ctr = 0;
8        while (ctr<nbrOfWorkers) {
9          Fut<DC> fs = cloud!launchInstance(map[Pair(Speed, nResources)]); DC vm = fs.get;
10         [DC: vm] Worker w = new WorkerObject(db); lb!addWorker(w);
11         Time startTime = await vm!getCreationTime(); await c!addMachine(vm,startTime);
12         ctr=ctr+1;
13       }
14       println("[Time: "+toString(timeValue(now()))+"] **********INIT: CREATED "+toString(nbrOfWorkers)+" WORKERS");
15       this!resize();
16   }
17   Unit resize(){
18       [Cost: 2] skip;
19       Int ctr = 0;
20       await duration(cycle, cycle);
21       Int available = await lb!getNbrAvailableW();
22       Int inuse = await lb!getNbrInuseW();
23       if (available < ((available+inuse)/4) && (available+inuse)<=maxWorkers/2){
24         ctr = 0;
25         Rat extraworkers= inuse;
26         while (ctr<extraworkers ) {
27           Fut<DC> fs = cloud!launchInstance(map[Pair(Speed, nResources)]); DC vm = fs.get;
28           [DC: vm] Worker w = new WorkerObject(db); await lb!addWorker(w);
29           Time startTime = await vm!getCreationTime(); await c!addMachine(vm,startTime);
30           ctr=ctr+1;
31         }
32       }
33       if ((inuse/3 < available) && (available > nbrOfWorkers))
34       {
35         ctr = 0;
36         Rat removeworkers= available/2;
37         while (ctr<removeworkers) {
38           Worker w = await lb!firingWorker();
39           DC dc = await w!getDC();
40           Bool down = await cloud!shutdownInstance(dc);
41           await c!addShutdown(dc, now());
42           ctr=ctr+1;
43         }
44       }
45       this!resize();
46     }
47 }
```

Figure 14: ABS model of the autoscaler.

invokeService which is called by the clients. This method forwards the call to one of the workers. The method also checks weather the worker manages to process the call within the deadline.

The main block. Figure 9 contains a main block in ABS which deploys the current model. In this case we have a deployment scenario in which we initially create two servers frontendServer and backendServer. In the backendServer we deploy the database db and in the frontendServer we deploy the load balancer lb and the service endpoint se. We also deploy a fixed number of virtual machines with object workers on the cloud. As in the previous iterations, we also include a small test to check that it is possible to make an invocation to the service that we just deployed.

```
1  { CloudProvider cloud = new CloudProvider("cloud");
2    println("[Time: "+toString(timeValue(now()))+"] Created the CloudProvider");
3
4    // Parameters
5     Int nResources = 25;
6     Int nWorkers=4;
7     Int maxWorkers = 25;
8     Int interval = 1;
9     Duration respTime = Duration(5);
10    Int nCloseClients = 4;
11    Int nOpenClients = 10;
12    Rat taskCost = 81;
13    Int nbrOfJobs = 2;
14
15    // Create server machines
16     DC frontendServer = await cloud!launchInstance(map[Pair(Speed, 35)]);
17     DC backendServer = await cloud!launchInstance(map[Pair(Speed, 20)]);
18
19     Counter c = new Counter();
20     // Deploy system
21     [DC: backendServer] Database db = new Database(2);
22     [DC: frontendServer] LoadBalancer lb = new RoundRobinLoadBalancer();
23     [DC: frontendServer] Autoscaler as = new Autoscaler(cloud,lb,nWorkers,maxWorkers,nResources,db,interval,c);
24     [DC: frontendServer] SE endpoint = new ServiceEndpoint(lb, respTime);
25
26     await duration(10,10);
27     // Start up the close clients
28     Int nClients = nCloseClients;
29     while (nClients > 0) {
30          new ClosedClient(endpoint, 2,taskCost, nbrOfJobs,c); nClients = nClients − 1;
31     }
32    println("[Time: "+toString(timeValue(now()))+"] *********CREATED " +toString(nCloseClients)+" CLOSE CLIENTS
WITH " +toString(nbrOfJobs)+ " JOBS EACH");
33     await duration(10,10);
34     // Start up the open clients
35     nClients = nOpenClients;
36     while (nClients > 0) {
37          new OpenClient(endpoint, 1, taskCost, nbrOfJobs,c); nClients = nClients − 1;
38     }
39     println("[Time: "+toString(timeValue(now()))+"] ********CREATED " +toString(nOpenClients)+" OPEN CLIENTS WITH "
+toString(nbrOfJobs)+ " JOBS EACH");
40      await duration(10,10);
41     // Start up the close clients
42     nClients = nCloseClients;
43     while (nClients > 0) {
44          new ClosedClient(endpoint, 2,taskCost, nbrOfJobs,c); nClients = nClients − 1;
45     }
46   println("[Time: "+toString(timeValue(now()))+"] **********CREATED " +toString(nCloseClients)+" CLOSE CLIENTS
WITH " +toString(nbrOfJobs)+ " JOBS EACH");
47     await duration(10,10);
48     // Start up the open clients
49     nClients = nOpenClients;
50     while (nClients > 0) {
51          new OpenClient(endpoint, 1, taskCost, nbrOfJobs,c); nClients = nClients − 1;
52     }
53     println("[Time: "+toString(timeValue(now()))+"] CREATED " +toString(nOpenClients)+" OPEN CLIENTS WITH "
+toString(nbrOfJobs)+ " JOBS EACH");
54     await duration(10,10);
55     await c!printSuccess(); await c!printFail();
56     await c!calculateCost(cost,interval,now()); await c!printMachineUseLog();
57   }
```

Figure 15: Dynamic deployment of the service.

3.4 Modelling the Environment and Monitoring of the Service

The environment. Let us now model the environment. In this example the environment consists of clients calling the service. We are going to model two kind of clients ClosedClient and OpenClient. Figure 10 contains the implementation of a client which does not flood the system. This class has four parameters. The first parameter is a connection to the service endpoint, the second parameter captures how long the client waits before it sends an invocation to the service, the third parameter is the computing cost of each invocation, and the fourth parameter is the number of jobs (invocations) that a client will send. The behaviour of the client is captured in the run method. In each iteration the client waits for a cycle to pass using the instruction **await duration**(cycle, cycle); the **duration** instruction has two parameters, the minimum and maximum amount of simulated time to suspend the process. During execution, the simulation tool will randomly chose a number between these to parameters. Since in our example both parameters are the same, then this client will be waiting (or suspended) for exactly a cycle. After that the client will send an invocation for the service and wait for the result using the instruction Bool result = **await** ep!invokeService(cost). The iteration will finish once the client manage to send the desired number of jobs.

Figure 11 contains the implementation of a client which might flood the system. This class has the same parameters of the ClosedClient. Similar to the previous client, the behaviour is captured in the run method. In each iteration the client creates a future variable (mail box) which will store the result of the invocation and sends the call with the instruction **Fut**<Bool> fresult = ep!invokeService(cost). Note that there is not **await** in this instruction, and therefore the client continues its execution without waiting for the result (opposite to the ClosedClient). As before the client waits for a cycle to pass before it call a new run method. Finally the method waits for the result to arrive using the instruction **await** fresult?. As before, the client will stop after it manage to send invocations for the desired number of jobs. Note that if this client sends many jobs in a very high frequency, the system might experience congestion and will not manage to retrieve the data on time.

Monitoring the service. The service can interact with an object which is in charge of keeping the accounting of the successes and fails of calling the service as well as the number of machines that have been created on the cloud (See Section 3.5). Figure 13 contains the ABS implementation of the class Counter which has various methods, two of them are in charge of adding either a success or a fail to the counters success and fail respectively and two methods print these two counters. The rest of the methods, which manipulate the map machines, are used in Section 3.5. The map machines records all the virtual machines that the service has used on the cloud with their starting up and shutting down times. The method addMachine adds machines to the map with their starting time. The method addShutdown adds the shutdown time of existing machines on the map. The method calculateCost calculates and prints the total cost, until a specified time and billing cost per interval, of all the machines recorded on the map. Finally, the method printMachineUseLog prints the map. These methods require the datatypes and functions defined in Figure 12.

3.5 Modelling the Autoscaler

In this section, we extend the model with an autoscaler to allow dynamic reconfiguration of the deployment on the cloud. Figure 14 models an autoscaler which increases and decreases the number of workers deployed on the cloud depending on the number of available workers. The run method models the initial deployment of workers. The method resize acts as a monitor which periodically checks if we need to adjust the number of workers. These methods use the counter monitor described in Section 3.4 to record the varying number of machines available per time interval. The autoscaler uses the following *ad hoc* policy:

> *"If the number of available workers is less than one quarter of the total number of workers and less than half of the maximum number of workers, then we double the number of available workers. If the number of available workers is greater than one third of the busy workers, then we reduce the number of available workers to half."*

The main block. Figure 15 models the initial deployment of the system as depicted in Figure 1. In this case we create the database, the load balancer, the autoscaler with an initial number of workers and a maximum number of workers, and the service endpoint. We also need to add the monitor counter and a workload. In this case the workload consist of to first create 4 closed clients at time 10 and at time 30 and 10

open clients at time 20 and at time 40. Each client will send 2 requests, where each request has an average resource cost of 81, specified in the parameter taskCost. The closed client has a cycle of 2 time units while the open client has a cycle of one time unit, creating peaks of requests. We will use this workload to test the QoS and the accumulated billing cost of the different deployment scenarios for the system.

4 Simulation Results

Using an static deployment (allowing scaling) and dynamic deployment (allowing elasticity), we now can run simulations and observe the behaviour of the service. As an example, let us observe if the model satisfies the following Service Level Agreement (SLA), formulated in terms of modelling time:

> *"The service must maintain a response time of less than 5 time units with an average success rate of at least 90%. In addition, for an interval of 50 time units, the billing cost should not exceed the amount of 25000, where the billing cost is 50 per virtual machine, charged every time unit."*

For the static scenario we are considering a main block which creates 15 machines on the cloud (Similar to the deployment of Figure 9) with the same workload as specified in Figure 15. Figure 16 contain the result of comparing one run of these two scenarios until time 50. We can observe that the static scenario breaks the SLA because the success rate is 82% and the billing cost is 37500. On the other hand, the dynamic scenario complies with the SLA with a success rate of 92% and a billing cost of 17800.

```
1 // Result until time 50 for an static deployment with 15 virtual machines. Similar to the deployment of Figure 9.
2 [Time: 50] **************************Total successes:46
3 [Time: 50] **************************Total Fails:10
4 [Time: 50] **************************Total Cost on the cloud (until time 50):37500
5
6 // Result until time 50 for the dynamic deployment in Figure 15.
7 [Time: 50] **************************Total successes:52
8 [Time: 50] **************************Total Fails:4
9 [Time: 50] **************************Total Cost on the cloud (until time 50):17800
```

Figure 16: Simulation results for one run of the service with static deployment and one run of the service with dynamic deployment.

Results of more complex scenarios of a similar example can be found in the following white paper: Modeling Deployment Decisions for Elastic Services with ABS.[5]

[5] http://www.envisage-project.eu/modeling-deployment-decisions-for-elastic-services-with-abs/