Project Nº: **FP7-610582**

Project Acronym: **ENVISAGE**

Project Title: **Engineering Virtualized Services**

Instrument: **Collaborative Project**

Scheme: **Information & Communication Technologies**

Deliverable D4.4.2
Resource Aware Modelling the ENG 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: **ENG**

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 Modelling the ENG Case Study

This document summarises deliverable D4.4.2 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 detailed modelling of the different deployment scenarios of the ENG case study in the abstract behavioural specification language.

List of Authors

Keven T. Kearney (ENG)

1. Introduction

This deliverable details resource and deployment scenario modelling for the ENG case study, and is a refinement of the initial case study model given in D.4.4.1[1]. The case study concerns "ETICS", an online code build and test service for software developers. For the present deliverable, the architecture of ETICS has been simplified to just two components (indicated in Figure 1-*a*):

- ***Resource Pool***: a dynamic collection of virtual computing machines (VMs), sourced on-demand from 3*rd*-party cloud providers;
- ***Resource Pool Manager*** (RPM), responsible for deciding, on the basis of a *utility* function:
  - which VMs to deploy in the Resource Pool (the RPM can add/remove VMs as needed);
  - whether to accept or reject end-user (build/test) requests, and if accepted, which of the pooled VMs should process those requests, and in what order.

Each VM has a dedicated input queue/buffer for receiving requests. The RPM 'assigns' a request to a VM just by adding that request to the VM's input queue. The VM processes queued requests *one at a time* in FIFO order.

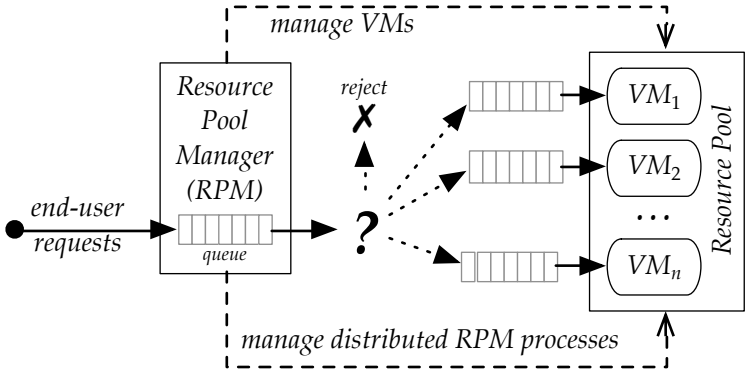

Figure 1-a: Simplified ETICS Architecture

Although simple in outline, the complexity of the *request assignment* task facing the RPM grows exponentially with the number of requests and VMs. For a single request, p, with one pooled VM, m, for example, the RPM has only 3 options: reject p, assign p to m, or launch, and assign p to, a new VM. For 2 requests and 2 pooled VMs, however, the number of options rises dramatically to 20 (the options are enumerated in Box 1-*a*).

Box 1-a: Assignment options for 2 requests, p_1 & p_2, and 2 pooled VMs, m_1 & m_2:

- reject both requests, which we can denote: $\langle p_1 \rightarrow \times, p_2 \rightarrow \times \rangle$
- reject one of the requests and assign the other to an existing VM, with 4 options: $\langle p_1 \rightarrow m_1, p_2 \rightarrow \times \rangle$, $\langle p_1 \rightarrow m_2, p_2 \rightarrow \times \rangle$, $\langle p_1 \rightarrow \times, p_2 \rightarrow m_1 \rangle$ or $\langle p_1 \rightarrow \times, p_2 \rightarrow m_2 \rangle$
- reject one of the requests and assign the other to a *new* VM, m_3: $\langle p_1 \rightarrow m_3, p_2 \rightarrow \times \rangle$ or $\langle p_1 \rightarrow \times, p_2 \rightarrow m_3 \rangle$
- assign each request to a different pooled VM: $\langle p_1 \rightarrow m_1, p_2 \rightarrow m_2 \rangle$ or $\langle p_1 \rightarrow m_2, p_2 \rightarrow m_1 \rangle$
- assign one request to an existing VM and the other to a new VM: $\langle p_1 \rightarrow m_1, p_2 \rightarrow m_3 \rangle$, $\langle p_1 \rightarrow m_2, p_2 \rightarrow m_3 \rangle$, $\langle p_1 \rightarrow m_3, p_2 \rightarrow m_1 \rangle$ or $\langle p_1 \rightarrow m_3, p_2 \rightarrow m_2 \rangle$
- assign both requests to new, but different VMs: $\langle p_1 \rightarrow m_3, p_2 \rightarrow m_4 \rangle$
- assign both requests to a single existing VM (the order that requests are queued is significant, hence there are 4 options): $\langle p_1, p_2 \rightarrow m_1 \rangle$, $\langle p_1, p_2 \rightarrow m_2 \rangle$, $\langle p_2, p_1 \rightarrow m_1 \rangle$ or $\langle p_2, p_1 \rightarrow m_2 \rangle$
- assign both requests to a new VM (again, order matters): $\langle p_1, p_2 \rightarrow m_3 \rangle$ | $\langle p_2, p_1 \rightarrow m_3 \rangle$

[1] For reference, the key changes *w.r.t.* D4.4.1 are outlined in Appendix B.

In all cases, the *best* option is determined by a *utility* function, which is calculated according to both the *state* of requests & VMs (e.g. queuing-time, busy *vs* idle) and their governing *QoS constraints* (as defined by SLAs). For requests, the key QoS constraint is *completion-time* (how long it takes, from the end-user's point-of-view, to execute the request), with the ETICS service provider penalised for 'late' requests. Regardless of how many requests it has received or how many VMs are available, therefore, the RPM must always make its decisions *quickly*. To achieve this, we employ a distributed mechanism for determining the best assignment option, with the computational load shared evenly between the RPM and the pooled VMs. In particular, we have chosen to use a *distributed genetic algorithm* (DGA).

The sections below describe the utility function (§2), the DGA (§3), the revised ABS model for ETICS (§4), and a summary of the status of the case study *w.r.t.* project objectives (§5). A formal specification of the assignment task and utility function is provided in Appendix A. The complete listing of the ABS code is provided in the zipped file, **`ENG_case_study_442`**, accompanying this deliverable.

In light of the formal specification, the overall *business* goal of the ENG Case Study can be defined in precise terms as the determination of suitable values for a handful of constants employed in the utility function[2]. This objective will be properly addressed in Deliverable D4.4.3 (due M34), but the basic approach is to tune the values of these constants by *simulation* of the ETICS service under diverse (stochastic) usage patterns. The simulator forms part of the ABS model described in §4.

With respect to the ABS analytic tools, the case study has two key objectives:

1) *To ensure (if possible) that the DGA is deadlock free.* In parallel with the development of the ABS model, we have also built a working prototype in Swift, with a graphical front-end showing an animated display of the resource pool and request assignments. On most runs, however, this prototype eventually stalls, with the animation frozen and CPU usage dropping to zero. Formal analysis of the ABS model could help show whether this is due to the DGA or other factors (e.g. some quirk of the Swift/Cocoa graphics system).
2) *To ensure that the DGA scales*. Thus far the *distributed* algorithm has only been tested in simulation on a *serial* machine, and with only small numbers (up to 1000) of requests and VMs. Testing on a larger scale on a single machine is impractical (the CPU & memory requirements are too high). The intention, therefore, is to use formal ABS analytic methods (if possible) to ensure scalable performance under high/extreme loads.

Once again, these objectives will be addressed in Deliverable D4.4.3. For this deliverable, the goal is just to describe the operation of the simulator/RPM and in particular the utility function and DGA.

2. Utility Function

At any point in time, the RPM will have a set of requests that it needs to deal with, and a set of VMs to which these requests may be assigned for processing. In all but the most trivial cases, as described above (*cf*. Box 1-*a*), there will be many possible assignment options. The utility function essentially calculates the expected financial profit (or loss) *for a given option*, and the RPM then selects the option with the largest profit. The formal definition of the utility function is given in Appendix A.3, here we provide a simplified overview. The basic formula for calculating profit is:

profit = income - penalties - machine costs

where:

- *income* = the price paid by the ETICS end-user for completed requests;
- *penalties* = refunds paid by the ETICS provider for SLA violations;
- *machine costs* = the price paid by the ETICS provider for using VMs.

[2] Specifically: the 'action-time', δ_{AT} (Appendix A.2.1) - which is essentially a limit on the time taken for the RPM to decide what to do with requests - and the global constants χ_ρ, κ_{CT}, χ_{CT} and χ_{FR} described in Appendix A.2.3.

The *income*, *penalties* & *machine cost* values are derived from SLAs, which for present purposes we treat in a simplified abstract fashion, capturing only the essential details. There are two kinds of SLA (the formal definitions are given in Appendices A.1.3 & A.2.3):

- ***Consumer Facing SLAs***: governing the use of the ETICS service, specifying:
  - a *service level*: either *bronze* (cheap with low QoS), *silver* or *gold* (expensive, high QoS);
  - *request income*: how much the end-user pays (in €) for each request accepted by the service ~ dependent on the service level, request *priority*[3], and request *size* (where 'size' is an abstract measure of the computational complexity of the request[4]);
  - *maximum completion-time*: the maximum time, from the receipt of a request, that the provider has to satisfy that request without incurring penalties ~ once again dependent on service level, request size & request priority.
  - *completion-time penalty*: the penalty (€) for requests that fail to complete within the maximum completion-time ~ defined in proportion to the time in excess of the maximum;
  - *maximum failure rate*: a limit on the number of requests (from a given user) that the provider can reject without incurring penalties ~ dependent just on service level;
  - *failure-rate penalty*: the penalty (€) for each rejected request in excess of the maximum failure rate ~ dependent again just on service level.
- ***Cloud-Provider Facing SLAs***: governing the ETICS provider's use of third party VMs, specifying:
  - *cost-per-hour*: how much a VM costs ~ defined in proportion to the VM's capabilities (e.g. faster machines with more memory & disk space cost more);
  - *deploy-time*: the time, from launching (or requesting) a VM, until that VM is available for use ~ dependent again on the machine's capabilities (e.g. higher spec' VMs take longer to deploy).

The capabilities of a VM are captured by a single value which is the just sum of the its disk capacity and processing power, which latter is a function of the VM's CPU speed, number of computing cores, and memory (see Appendix A.1.2[5]). A VM's capabilities determine which requests it can process, and how quickly it can process them:

- a VM can only process a request if it has sufficient disk capacity, where the minimum capacity is defined as a function of the request's size.
- the time taken for a VM to process a request is proportional to the size of the request (big requests take longer) over the VM's processing power (fast machines process requests more quickly).

To illustrate the application of these SLAs, consider a simple case in which the RPM has received just one request, $\boldsymbol{p}$, from an end-user, $\boldsymbol{u}$, and has just one pooled VM, $\boldsymbol{m}$ (with sufficient disk capacity to process $\boldsymbol{p}$). The RPM has 3 options:

i) *Reject* $\boldsymbol{p}$: if the maximum failure *w.r.t.* end-user, $\boldsymbol{u}$, has been reached[6], then rejecting $\boldsymbol{p}$ will incur a failure-rate penalty (the value of which depends on the service level of the governing SLA);

ii) *Assign* $\boldsymbol{p}$ *to* $\boldsymbol{m}$: end-user, $\boldsymbol{u}$, must pay for the request (according to the size of the request and the service-level of the governing SLA), but this income is offset by two factors:
  - The processing time for the request may exceed the maximum completion-time, and so incur a completion-time penalty. This is determined by the size of $\boldsymbol{p}$ and the processing power of $\boldsymbol{m}$ (as stated above), but also by the *state* of $\boldsymbol{m}$. In particular, $\boldsymbol{m}$ cannot begin processing $\boldsymbol{p}$ until:
    - it has completed its deployment phase ($\boldsymbol{m}$ may have only recently been launched);
    - it has finished processing any and all requests, $\boldsymbol{p}_1$, $\boldsymbol{p}_2$, $\boldsymbol{p}_3$, …, assigned to it prior to $\boldsymbol{p}$ (recall that each VM has a FIFO queue of assigned requests), where the time required to complete $\boldsymbol{p}_1$, $\boldsymbol{p}_2$, $\boldsymbol{p}_3$, … again depends on their sizes and the capabilities of $\boldsymbol{m}$.

---

[3] We distinguish low priority 'scheduled' requests (e.g. nightly builds) from high priority 'ad-hoc' requests.

[4] We assume that the 'size' of a request is readily ascertainable.

[5] The formal definitions allow for ≈ 300 different VM configurations, and are chosen such that VM *cost-per-hour* and *deploy-time* are comparable to those of Amazon EC2 (https://aws.amazon.com/ec2), see Tables A.1.3-*b*..*d* for comparisons.

[6] The failure count (for each user) is reset after every 50 requests received from that user, i.e. the failure 'rate' is in fact a ratio.

- Since VMs are paid for by the hour, assigning ***p*** to ***m*** will incur a machine cost whenever the processing of ***p*** requires that an additional hour of VM time be purchased.

iii) *Assign **p** to a new VM*: as for *ii*, but with only the deploy-time of the VM to account for (since its queue will be empty).

There is one additional factor that affects the completion-time of requests, and that is the time that it takes for the RPM to decide what to do with the request. We refer to this as the '*action time*'. By way of summary, Figure 2-*a* shows the main points along the time-line of an accepted request.

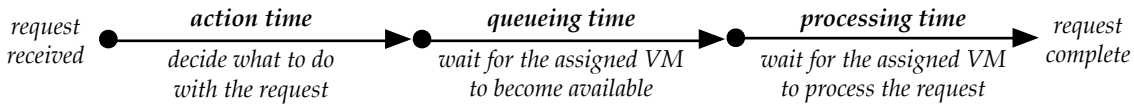

Figure 2-a: Time-line for an assigned request

The utility function essentially takes all of the preceding factors into account to calculate an estimate of the profit/loss to be expected from either rejecting request or assigning it to given VM. The next section (§3) explains how the utility function is applied in the general case (with multiple requests and multiple pooled VMs).

To close this section, it should be apparent that there is a basic trade-off in the assignment of requests to VMs. Powerful VMs process requests more quickly, which leads to shorter completion-times, hence higher income and fewer completion-time penalties, and so more profit. But powerful VMs also cost more, which cuts into this profit. Ignoring other factors (namely action & queuing times), this trade-off means that for every request there should be an optimal VM configuration for processing that request (i.e. there is a peak, as indicated in Figure 2-*b*, in the plot of profit against processing power).

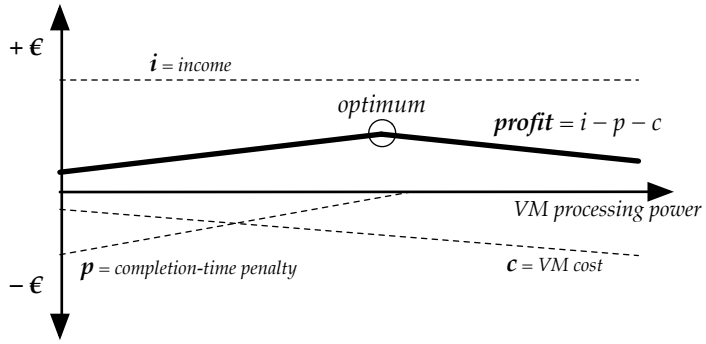

Figure 2-b: Optimal VM for a Given Request
*Indicative plot showing how (for a given request) the income (**i**), completion-time penalty (**p**), VM cost (**c**) & profit (**i - p - c**), vary with the processing power of the VM*

Under the particular formalisation of the utility function given in Appendix A, it turns out that the characteristics of this optimal VM depend only on the size of the request. The other factors (action-time, queueing-time, etc.) evaporate. Specifically (as detailed in Appendix 5.1), the optimal VM has the following properties:

$$\textit{disk capacity} = 2^{\lceil \log_2(\textit{ request size }) \rceil}$$

$$\textit{processing power} = \sqrt{(8.25 \times \textit{request size})}$$

For *any* request that it receives, the RPM may choose to launch, and assign that request to, a new VM. The best choice for this new VM is the optimal VM as just defined, which we will henceforth refer to as the 'default' VM. The fact that this default VM is dependent only on the size of requests greatly simplifies the task of the RPM (since the default only needs to be calculated once for each request upon its receipt). Almost everything else that RPM does is then covered by the distributed genetic algorithm (DGA), which is described in the next section.

3. Distributed Genetic Algorithm

Genetic algorithms (GAs) are a class of heuristic inspired by natural selection processes and often applied for solving optimisation problems. They require:

- a means to encode solutions as modifiable 'gene' structures;
- a 'fitness' function for ranking (assigning a qualitative score to) solutions;

They operate, briefly, as follows:

i) an initial set of random solutions (genes) is generated;
ii) the fitness function is applied to each member of the set;
iii) solutions with highest fitness are used to generate a new set of solutions by the application of 'mutation' and 'crossover' operators (described shortly);
iv) steps *ii* & *iii* are applied iteratively to the new set of solutions, for as long as required.

A *distributed* GA processes multiple sets of solutions concurrently, with an occasional exchange of the fittest solutions between sets. For this case study, the fitness function is the utility function described in the previous section, and the GA is distributed over the RPM and every pooled VM. The remainder of this section briefly describes the genetic encoding of solutions, and mutation & crossover operators.

To begin, at any point in time, the RPM will have a set, $\boldsymbol{P}$, of requests that need to be dealt with, and a set $\boldsymbol{M}$, of pooled VMs. For each request in $\boldsymbol{P}$ there is also a corresponding default VM (introduced in the previous section), such that we also have a set $\boldsymbol{D}$ of default (not yet launched) VMs. A 'solution' to the assignment task then consists of a partial mapping from a (random) ordering over $\boldsymbol{P}$ onto the set $\boldsymbol{M} \cup \boldsymbol{D}$ of both pooled and default VMs. To illustrate:

- Suppose $\boldsymbol{P}$ (the set of requests) = $\{ p_1, p_2, p_3, p_4 \}$, with a corresponding set, $\boldsymbol{D} = \{ d_1, d_2, d_3, d_4 \}$, of default VMs (i.e. the default VM for p_1 is d_1, the default VM for p_2 is d_2, etc.);
- Suppose $\boldsymbol{M}$ (the set of pooled VMs) = $\{ m_1, m_2 \}$
- We impose a random order on $\boldsymbol{P}$, say $\boldsymbol{P'} = \langle p_2, p_4, p_1, p_3 \rangle$, and define a partial function from $\boldsymbol{P'}$ onto $\boldsymbol{M} \cup \boldsymbol{D}$, producing, for example, $\langle p_2 \rightarrow m_1, p_4 \rightarrow m_1, p_1 \rightarrow \times, p_3 \rightarrow d_1 \rangle$ (compare to Box 1-*a*) ~ where:
  - $p_2 \rightarrow m_1, p_4 \rightarrow m_1$ denotes that p_2 & p_4 are assigned *in that order* to the same pooled VM m_1;
  - $p_i \rightarrow \times$ denotes that p_i is not mapped to any VM, i.e. it is *rejected*;
  - $p_3 \rightarrow d_1$ denotes that p_3 is assigned to the default VM, d_1, which will have to be acquired and launched before it can process p_3 (note that this mapping is perfectly valid even though d_1 is *not* the default VM for p_3).

Figure 3-*a* provides a visual representation of the mapping $\langle p_2 \rightarrow m_1, p_4 \rightarrow m_1, p_1 \rightarrow \times, p_3 \rightarrow d_1 \rangle$. Each such mapping encodes a complete solution to the assignment task (for some $\boldsymbol{P}$ & $\boldsymbol{M}$). The utility function is applied to individual solutions, and gives the profit that can be expected from applying that solution.

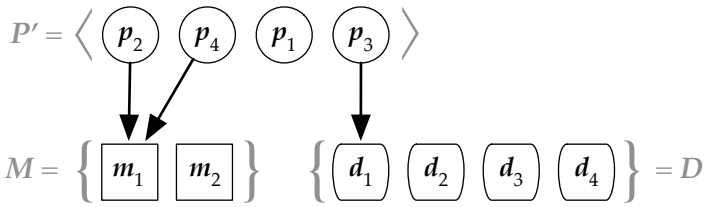

Figure 3-a: 'Genetic' Representation of an Assignment Solution

Given this 'genetic' representation of the solution space, the mutation operator is trivial: an element of $\boldsymbol{P'}$ (the ordered set of requests) is chosen at random, and its mapping onto $\boldsymbol{M} \cup \boldsymbol{D}$ is randomly modified. Given the solution $\langle p_2 \rightarrow m_1, p_4 \rightarrow m_1, p_1 \rightarrow \times, p_3 \rightarrow d_1 \rangle$, for example, we might arbitrarily choose to mutate $p_1 \rightarrow \times$ into $p_1 \rightarrow m_2$, producing the *new* solution $\langle p_2 \rightarrow m_1, p_4 \rightarrow m_1, p_1 \rightarrow m_2, p_3 \rightarrow d_1 \rangle$.

The crossover operator is only slightly more complex. The goal of crossover is to combine 2 solutions to produce another which (in some useful sense) combines the characteristics of both originals. Given

two mappings, say $\alpha = \langle p_2 \rightarrow m_1, p_4 \rightarrow m_1, p_1 \rightarrow \times, p_3 \rightarrow d_1 \rangle$ and $\beta = \langle p_1 \rightarrow m_1, p_3 \rightarrow \times, p_4 \rightarrow d_2, p_2 \rightarrow d_3 \rangle$, crossover proceeds in three steps (illustrated in Figure 3-*b*):

i) Select a random element in α, and remove all the subsequent elements ~ e.g. choosing the 2nd element ($p_4 \rightarrow m_1$), gives the subset $\alpha' = \langle p_2 \rightarrow m_1, p_4 \rightarrow m_1 \rangle$;

ii) For each element, $p_i \rightarrow ?$, in α', remove from β the element that has the same request, p_i ~ e.g. for $\alpha' = \langle p_2 \rightarrow m_1, p_4 \rightarrow m_1 \rangle$, remove the elements $p_2 \rightarrow d_3$ & $p_4 \rightarrow d_2$ from β, to give $\beta' = \langle p_1 \rightarrow m_1, p_3 \rightarrow \times \rangle$;

iii) Concatenate α' with β' to get the resulting 'spliced' solution ~ e.g. from above, appending β' onto α' results in $\langle p_2 \rightarrow m_1, p_4 \rightarrow m_1, p_1 \rightarrow m_1, p_3 \rightarrow \times \rangle$

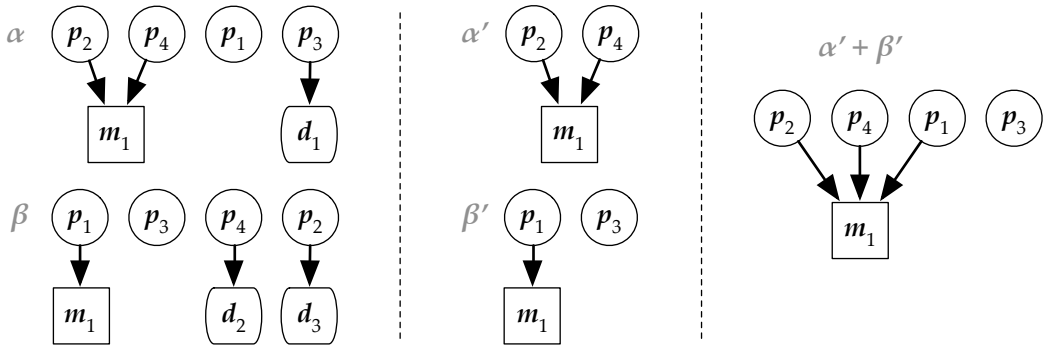

Figure 3-b: Example application of the crossover operator

The following pseudo-code briefly summarises the *distributed* GA. If **P** is a set of requests, **V** = $M \cup D$ is the combined set of pooled & default VMs, and **solvers** is the set of distributed GA processes (one for the RPM, and one for each pooled VM), then each 'solver' (more or less) executes the following[7]:

```
solutions = [] // clear any previous results
self.bestSolution = NULL
for i in 0..n{ // where n is dependent on machine processing power
  solutions += aRandomlyGeneratedSolution( P, V ) // initial random solutions
}
while !stopped{ // stopped is set by the RPM (explained in the next section)
  ordered_solutions = orderByDecreasingUtility( solutions )
  b = ordered_solutions[0] // b is the best solution
  if b.utility > self.bestSolution.utility{
    self.bestSolution = b // keep track of the best solution found so far
  }
  solutions = [] // clear the existing solutions, and generate new ones ..
  for i in 0..n{
    j = random(0..n/20) // random index to the best 5% of solutions
    s = ordered_solutions[ j ]
    action = none | mutate | crossover | exchange // a random choice
    switch action{
      case mutate: s = mutate( s, V )
      case crossover:
        t = ordered_solutions[ random(0..n/20) ] // select another top solution
        s = crossover( s, t, V )
      case exchange:
        x = solvers[ random( 0..solvers.count ) ] // choose a random solver
        s = x.bestSolution // request the best solution found by that solver
    }
    solutions += s
  }
}
```

The next section outlines the complete ABS model for the ETICS service, and in particular how the concurrent GA processes are managed (e.g. triggered & stopped) and how solutions are applied.

[7] All the code presented in this deliverable is *pseudo code.*

4. ABS Model

The following subsections give a high-level outline of the ABS model. As a design principle we have chosen to model only the active components of the system (namely the simulator, RPM and pooled VMs) as interfaces/classes, and to restrict the information exchanged between these components to ABS datatype items (the aim being to avoid exchanging class instances between distributed processing elements). Section §4.1 describes the datatype model, Section §4.2 the interface/class model, and finally Section §4.3 provides some notes on our experience of using ABS as feedback to the technical WPs.

4.1. Datatype Model

The datatype model is conceived in two parts, capturing: *i*) information about requests, SLAs and VMs, and *ii*) the input & output parameters of the DGA. These are respectively summarised (in the form of UML class diagrams) in Figures 4.1-*a* & 4.1-*b*, with brief explanations below each figure.

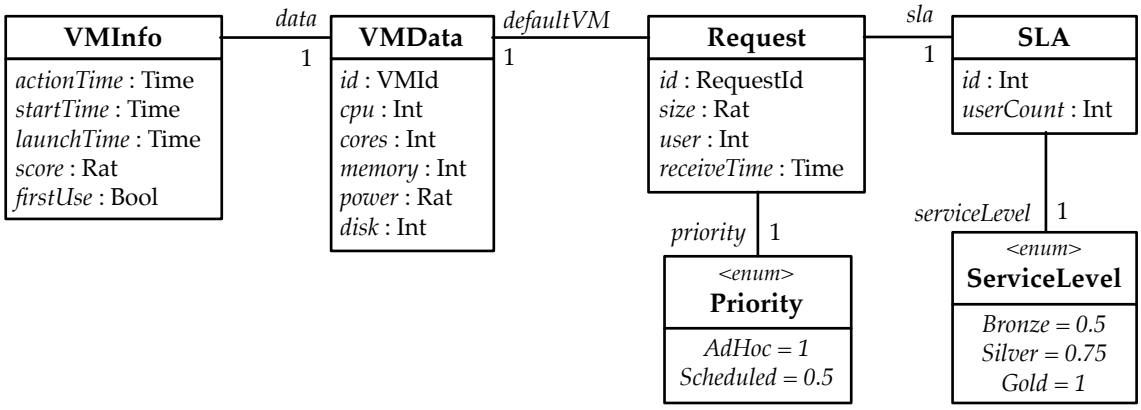

Figure 4.1-a: Requests, SLAs & VM-Related Datatypes
`Time`, **`Rat`** (= rational number), **`Bool`** & **`Int`** are all part of the core ABS language.

The **`Request`**, **`SLA`** & **`VM~`** datatypes capture most of the information required by the utility function. Their basic properties (e.g. request size & priority, VM power & disk capacity, …) have already been outlined in Section §2. A few additional notes are in order:

- The SLA datatype is for *consumer*-facing SLAs only, and all QoS properties, for both *consumer*- & *cloud-provider*-facing SLAs, are captured by global ABS functions, such as:

  ```
  def Duration vmDeployTime(VMData d) = ..
  def Duration vmUnitCost(VMData d) = ..
  def Duration requestMaxCT(Request r) = ..
  ```
  etc.

  These functions in turn depend on a handful of global constants (one for each of the constants defined in Appendix A), such as:

  ```
  def Rat global_du = 60  // Unit Time Period constant, δμ, for VMs (see Table A.1.3-a)
  def Rat global_kCT = 1  // Max. Completion-Time constant, κCT (see Table A.2.3-a)
  ```
  etc.

- **`Request`**, **`VMData`** & **`SLA`** datatypes all carry unique integer identifiers (**`VMId`** & **`RequestId`** are type-aliases for **`Int`**), which serve just to simplify the modelling of DGA parameters (below);
- End-users are minimally modelled just by integer identifiers, i.e. the **`Request.user`** property, the values of which are limited by **`SLA.userCount`**.
- The **`VMData`** datatype captures the information about VMs that is published by cloud providers (i.e. describing the essential VM capabilities). **`VMInfo`**, instead, captures details of deployed VM instances (e.g. the time they were launched) that are required by the utility function and DGA;
- The numeric values associated with the **`Priority`** & **`ServiceLevel`** enumeration types are as defined in Appendices A.2.2 & A.2.3 (*resp.*).

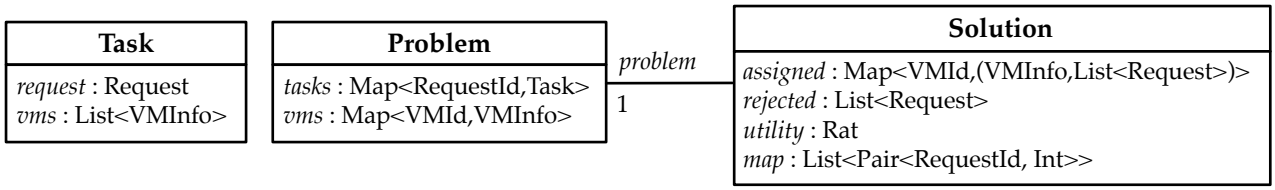

Figure 4.1-b: Input/Output Parameters for the DGA

The input/output parameters of the DGA are captured by the datatypes shown in Figure 4.1-*b*:

- The **`Problem`** datatype models the input to the DGA and consists of a list of current requests (the set $\boldsymbol{P}$ as defined in §3) and **`VMInfo`** items (corresponding to the set $\boldsymbol{M} \cup \boldsymbol{D}$ of pooled/default VMs as defined in §3).
  - The **`Task`** datatype pairs each request with the set of VMs that can process that request (recall from §3 that requests can only be processed by VMs with sufficient disk capacity).
- The **`Solution`** datatype captures the output of the DGA, and represents a possible solution to the assignment task, with the following properties:
  - **`assigned`** is a set of assignments of the (reversed) form $\boldsymbol{m} \leftarrow \boldsymbol{p_1}, \boldsymbol{p_2}, \boldsymbol{..}$, where $\boldsymbol{p_1}, \boldsymbol{p_2}, \boldsymbol{..}$ is an ordered sequence of requests and $\boldsymbol{m}$ is either a pooled VM or the default VM for $\boldsymbol{p}$;
  - **`rejected`** is list of rejected requests, those not assigned to any VM;
  - **`utility`** is the value calculated by the utility function for the solution;
  - **`map`** is an implementation specific detail facilitating the mutation & crossover operators;
  - **`problem`** is a back reference to the **`Problem`** instance that spawned the **`Solution`**;
- The use of **`Maps`** (key/value dictionaries), as opposed to flats **`Lists`**, in the **`Problem`** & **`Solution`** datatypes serves just to simplify the implementation.

4.2. Class Model

Figure 4.2-*a* shows a UML class diagram for the active components of the ABS model ~ primarily the RPM, pooled VMs and encompassing simulator. These components are explained in the subsections below (next page).

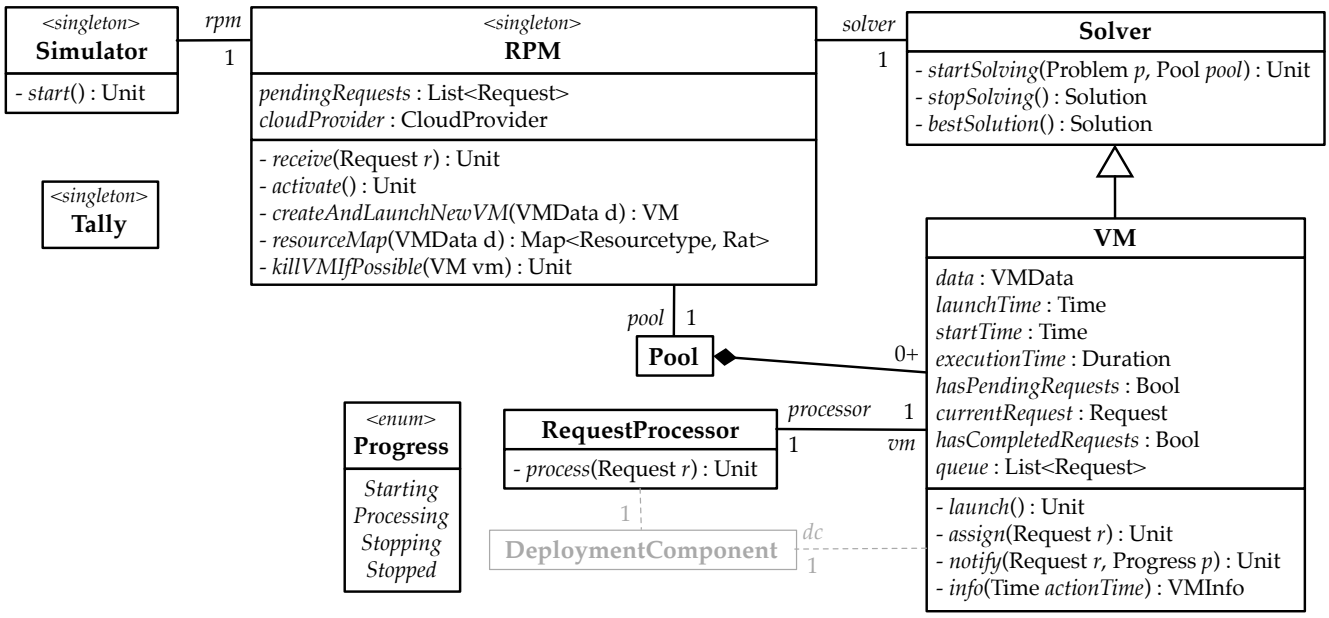

Figure 4.2-a: ETICS (Simulator) Components

`Unit` (= null/void), **`Duration`**, **`DeploymentComponent`** & **`CloudProvider`** are all part of the core ABS language. **`DCDescriptor`** is a type-alias for **`Map<Resourcetype, Rat>`**, used in ABS to describe **`DeploymentComponent`** capabilities.

Simulator

The entry point to the model is the **Simulator** class[8], with a single method **start()**, which simply posts a fixed number of random requests at random times to the **RPM**. In pseudo code:

```
start(){
  RPM rpm = new RPM()
  Int count = 0
  while (count++ < 500){ // 500 is the (arbitrary) number of requests to generate
    Request r = // create a random request
    sleepUntil( r.receiveTime ) // do nothing until it's time to send the request
    rpm.receive( r ) // send the request to the RPM
  }
}
```

The **RPM** adds all the requests it receives to its internal **pendingRequests** queue, from where they are passed in batches, at regular intervals, to the DGA - as will be described in more detail shortly.

Solver

The DGA is realised as a collection of instances of the **Solver** class, which exposes 3 methods:

- **startSolving(Problem p, Pool pool)**: this method implements the basic GA mechanism detailed earlier in §3;
  - In principle, the **Problem** argument, **p**, should carry all the information necessary for the DGA. We require, however, that arbitrary **Solver** instances exchange solutions, and the most simple way to achieve this is just to give each **Solver** object direct access to the others. Hence the 2nd argument, **pool**, which carries the pooled **VM** objects (each of which is a **Solver** object). Note, however, that this approach runs contrary to our stated design principles (see §4 intro.), and so remains unsatisfactory[9].
- **stopSolving()**: invoked by the **RPM** to stop the DGA (*cf.* the role of the **stopped** flag in the code excerpt in §3). For convenience, this method also returns the best solution (next bullet);
- **bestSolution()**: returns the best solution found (since the last invocation of **startSolving**). This method is invoked by other **Solver** objects in order to share solutions.

VM & Processor

The **VM** class, as just noted, extends the **Solver** class, such that each **VM** object forms part of the DGA. As outlined in §1, each **VM** object also maintains its own **queue** of **Request** instances, where **Requests** are assigned to a **VM** (by the **RPM**) via its **assign(Request r)** method. The **VM** instance sends these queued **Requests** one at a time, in FIFO order to the **process(Request r)** method of an attached, but asynchronous **Processor** object, which simulates request processing, and notifies the **VM** of its progress, as follows:

```
process(Request r){
  Duration dXT = .. // the request execution time (defined in Appendix A.2.2)

  vm!notify(r, Starting) // "x!foo()" is ABS syntax for an asynchronous invocation
  sleepForTime(dXT / 2) // setup phase

  vm!notify(r, Processing)
  sleepForTime(dXT) // request execution phase

  vm!notify(r, Stopping)
  sleepForTime(dXT / 2) // cleanup phase

  vm!notify(r, Stopped)
}
```

[8] All the UML classes in Figure 4.2-*a* are modelled as ABS interfaces, with unique class implementations.

[9] A more realistic approach would be to implement a remote peer-to-peer message exchange protocol, and inform each **Solver** object of just the addresses (e.g. URLs) of its peers.

Each **VM** object is intended to represent a VM purchased from cloud provider. In ABS, however, VMs are represented by **DeploymentComponent** objects, obtained opaquely from **CloudProvider** objects. This opaqueness prohibits us from defining the **VM** class as **DeploymentComponent** extension, so we must instead conceive the **VM** object as a kind of utility process running on a **DeploymentComponent**. This relationship is established by the **RPM** when the **VM** object is created (explained below). The **VM**'s **launch** method performs additional necessary setup (e.g. ensuring that the **Processor** runs in the same **DeploymentComponent** instance), and must be invoked immediately after the **VM** is created[10]. The remaining **VM** methods primarily support the utility function and are not covered here.

RPM

The **RPM** manages the pool of **VM**/**DeploymentComponent** instances, and controls the overall activity of the DGA: creating new **Problem** instances, distributing these to the pooled **VM** (**Solver**) processes, collecting the results, and implementing the best **Solution**. It does this in a continuous loop, effected by recursive invocation of its **activate()** method ~ roughly (ignoring many fine details) as follows:

```
activate(){
  // stop the (previous run of the) DGA & get the best solution ..
  // note: "await x!foo()" is ABS syntax for a blocking asynchronous invocation
  Solution best = await this.solver!stopSolving() // the RPM's solver
  for vm in pool{
    Solution s = await vm!bestSolution() // the distributed solvers
    if s.utility > best.utility{ best = s }
  }
  // implement the best solution ..
  // 1. assign requests to VMs (creating the VMs as necessary) ..
  for (vm_info, requests) in best.assigned.values{
    VM vm = // the pooled vm identified by vm_info
    if vm == NULL{ // vm_info identifies a default (not yet launched) VM
      vm = createAndLaunchNewVM( vm_info ) // described below
    }
    for r in requests{
      vm.assign(r) // see the VM section above
    }
  }
  // 2. reject rejected requests ..
  for request in best.rejected{
    // reject the request (delegated to the Tally class, below)
  }
  // if there are pending requests, create a new problem ..
  if !pendingRequests.isEmpty{
    List<Task> tasks = // create a 'task' for each pending request
    Map<VMId, VMInfo> vmis = // combined list of pooled & default VMs
    Problem problem = Problem( tasks, vmis )
    // start the DGA ..
    this.solver!startSolving( problem, pool ) // the RPM's solver
    for vm in pool{
      vm!startSolving( problem, pool ) // the distributed solvers
    }
  }
  // wait for a fixed 'action-time' (AT), and repeat ..
  sleepForDuration( global_AT )
  await this!activate()
}
```

Note that this loop repeats with a regular period given by the *action-time* constant **global_AT**, which has significance for the utility function (refer back to §2, e.g. Figure 2-*a*).

[10] This additional setup involves asynchronous calls and timed delays, which cannot be used in an object's initialiser.

The **createAndLaunchNewVM** method, as its name implies, creates & launches a new VM instance:

```
VM createAndLaunchNewVM(VMData data){
  // create a DeploymentComponent (DC)
  // for simplicity we can assume a single cloud provider
  Map<Resourcetype, Rat> rs = this.resourceMap( data )
  DeploymentComponent dc = this.cloudProvider.launchInstance( rs )
  // create a new VM instance on the DeploymentComponent ..
  [DC: dc] VM vm = new VM( data ) // as per the ABS syntax
  vm!launch() // see the VM section above
  this.pool += vm // add the VM to the pool
  this!killVMIfPossible( vm ) // explained below
  return vm
}
```

As well as deciding when to create new **VM** instances, the RPM also decides when to kill them. This is achieved by the **killVMIfPossible** method invoked in the previous code snippet. Recall from §2 that VMs are paid for by the hour. Trivially, therefore, the **killVMIfPossible** just checks in every hour to see whether the VM is being used, and if not kills it:

```
Unit killVMIfPossible(VM vm){
  sleepForDuration( global_du ) // global_du = 60 minutes
  // is the VM in use ?
  Request currentRequest = await vm!currentRequest()
  Bool hasPendingRequests = await vm!hasPendingRequests()
  if (currentRequest == NULL && !hasPendingRequests){
    // kill the VM ..
    this.pool -= vm // remove the VM from the pool
    DeploymentComponent dc = await vm!dc
    dc.release() // we no longer need the DC
    this.cloudProvider.killInstance( dc )
  }else{
    this!killVMIfPossible( vm ) // check again in 1 hour
  }
}
```

These two methods, **createAndLaunchNewVM** & **killVMIfPossible**, driven respectively by the DGA and a periodic timer, encapsulate the **RPM**'s dynamic & elastic management of the VM pool.

Tally

The final class shown in Figure 4.2-*a* is the (singleton) **Tally** class, whose function is to keep track of the overall progress of the simulation and to maintain a cumulative tally of *actual* (as opposed to *expected*) profit. To this end the **Tally** class exposes methods (not shown in Figure 4.2-*a*) to receive various progress notifications from the other components - such as:

```
Unit simulationStarted()
Unit rpmReceivedRequest(Request r)
Unit vmEnqueuedRequest(VM_info vmi, Request r)
Unit vmExecutingRequest(VM_info vmi, Request r)
```
and so on for all significant simulation events ..

The calculation of actual profit mirrors that for expected profit, but operates *post-fact* on observed, rather than predicted values (e.g. using data automatically collated by the **CloudProvider** instance on VM usage). As part of this function, the **Tally** class also maintains a historical record of requests (& their outcomes) for individual end-users, which record provides the information necessary for computing failure-rates and failure-rate penalties.

This concludes the outline of the ABS model for the ETICS simulation. A complete listing of the ABS code, as noted earlier, is provided in the zipped file accompanying this deliverable.

4.3. Experience With ABS

In D4.4.1 (§3.6) we listed some of the difficulties we encountered in developing the initial ABS model for the ENG Case Study. Most of the issues still hold[11], the most significant being:

- A lack of comprehensive and up-to-date documentation. There is now an official URL for ABS documentation (**`docs.abs-models.org`**) which has proven useful, but remains incomplete and in parts incorrect. Of particular relevance to this deliverable, for example: the **`CloudProvider`** class is undocumented, and the description of the **`DeploymentComponent`** class is incorrect.
- The requirement to use different syntactic forms for blocking method calls dependent on whether sender & receiver are located in the same or different COGs (see D4.4.1, §3.6 for an explanation). We view this as a major potential source of run-time errors.

In light of continued experience with ABS, we also add the following issues:

- No floating point numbers. Only integer (**`Int`**) are rational (**`Rat`**) numbers are available, where a rational is expressed as ratio of integers. So, for example, **`0.025`** must be written as **`25/1000`**.
- The restriction to **`while`** statements for iteration, combined with the functional nature of the basic collection types (**`List`**, **`Set`**, **`Map`**) leads to verbose code - e.g. to iterate through a **`List`** in ABS:

```
List<X> i = list;
while (i != Nil){
  X x = head(list);
  // do something with x
  i = tail(list);
}
```

  Compare this to the Java (enhanced **`for`** loop) syntax:

```
for (X x : list){
  // do something with x
}
```

- Limited expressivity of ABS function declarations. As a specific case in point, the present ABS model makes use of the quicksort[12] algorithm (e.g. for sorting lists of GA solutions by utility, §3). In Haskell this algorithm has a fairly concise definition[13]:

```
qsort [] = []
qsort (head:tail) = (qsort lesser) ++ [head] ++ (qsort greater)
  where
    lesser = filter (< head) tail
    greater = filter (>= head) tail
```

  We were unable, however, to translate this definition into ABS functions, and so resorted to a longer imperative implementation (using class methods). Moreover, our model applies quicksort over two lists of different types and with different comparison operators, but we could find no succinct modular way to realise this, preferring in the end to code the complete algorithm twice.

On the whole, although ABS is not particularly difficult to use (once you have grasped the basic principles), we feel that it never-the-less misses some 'basic' features (e.g. floating-point numbers, **`for`** loops, list/map subscripts) and has limited support for modularity (as in the quicksort case above, or the fact that there is no class inheritance, and hence no calls to super methods). In general, while we understand the need to constrain the language (to support formal analysis), these constraints result in code that is overly verbose. The point here is not that conciseness *per se* is good, or that verbosity *per se* is bad. Rather it is just that an ABS *model* of a system can easily end up with significantly more lines of code than are required (in Java 8 or Swift, say) to implement that system[14].

---

[11] The issues reported in D4.4.1 that *have* been addressed are: *i*) it is no longer necessary to assign the results of method calls to variables, and *ii*) nested method calls are now also permitted.

[12] E.g. see https://en.wikipedia.org/wiki/Quicksort

[13] Even more concisely (but harder to read): `qsort (h:t) = qsort [x | x<-t, x<h] ++ [p] ++ qsort [x | x<-t, x>=h]`

[14] This is, for example, why we chose *not* to use ABS-style pseudo code for the examples in previous sections.

5. Summary

This deliverable refines and expands the specification of the ENG Case Study originally presented in D.4.4.1, focusing in particular on resource and deployment scenario modelling. Resources are virtual machines (VMs) purchased on a pay-per-use basis from third-party cloud providers. A detailed set of modelling requirements for VMs has been formulated (Appendix A), covering their life-cycle, defining characteristics and QoS constraints (as derived from cloud-provider SLAs). The functional (non QoS) aspects of these requirements have already been incorporated into the ABS resource model (WP1). VMs are deployed (i.e. purchased, configured & subsequently destroyed) dynamically, in a scalable and elastic fashion, in response to the quantity and type of end-user service requests. In the business context of the case study, deployment decisions are based entirely on operational criteria - specifically: a utility/profit function which takes into account the income received from end-users, the cost of VMs, and SLA-specified penalties for violation of QoS constraints. These operational aspects have also been formally specified (Appendix A), and *all* the formal specifications have been successfully implemented in ABS, with the decision process realised by a distributed genetic algorithmic (DGA).

The relation of the ENG Case Study to the project objectives has already been described in D4.4.1. This deliverable primarily addresses objective O2 (*"Behavioural Specification Language for Virtualised Resources"*) - and in particular provides input to and incorporates results from T1.1 (modelling support for scalable infrastructure - *esp.* dynamic creation & management of VMs), T1.2 (modelling of resources) and T1.3 (deployment modelling). The formal SLA specifications given in this deliverable (together with their application in the decision making process) are also relevant to objective O3 (*"Design-by-Contract Methodology for Service Contracts"*), and in particular provide input to T2.2 (*w.r.t.* the specification of, and conformance criteria for, QoS). We have also noted (§4.3) specific issues, based on our experience of coding in ABS, with the language *per se*.

For the remainder of the project, the ENG Case Study will focus on:

- Objective O4 (*"Model Conformance Demonstrator"*): automatic generation of executable Java code[15] and conformance checking of this code against the formal semantics of the ABS source. In D4.4.1 we also stated an intention to employ the Envisage tools for automatic *test-case generation* (TCG). It turns out, however, that the TCG tools cannot be used on code that contains rational number (`Rat`) types, and so we are unable to employ TCG.
- Objective O5 (*"Model Analysis Demonstrator"*): verification of the ABS model *w.r.t.* non-functional requirements, with the general aim to ensure the *scalability* and *cost-effectiveness* of the final implemented system. The most immediate concerns, as already noted in §1, are:
  - *To ensure (if possible) that the DGA is deadlock free*;
  - *To ensure that the DGA scales*;
  - *W.r.t. business objectives, to identify appropriate/optimal values for the key system parameters*[16].

The results of this ongoing work will be presented in D4.4.3.

[15] At the time of writing, the Java code generation tool does not yet fully support ABS resource models.

[16] Namely: the 'action-time', δ_{AT} (Appendix A.2.1), and the global constants χ_ρ, κ_{CT}, χ_{CT} and χ_{FR} (described in Appendix A.2.3).

A. Formal Definitions

This appendix presents formal definitions for the following (detailed in §A.1 to §A.5 *resp.*):

- *Virtual Machines (VMs)*
- *Requests*
- *Utility (Profit/Loss)*
- *The Assignment Task*
- *The Default VM for a Request*

Table A-*a* lists various abbreviations, symbols & functions used in this appendix:

Table A-a: Abbreviations, Symbols & Functions

Abbreviations	
RPM : Resource Pool Manager	EE : Execution Engine

Symbols (for variable types)		
τ : a point in time	δ : a duration	ε : an error or variance
κ : a constant	λ : a count or ratio	$€, \chi$: monetary values
μ : a virtual machine	ρ : a process	

Functions	
$\lvert S \rvert$: number of elements in set S	$\lceil x \rceil$: ceiling of x

A.1. Virtual Machines

This section presents formal definitions of the following:

- *VM Life-Cycle* (§A.1.1)
- *VM Types* (§A.1.2)
- *SLAs Governing VMs* (§A.1.3)

A.1.1. VM Life-Cycle

The life-cycle of a VM is captured by the state diagram in Figure A.1.1-*a*, with key triggers, times and durations listed in Table A.1.1-*a* below:

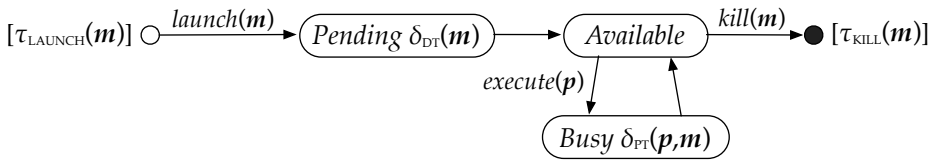

Figure A.1.1-a: Life-Cycle of VM

States (ovals) display both state names & (if applicable) variables, δ, giving the duration of the state.

Table A.1.1-a: VM Life-Cycle Triggers, Times & Durations

State Transition	*Trigger*	*(**m** is a VM, **p** is a process running on **m**)*	*Occurs at*
○ → *Pending*	*launch*(**m**) :	Creates a new VM, **m**.	$\tau_{\text{LAUNCH}}(m)$
Pending → *Available*	$\tau_{\text{LAUNCH}}(m) + \delta_{\text{DT}}(m)$:	*Pending* lasts until **m** is deployed & ready to be used: ending after a *deploy-time* delay, $\delta_{\text{DT}}(m)$.	$\tau_{\text{POOL}}(m)$
Available → *Busy*	*execute*(**p**,**m**) :	Executes process **p** on **m**.	*t*
Busy → *Available*	$t + \delta_{\text{PT}}(p,m)$:	*Busy* lasts while **m** executes process **p**: ending after (a machine & process dependent) *processing-time* $\delta_{\text{PT}}(p,m)$.	~
Available → ●	*kill*(**m**) :	Destroys **m**.	$\tau_{\text{KILL}}(m)$

A.1.2. VM Types

A VM type is completely defined by the properties listed in Table A.1.2-*a*. Any given VM type may have multiple instances[17].

Table A.1.2-a: Properties of VMs

Property	*Definition (**m** is a VM)*	*Units*	*Permitted Values*
	Basic Properties		
CPU Clock Speed	$\mu_{\text{CLK}}(\boldsymbol{m})$	GHz	1.0, 1.25, 1.5
CPU Cores	$\mu_{\text{CORES}}(\boldsymbol{m})$	~	1, 2, 4, 8
Memory Capacity	$\mu_{\text{MEM}}(\boldsymbol{m})$	GB	1, 2, 4, 8
Hard Disk Capacity	$\mu_{\text{DISK}}(\boldsymbol{m})$	GB	1, 2, 4, 8, 16, 32
	Derived Properties		
Processing Power	$\mu_{\text{POW}}(\boldsymbol{m}) = \mu_{\text{CLK}}(\boldsymbol{m}) \cdot \mu_{\text{CORES}}(\boldsymbol{m}) + 0.6 \cdot \mu_{\text{MEM}}(\boldsymbol{m})$	~	1.6 *to* 16.8
Machine "Spec."	$\mu_{\text{SPEC}}(\boldsymbol{m}) = \mu_{\text{POW}}(\boldsymbol{m}) + \mu_{\text{DISK}}(\boldsymbol{m})$	~	2.6 *to* 48.8

Number of distinct μ_{POW} values $= 3 \times 4 \times 4 = 48$; Number of distinct μ_{SPEC} values $= 48 \times 6 = 288$

- The constant 0.6 in the μ_{POW} term is chosen such that μ_{POW} is comparable to the 'ECU' (EC2 Compute Unit) measure employed by Amazon EC2[18].

Loosely speaking, a VM with $\boldsymbol{n}$ times the processing power of another will execute a given task $\boldsymbol{n}$ times as fast ~ i.e. for any process, $\boldsymbol{p}$, executing on machines $\boldsymbol{m_1}$, $\boldsymbol{m_2}$:

$$\mu_{\text{POW}}(\boldsymbol{m_1}) \cdot \delta_{\text{PT}}(\boldsymbol{p},\boldsymbol{m_1}) \approx \mu_{\text{POW}}(\boldsymbol{m_2}) \cdot \delta_{\text{PT}}(\boldsymbol{p},\boldsymbol{m_2}) \qquad (\sharp 1)$$

The μ_{SPEC} term determines the pricing for VMs as described in the next two sections.

A.1.3. SLAs governing VMs

The SLA terms governing VMs are encoded by the properties listed in Table A.1.3-*a*.

Table A.1.3-a: SLA Terms for VMs

Property	*Symbol/Definition (**m** is a VM)*	*Units*
	Global Constants	
Unit Time Period	$\delta_{\mu} = 60$	*mins*
	Derived Properties	
Cost per Unit Time Period	$\chi_{\mu}(\boldsymbol{m}) = 0.00266 \cdot \mu_{\text{SPEC}}(\boldsymbol{m})$	€
Deploy-Time	$\delta_{\text{DT}}(\boldsymbol{m}) = 0.2 \cdot \mu_{\text{SPEC}}(\boldsymbol{m})$	*mins*

- The constant 0.00266 in the χ_{μ} term is chosen such that pricing is comparable to the general purpose, on-demand machines offered by Amazon EC2: Tables A.1.3-*b* and A.1.3-*c* (next page) respectively show an actual Amazon pricing sample[18] and comparable χ_{μ} values.
- The constant 0.2 in the δ_{DT} (deploy-time) term is chosen to give values of δ_{DT} of up to 10 minutes[19], as illustrated in Table A.1.3-*d* (next page).
  - In D4.4.1 a maximum deploy-time guarantee was also included in provider-facing SLAs. This has been removed as an unnecessary complication.

[17] For simulation purposes, each VM instance is distinguished by a unique identifier.

[18] See https://aws.amazon.com/ec2

[19] Based loosely on: http://www.philchen.com/2009/04/21/how-long-does-it-take-to-launch-an-amazon-ec2-instance

Table A.1.3-b: Sample Price List for Amazon EC2 General Purpose, On-Demand Instances

VM Type	*vCPU*	*ECU*	*Memory* (GiB)	*Storage* (GB)	***$ per hour***
t2.micro	**1**	variable	**1**	EBS	**0.013**
t2.small	**1**	variable	**2**	EBS	**0.026**
t2.medium	**2**	variable	**4**	EBS	**0.052**
m3.medium	**1**	3.0	**3.75**	1×4 SSD	**0.070**
m3.large	**2**	**6.5**	**7.5**	1×32 SSD	**0.140**
m3.xlarge	**4**	**13.0**	**15**	**2×40 SSD**	**0.280**
m3.2xlarge	**8**	**26.0**	**30**	**2×80 SSD**	**0.560**

ECU = elastic compute unit; EBS = elastic block storage; ***bold*** *values correspond to entries in Table 1.1.3-c (below)*

Table A.1.3-c: Price List Based on the Definitions Given in A.1.2 & A.1.3

$\mu_{CLK} \cdot \mu_{CORES}$	μ_{MEM}	μ_{POW}	μ_{DISK}	μ_{SPEC}	χ_μ	*$ equiv.**
1	**1**	1.6†	2.7†	4.30	0.011	**0.013**
1	**2**	2.2†	6.4†	8.60	0.023	**0.026**
2	**4**	4.4†	12.9†	17.30	0.046	**0.052**
1	**3.75**	3.25‡	20!	23.25	0.062	**0.070**
2	**7.5**	**6.5**	40‡	46.50	0.124	**0.140**
4	**15**	**13.0**	**80**	93.00	0.247	**0.280**
8	**30**	**26.0**	**160**	186.00	0.495	**0.560**

** calculated at $1 = 0.88 €; † values are "elastic" in EC2; ‡ approximates EC2; ! only 4 GB in EC2*

Table A.1.3-d: Calculated Deploy-Times for a Sample of VM Types

μ_{CLK}	μ_{CORES}	μ_{MEM}	μ_{POW}	μ_{DISK}	μ_{SPEC}	δ_{DT}	
1.00	1	1	1.6	1	2.6	**0.52**	*
1.00	2	1	2.6	2	4.6	**0.92**	
1.25	2	2	3.7	4	7.7	**1.54**	
1.25	4	2	6.2	8	14.2	**2.84**	
1.25	8	4	12.4	16	28.4	**5.68**	
1.50	2	4	5.4	32	37.4	**7.48**	
1.50	4	8	10.8	32	42.8	**8.56**	
1.50	8	8	16.8	32	48.8	**9.76**	†

** and † respectively correspond to the lowest & highest possible machine specs.*

VM Pricing & Billing

Virtual machines are billed throughout the *entire* period they exist (i.e. τ_{LAUNCH} to τ_{KILL}), according to the formulas shown in Table A.1.3-*e*. The *minimum* cost for a VM is χ_μ (the case that $\boldsymbol{t} - \tau_{LAUNCH} < \delta_\mu$).

Table A.1.3-e: VM Costs

Property	*Definition (**m** is a VM)*	*Units*
VM Cost to Time $\boldsymbol{t}$ $(> \tau_{LAUNCH})$	${}^{t}€_\mu(\boldsymbol{m},\boldsymbol{t}) = \chi_\mu(\boldsymbol{m}) \cdot \lceil (\boldsymbol{t} - \tau_{LAUNCH}(\boldsymbol{m})) / \delta_\mu \rceil$	€
Total VM Cost	$€_\mu(\boldsymbol{m}) = {}^{t}€_\mu(\boldsymbol{m}, \tau_{KILL}(\boldsymbol{m}))$	€

A.2. Requests

This section presents formal definitions for the following:

- *Request Life-Cycle* (§A.2.1)
- *Request Types* (§A.2.2)
- *SLAs Governing Requests* (§A.2.3)

A.2.1. Request Life-Cycle

The life-cycle of a request is given by the state diagram in Figure A.2.1-*a*, described in Table A.2.1-*a* (the durations, δ_{AT}, δ_{TO}, δ_{QT}, δ_{PRE}, δ_{XT} and δ_{POST}, are explained in more detail in later sections).

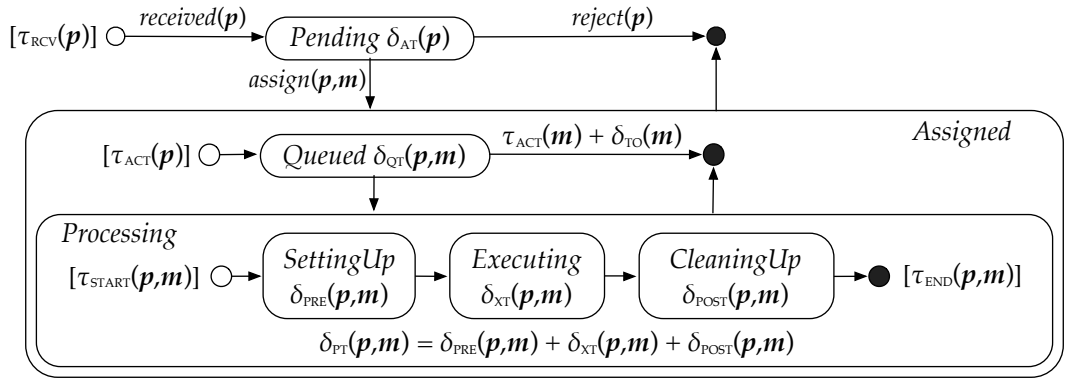

Figure A.2.1-a*: *Life-Cycle of a Request
States (ovals) display both state names & (if applicable) variables, δ, giving the duration of the state.

Table A.2.1-a: Request Life-Cycle Triggers, Times & Durations

State Transition	*Trigger*	*(**p** is a request, **m** is a VM)*	*Occurs at*
○ → *Pending*	*received*(**p**) :	the RPM receives a new request, **p**.	$\tau_{RCV}(\boldsymbol{p})$
Pending → ●	*reject*(**p**) :	*Pending* lasts for the duration $\delta_{AT}(\boldsymbol{p})$ [*action-time*], ending when the RPM decides to either: ‣ *reject* **p** (ending its life-cycle); ‣ or *assign* **p** to machine **m** for processing: - **p** is added to **m**'s request queue; - **p** transitions to *Assigned.Queued*.	$\tau_{ACT}(\boldsymbol{p})$
Pending → *Queued*	*assign*(**p**,**m**) :		
Queued → *SettingUp*	$\tau_{ACT}(\boldsymbol{m}) + \delta_{QT}(\boldsymbol{p},\boldsymbol{m})$:	*Queued* lasts for the duration $\delta_{QT}(\boldsymbol{p},\boldsymbol{m})$ [*queuing-time*], ending when either: ‣ **m** becomes *Available* to process **p** - **p** transitions to *Processing.SettingUp*; - **m** transitions to *Busy*; ‣ or the *queuing-time*, exceeds a maximum *time-out* limit, δ_{TO}, i.e. $\delta_{QT}(\boldsymbol{p},\boldsymbol{m}) > \delta_{TO}(\boldsymbol{p},\boldsymbol{m})$: - **p** is rejected, ending its life-cycle;	$\tau_{START}(\boldsymbol{p},\boldsymbol{m})$
Queued → ●			~
SettingUp → *Executing*	$\tau_{START}(\boldsymbol{p},\boldsymbol{m}) + \delta_{PRE}(\boldsymbol{p},\boldsymbol{m})$:	*SettingUp* ends automatically after a *set-up* (pre-processing) delay, $\delta_{PRE}(\boldsymbol{p},\boldsymbol{m})$	***t***
Executing → *CleaningUp*	$\boldsymbol{t} + \delta_{XT}(\boldsymbol{p},\boldsymbol{m})$:	*Executing* ends automatically after an *execution-time* delay, $\delta_{XT}(\boldsymbol{p},\boldsymbol{m})$ ‣ marks the *time-of-completion*, τ_{CT}, of **p**;	$\tau_{CT}(\boldsymbol{p})$
CleaningUp → ●	$\tau_{CT}(\boldsymbol{p},\boldsymbol{m}) + \delta_{POST}(\boldsymbol{p},\boldsymbol{m})$:	*CleaningUp* ends automatically after a *clean-up* (post-processing) delay, $\delta_{POST}(\boldsymbol{p},\boldsymbol{m})$ ‣ **m** then transitions back to *Available*;	$\tau_{END}(\boldsymbol{p})$

The following sub-sections specify the *completion-time*, δ_{CT}, & *queuing-time*, δ_{QT}, of requests, and give a formal definition of request *failure-rate*, λ_{FR}.

Request Completion-Time

A request is considered *complete* (or *satisfied*) only if it is has finished executing, such that: we define the '*completion-time*', δ_{CT}, of a request to be the duration from its initial receipt (at τ_{RCV}) to the end of its execution (at τ_{CT}, ignoring the clean-up time, δ_{POST})[20]:

$$\delta_{CT}(\boldsymbol{p},\boldsymbol{m}) = \delta_{AT}(\boldsymbol{p}) + \delta_{QT}(\boldsymbol{p},\boldsymbol{m}) + \delta_{PRE}(\boldsymbol{p},\boldsymbol{m}) + \delta_{XT}(\boldsymbol{p},\boldsymbol{m})$$

- For simplicity, we will define both δ_{PRE} & δ_{POST} as a fixed ratios of δ_{XT}[21]:

$$\delta_{PRE}(\boldsymbol{p},\boldsymbol{m}) = \delta_{POST}(\boldsymbol{p},\boldsymbol{m}) = 0.5 \cdot \delta_{XT}(\boldsymbol{p},\boldsymbol{m})$$

  .. such that the *completion-time* formula reduces to:

$$\delta_{CT}(\boldsymbol{p},\boldsymbol{m}) = \delta_{AT}(\boldsymbol{p}) + \delta_{QT}(\boldsymbol{p},\boldsymbol{m}) + 1.5 \cdot \delta_{XT}(\boldsymbol{p},\boldsymbol{m}) \quad (\#2)$$

In contrast note that the total *processing-time*, δ_{PT} (see Table A.1.1-*a*), for a request is just the sum of the *set-up*, *execution* & *clean-up* delays (the duration from τ_{START} to τ_{END}):

$$\delta_{PT}(\boldsymbol{p},\boldsymbol{m}) \;=\; \delta_{PRE}(\boldsymbol{p},\boldsymbol{m}) + \delta_{XT}(\boldsymbol{p},\boldsymbol{m}) + \delta_{POST}(\boldsymbol{p},\boldsymbol{m}) \;=\; 2 \cdot \delta_{XT}(\boldsymbol{p},\boldsymbol{m}) \quad (\#3)$$

Figure A.2.1-*b* summarises the time-line (time points & durations) for *completed* requests:

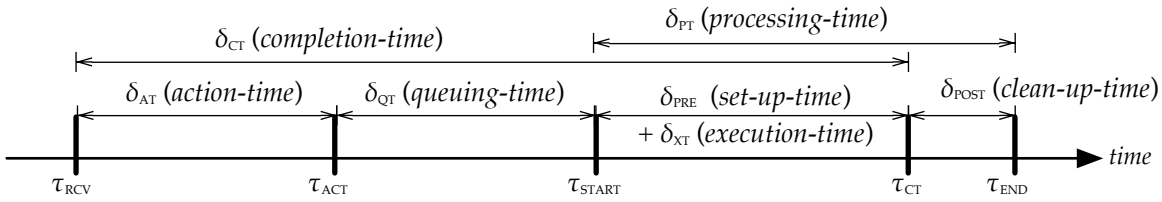

Figure A.2.1-b: Timeline for a Completed Request

Request Queuing-Time

As stated in Table A.2.1-*a*, when a request, $\boldsymbol{p}$, is assigned to a VM, $\boldsymbol{m}$, it is immediately added to $\boldsymbol{m}$'s request queue (i.e. transition: *Pending*→*Queued*). This queue may already contain other '*prior*' requests ~ those previously assigned to $\boldsymbol{m}$ but not yet executed ~ each of which has FIFO precedence over $\boldsymbol{p}$:

$$\text{Let } Q(\boldsymbol{p},\boldsymbol{m}) = \text{the set of all requests on } \boldsymbol{m}\text{'s request queue that arrived before } \boldsymbol{p}\text{, i.e.} \quad (\#4)$$
$$\forall\, \boldsymbol{r} \in Q(\boldsymbol{p},\boldsymbol{m}) : \tau_{ACT}(\boldsymbol{r}) < \tau_{ACT}(\boldsymbol{p})$$

Assuming for the moment that a request, $\boldsymbol{p}$, does *not* get timed-out, then if $\boldsymbol{p}$ is assigned to VM $\boldsymbol{m}$, the duration $\boldsymbol{\delta}_{QT}$ depends on *i*) the set $Q(\boldsymbol{p},\boldsymbol{m})$ of prior requests assigned to $\boldsymbol{m}$ and *ii*) the state of $\boldsymbol{m}$:

- First, all the requests in $Q(\boldsymbol{p},\boldsymbol{m})$ must be processed before $\boldsymbol{p}$ is processed:
  - If $\boldsymbol{p'}$ is the prior request (if any) immediately preceding $\boldsymbol{p}$. Then the time that $\boldsymbol{p}$ waits on the queue is at least the time that $\boldsymbol{p'}$ waits plus the *processing-time* for $\boldsymbol{p'}$:
    - $\boldsymbol{\delta}_{QT}(\boldsymbol{p},\boldsymbol{m}) \geq \boldsymbol{\delta}_{QT}(\boldsymbol{p'},\boldsymbol{m}) + 2\boldsymbol{\delta}_{XT}(\boldsymbol{p'},\boldsymbol{m})$;
- Second, in order for $\boldsymbol{m}$ to process $\boldsymbol{p}$, it must be in the *Available* state, hence:
  - Let $\sigma(\boldsymbol{m})$ = the time until machine $\boldsymbol{m}$ next becomes *Available*:
    - If $\boldsymbol{m}$ is already *Available* then $\sigma(\boldsymbol{m}) = 0$;
    - If $\boldsymbol{m}$ is *Busy* processing some other request $\boldsymbol{q}$, and $\lambda(\boldsymbol{q})$ is the degree, from 0 (at τ_{START}) to 1 (at τ_{END}), to which $\boldsymbol{q}$ has been processed, then $\sigma(\boldsymbol{m}) = (1-\lambda(\boldsymbol{q})) \cdot \boldsymbol{2} \cdot \delta_{XT}(\boldsymbol{q},\boldsymbol{m})$;
    - If $\boldsymbol{m}$ is *Pending*, and if $\lambda(\boldsymbol{m})$ is the progress, from 0 (at τ_{LAUNCH}) to 1 (at τ_{POOL}), in deploying $\boldsymbol{m}$, then $\boldsymbol{\sigma}(\boldsymbol{m}) = (1-\boldsymbol{\lambda}(\boldsymbol{m})) \cdot \boldsymbol{\delta}_{DT}(\boldsymbol{m})$.

[20] i.e. this is 'completion-time' from the point-of-view of the end-user (ignoring any network latency).

[21] The constant 0.5 is arbitrary.

- Otherwise, m has not yet been created, and we don't know when it will be (or indeed if it will ever be) created, so: $\sigma(m) = \infty$.

So assuming no time-outs, then $\delta_{QT}(p,m)$ is the sum $\delta_{QT}(p',m) + 2 \cdot \delta_{XT}(p',m) + \sigma(m)$, which reduces to just $\sigma(m)$ in the case that $Q(p) = \varnothing$ (there is no prior p'). Taking the time-out into consideration, where a time-out occurs just if $\delta_{TO} < \delta_{QT}$, gives the following definition for δ_{QT}:

$$\delta_{QT}(p,m) = \begin{cases} min(\ \delta_{QT}(p',m) + 2 \cdot \delta_{XT}(p',m) + \sigma(m)\ ,\ \delta_{TO}(p,m)\ ) & \text{if } Q(p) \neq \varnothing : \\ min(\ \sigma(m)\ ,\ \delta_{TO}(p,m)\ ) & \text{otherwise} \end{cases} \quad (\#5)$$

where:

$$\sigma(m) = \begin{cases} 0 & \text{if } m \text{ is Available} \\ (1 - \lambda(q)) \cdot 2 \cdot \delta_{XT}(q,m) & \text{if } m \text{ is Busy processing } q \\ (1 - \lambda(m)) \cdot \delta_{DT}(m) & \text{if } m \text{ is Pending} \\ \infty & \text{otherwise} \end{cases}$$

where:

$\lambda(q)$ = the degree (0..1) to which process q is complete.

$\lambda(m)$ =the degree (0..1) to which deployment of VM m is complete.

The decision graph in Figure A.2.1-*c* provides another view of the various conditionals in #5.

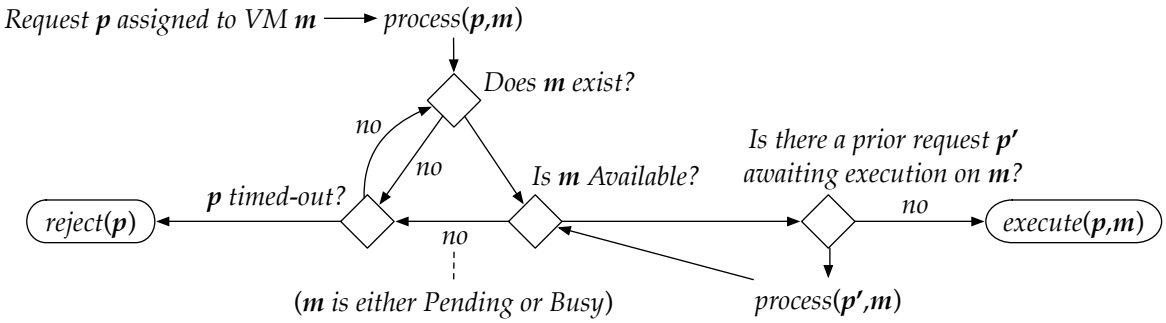

Figure A.2.1-c: Decision Graph for Assigned (Queued) Requests

Request Failure-Rate

A request, p, fails if it is rejected, either (*cf.* Figure/Table A.2.1-*a*):

- explicitly by the RPM invoking *reject*(p) [at time τ_{ACT}];
- or, when the queuing-time, $\delta_{QT}(p,m)$, for p exceeds the time-out, $\delta_{TO}(p,m)$.

The *failure-rate* for requests, *w.r.t.* a particular end-user, is defined as follows:

- First, we assume that the set of *all* requests received from a given user is totally ordered *w.r.t.* τ_{RCV} (i.e. no two requests are received at the same time from the same user), such that we can talk of consecutive fixed-length sequences of requests received from a given user:

  Let $Q(u)^i_N$ = the i^{th} sequence of N requests (i, $N > 0$) received from user u

  - i.e. if the user, u, sends a sequence of requests $p_1, p_2, p_3, p_4, \ldots$, then:

    $$Q(u)^i_N = \{\ p_{(i-1)N+1}, p_{(i-1)N+2}, \ldots, p_{Ni}\ \}$$

- Each sequence $Q(u)^i_N$ can contain 0+ failed requests:

  For *any* set, S, of requests: let $\perp(S)$ = the subset of S that *fail*.

- The *failure-rate*, $\lambda_{FR}(u)^i_N$, for the i^{th} set of n requests is just the number of failed requests in $Q(u)^i_N$:

  $$\lambda_{FR}(u)^i_N = |\perp(Q(u)^i_N)| \text{ = the } \textit{failure-rate} \text{ for the } i^{th} \text{ sequence of } N \text{ requests.} \quad (\#6)$$

A.2.2. Request Types

A Request type is completely defined by the properties listed in Table A.2.2-*a*. A given Request type may have multiple instances.

Table A.2.2-a: Properties of Requests

Property	*Symbol (**p** is a request)*		*Permitted Values*
Request Priority	$\rho_{\text{PRI}}(\boldsymbol{p})$		0.5 (*scheduled*), 1.0 (*ad-hoc*)
Request Size	$\rho_{\text{SIZE}}(\boldsymbol{p})$	*a measure of the computational complexity of the request*	1.0 *to* 32.0
Invoking End User	$\rho_{\text{USER}}(\boldsymbol{p})$	*identifies the end-user who posted the request*	~

For simplicity, we can treat $\boldsymbol{\rho_{\text{SIZE}}(p)}$ as the quantity (in GB) of data over which $\boldsymbol{p}$ operates. This data must be both copied to the VM on which $\boldsymbol{p}$ executes prior to execution, and deleted from the VM following execution. Accordingly, $\boldsymbol{\rho_{\text{SIZE}}(p)}$ determines the following:

- A minimum disk capacity, μ_{DISK}, for the VM required to execute the request:

$$\text{A request } \boldsymbol{p} \text{ requires a VM, } \boldsymbol{m}, \text{ with } \mu_{\text{DISK}}(\boldsymbol{m}) \geq \rho_{\text{SIZE}}(\boldsymbol{p}) \tag{\#7}$$

  .. and since $\mu_{\text{DISK}} \in \{ 1, 2, 4, 8, 16, 32 \}$ (§A.1.2), we also constrain ρ_{SIZE} to the interval [1,32].

- And for a given machine, $\boldsymbol{m}$:
  - The *set-up* time, $\delta_{\text{PRE}}(\boldsymbol{p},\boldsymbol{m})$ (for copying data), and *clean-up* time, $\delta_{\text{POST}}(\boldsymbol{p},\boldsymbol{m})$ (for deleting data);
  - The *execution-time*, $\delta_{\text{XT}}(\boldsymbol{p},\boldsymbol{m})$[22], which we will define as:

$$\delta_{\text{XT}}(\boldsymbol{p},\boldsymbol{m}) = \frac{\rho_{\text{SIZE}}(\boldsymbol{p}) \pm \varepsilon_{\text{XT}}(\boldsymbol{p})}{\mu_{\text{POW}}(\boldsymbol{m})} \tag{\#8}$$

  .. *where* ε_{XT} is a randomly generated value in the real interval $[0\,,0.1 \cdot \rho_{\text{SIZE}}(\boldsymbol{p})]$ ~ giving a range of values from ≈0.05 mins ($\rho_{\text{SIZE}}=1$, $\mu_{\text{POW}}=16.8$) to ≈22.0 mins ($\rho_{\text{SIZE}}=32$, $\mu_{\text{POW}}=1.6$);
    - From #3 this gives a range of *processing-times*, δ_{PT}, from ≈0.1 to ≈44.0 mins.

A.2.3. SLAs Governing Requests

The SLA terms governing requests are encoded by the basic and derived terms respectively listed in Tables A.2.3-*a* and A.2.3-*b* (next page).

Table A.2.3-a: Basic SLA Terms for Requests

Property	*Definition (**m** is a VM)*	*Units*	*Permitted Values*
	Global Constants		
Cost per Unit ρ_{SIZE}	$\chi_\rho =$ *to be determined by simulation*	€	> 0
Max. Completion-Time Constant	$\kappa_{\text{CT}} =$ "	~	> 0
Unit Cost for Completion-Time Penalties	$\chi_{\text{CT}} =$ "	€	> 0
Unit Cost for Failure-Rate Penalties	$\chi_{\text{FR}} =$ "	€	> 0
	Basic Properties		
Service Level	$\kappa_{\text{SL}}(\boldsymbol{u}) =$ *bronze*, *silver* or *gold**	€	0.5, 0.75, 1.0

**Service Levels: bronze = 0.5, silver = 0.75, gold = 1.0*

- For convenience, we introduce a term Φ to denote the product of κ_{SL} & ρ_{PRI}:

$$\text{Let } \Phi(\boldsymbol{p}) = \kappa_{\text{SL}}(\rho_{\text{USER}}(\boldsymbol{p})) \cdot \rho_{\text{PRI}}(\boldsymbol{p}) \tag{\#9}$$

[22] Assuming that more data means more computations to perform.

Table A.2.3-b: Derived SLA Terms for Requests

Property	*Definition (**p** is a request, **m** is a VM, **u** is an end-user)*	*Units*
Request Income	$€_\rho(\boldsymbol{p}) = \chi_\rho \cdot \rho_{\text{SIZE}}(\boldsymbol{p}) \cdot \Phi(\boldsymbol{p})$	€
Maximum Completion-Time	$^{m}\delta_{\text{CT}}(\boldsymbol{p}) = \dfrac{\kappa_{\text{CT}} \cdot \rho_{\text{SIZE}}(\boldsymbol{p})}{\Phi(\boldsymbol{p})}$	*mins*
Penalty for Violations of Maximum Completion-Time	$€_{\text{CT}}(\boldsymbol{p},\boldsymbol{m}) = \chi_{\text{CT}} \cdot \Delta_{\text{CT}}(\boldsymbol{p},\boldsymbol{m}) \cdot \Phi(\boldsymbol{p})$ *where:* $\Delta_{\text{CT}}(\boldsymbol{p},\boldsymbol{m}) = \begin{cases} \delta_{\text{CT}}(\boldsymbol{p},\boldsymbol{m}) - {}^{m}\delta_{\text{CT}}(\boldsymbol{p}) & \text{if } \delta_{\text{CT}}(\boldsymbol{p},\boldsymbol{m}) > {}^{m}\delta_{\text{CT}}(\boldsymbol{p}) \\ 0 & \text{otherwise} \end{cases}$	€
Maximum Failure-Rate	$^{m}\lambda_{\text{FR}}(\boldsymbol{u}) = \lceil 2/\kappa_{\text{SL}}(\boldsymbol{u}) \rceil$	~
Penalty for Violations of Maximum Failure-Rate	$€_{\text{FR}}(\boldsymbol{u})^{i}_{N} = \chi_{\text{FR}} \cdot \Delta_{\text{FR}}(\boldsymbol{u})^{i}_{N} \cdot \kappa_{\text{SL}}(\boldsymbol{u})^2$ *where:* $\Delta_{\text{FR}}(\boldsymbol{u})^{i}_{N} = \begin{cases} \lambda_{\text{FR}}(\boldsymbol{u})^{i}_{N} - {}^{m}\lambda_{\text{FR}}(\boldsymbol{u}) & \text{if } \lambda_{\text{FR}}(\boldsymbol{u})^{i}_{N} > {}^{m}\lambda_{\text{FR}}(\boldsymbol{u}) \\ 0 & \text{otherwise} \end{cases}$	€

*Recall from #6 that $\lambda_{\text{FR}}(\boldsymbol{u})^{i}_{N}$ is the failure-rate for the $\boldsymbol{i}^{th}$ sequence of **n** requests*

Request Pricing & Billing

End-users only pay for *completed* requests, the price, $€^{C}_{\rho}$, for each of which is the income, $€_\rho$, less the completion-time penalty (if any), $€_{\text{CT}}$:

$$€^{C}_{\rho}(\boldsymbol{p}) = €_\rho(\boldsymbol{p}) - €_{\text{CT}}(\boldsymbol{p},\boldsymbol{m}) \tag{#10}$$

Note that $€^{C}_{\rho}$ may be negative, in which case the user gets a refund. The user is also refunded when the number of *rejected* requests exceeds the maximum. For a rejected request $\boldsymbol{p} \in Q(\boldsymbol{u})^{i}_{N}$, with $\boldsymbol{u} = \rho_{\text{USER}}(\boldsymbol{p})$:

- Let $Q(\boldsymbol{p})_N$ = the sub-sequence of $Q(\boldsymbol{u})^{i}_{N}$ up to, but not including, $\boldsymbol{p}$ (where $\boldsymbol{i}$ is fixed by $\boldsymbol{p}$ & $\boldsymbol{N}$).
  - e.g. if $Q(\boldsymbol{u})^{i}_{N} = \{ \boldsymbol{p_1}, \boldsymbol{p_2}, \boldsymbol{p_3}, \boldsymbol{p_4}, \ldots \}$ then $Q(\boldsymbol{p_4})_N = \{ \boldsymbol{p_1}, \boldsymbol{p_2}, \boldsymbol{p_3} \}$.
- Then, the contribution, $€_{\text{FR}}(\boldsymbol{p})_N$, made by $\boldsymbol{p}$ to the failure-rate penalty $€_{\text{FR}}(\boldsymbol{u})^{i}_{N}$ is:

$$€_{\text{FR}}(\boldsymbol{p})_N = \begin{cases} \chi_{\text{FR}} \cdot \kappa_{\text{SL}}(\boldsymbol{u})^2 & \text{if } \boldsymbol{p} \text{ is rejected} \wedge |\perp(Q(\boldsymbol{p})_N)| \geq {}^{m}\lambda_{\text{FR}}(\boldsymbol{u}) \\ 0 & \text{otherwise} \end{cases} \tag{#11}$$

.. *where* $\boldsymbol{u} = \rho_{\text{USER}}(\boldsymbol{p})$

A.3. Utility (Profit)

This section presents formal definitions of the following utility-related functions:

- *Total Utility,* $\mathcal{U}_\omega$ (§A.3.1)
- *Utility of Individual Requests,* $\mathcal{U}_\rho$ (§A.3.2)
- *Expected Utility:* $\langle \mathcal{U}_\rho \rangle$ (§A.3.3)

A.3.1. Total Utility

The total *actual* profit/loss incurred over some period of time, ω, is calculated as follows:

- Let $\boldsymbol{P}_\omega$ = the set of all requests *received* by the RPM during ω;
- Let $\boldsymbol{M}_\omega$ = the set of all VMs available during ω;
- Then the profit/loss, $\mathcal{U}_\omega$, is defined as:

$$\mathcal{U}_\omega = \rho_{\text{COST}} - \rho_{\text{REFUNDS}} - \mu_{\text{COST}} \tag{#12}$$

.. *where*: ρ_{COST} = the total, $\sum_{\forall \boldsymbol{p} \in \boldsymbol{P}_\omega} €^{C}_{\rho}(\boldsymbol{p})$, net income (#10) from completed requests;
ρ_{REFUNDS} = the total, $\sum_{\forall \boldsymbol{p} \in \boldsymbol{P}_\omega} €_{\text{FR}}(\boldsymbol{p})_N$, of failure-rate penalties (#11) for *rejected* requests;
μ_{COST} = the total, $\sum_{\forall \boldsymbol{m} \in \boldsymbol{M}_\omega} €_\mu(\boldsymbol{m})$, cost (Table A.1.3-*e*) for VMs.

A.3.2. Utility of Individual Requests

The profit/loss, $\mathcal{U}_\rho$, derived from a *particular* request $\boldsymbol{p}$ is defined as follows:

$$\begin{array}{rl} \textit{for executing } \boldsymbol{p} \textit{ on machine } \boldsymbol{m}: & \mathcal{U}_\rho(\boldsymbol{p},\boldsymbol{m}) = \text{€}^C_\rho(\boldsymbol{p}) - {}^\rho\text{€}_\mu(\boldsymbol{p},\boldsymbol{m}) \\ \textit{for rejecting } \boldsymbol{p}: & \mathcal{U}_\rho(\boldsymbol{p}) = -\text{€}_{\text{FR}}(\boldsymbol{p})_N \end{array} \quad (\#13)$$

.. where:

- $\text{€}^C_\rho(\boldsymbol{p})$ and $\text{€}_{\text{FR}}(\boldsymbol{p})_N$ are as defined in #10 & #11 (*resp.*);
- ${}^\rho\text{€}_\mu(\boldsymbol{p},\boldsymbol{m})$ is the *processing cost* for executing $\boldsymbol{p}$ on $\boldsymbol{m}$, defined as follows (see Table A.1.3-*e*):

$${}^\rho\text{€}_\mu(\boldsymbol{p},\boldsymbol{m}) = {}^t\text{€}_\mu(\boldsymbol{m},\tau_{\text{END}}(\boldsymbol{p},\boldsymbol{m})) - \boldsymbol{x} \quad (\#14)$$

where:

$$\boldsymbol{x} = \begin{cases} {}^t\text{€}_\mu(\boldsymbol{m},\tau_{\text{START}}(\boldsymbol{p},\boldsymbol{m})) & \textit{if } \boldsymbol{p} \textit{ is } \underline{\textit{not}} \textit{ the } 1^{st} \textit{ request processed by } \boldsymbol{m} \\ 0 & \textit{otherwise} \end{cases}$$

If we assume (for the moment) that the processing cost ${}^\rho\text{€}_\mu(\boldsymbol{p},\boldsymbol{m})$ is some positive *constant* less than $\text{€}_\rho(\boldsymbol{p}) + \text{€}_{\text{FR}}(\boldsymbol{p})_N$ (explained later), then a plot of profit/loss, $\mathcal{U}_\rho(\boldsymbol{p},\boldsymbol{m})$, against completion-time, $\delta_{\text{CT}}(\boldsymbol{p},\boldsymbol{m})$, has the general characteristics shown in Figure A.3.2-*a* - namely:

- *For a completed* request: $\mathcal{U}_\rho(\boldsymbol{p},\boldsymbol{m})$ stays constant at $\text{€}_\rho(\boldsymbol{p}) - {}^\rho\text{€}_\mu(\boldsymbol{p},\boldsymbol{m})$ until the maximum completion-time, ${}^m\delta_{\text{CT}}(\boldsymbol{p})$, after which it declines steadily as the completion-time penalty, €_{CT}, accumulates;
- *For a rejected* request: $\mathcal{U}_\rho(\boldsymbol{p})$ remains constant at $-\text{€}_{\text{FR}}(\boldsymbol{p})_N$.

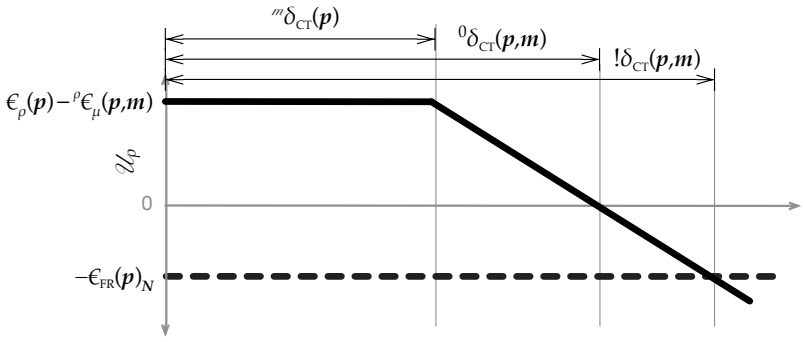

Figure A.3.2-a: Plot of Profit vs Completion-Time
solid line = $\mathcal{U}_\rho$ for a completed request; dotted line = $\mathcal{U}_\rho$ for a rejected request

By inspection of Figure A.3.2-*a*, we can distinguish the following 3 key values for $\delta_{\text{CT}}(\boldsymbol{p},\boldsymbol{m})$ (giving rise to the intervals listed in Table A.3.2-*a*):

- ${}^m\delta_{\text{CT}}(\boldsymbol{p})$, the maximum completion-time (up to which $\mathcal{U}_\rho(\boldsymbol{p},\boldsymbol{m})$ is maximal);
- ${}^0\delta_{\text{CT}}(\boldsymbol{p},\boldsymbol{m})$, the 'zero' point at which $\mathcal{U}_\rho(\boldsymbol{p},\boldsymbol{m}) = 0$;
- $!\delta_{\text{CT}}(\boldsymbol{p},\boldsymbol{m})$, the 'cut-off' point at which $\mathcal{U}_\rho(\boldsymbol{p},\boldsymbol{m}) = \mathcal{U}_\rho(\boldsymbol{p})$.

Table A.3.2-a: Significant Completion-Time Intervals

δ_{CT}	*Description*
$[0, {}^m\delta_{\text{CT}}(\boldsymbol{p})]$	maximum possible profit
$({}^m\delta_{\text{CT}}(\boldsymbol{p}), {}^0\delta_{\text{CT}}(\boldsymbol{p},\boldsymbol{m}))$	positive but less than maximum profit.
$[{}^0\delta_{\text{CT}}(\boldsymbol{p},\boldsymbol{m}), !\delta_{\text{CT}}(\boldsymbol{p},\boldsymbol{m}))$	Zero profit (breaking even) or loss, but not as bad as rejecting the request.
$[!\delta_{\text{CT}}(\boldsymbol{p},\boldsymbol{m}), \infty]$	Completing the request results in greater loss than rejecting it: - for $\delta_{\text{CT}} = !\delta_{\text{CT}}$ we assume rejecting the request is best, since completing it merely consumes VM processing time to no financial benefit.

The *cut-off* point, $!\delta_{CT}(\boldsymbol{p},\boldsymbol{m})$, is given by the following formula:

$$!\delta_{CT}(\boldsymbol{p},\boldsymbol{m}) = {}^{m}\delta_{CT}(\boldsymbol{p}) + [\ \epsilon_{\rho}(\boldsymbol{p}) - {}^{\rho}\epsilon_{\mu}(\boldsymbol{p},\boldsymbol{m}) + \epsilon_{FR}(\boldsymbol{p})_N\ ] / \Phi(\boldsymbol{p}) \cdot \chi_{CT} \qquad (\#15)$$

Derivation: by definition $\delta_{CT} = !\delta_{CT}$ is the point at which $\mathcal{U}_{\rho}(\boldsymbol{p}) = \mathcal{U}_{\rho}(\boldsymbol{p},\boldsymbol{m})$, so from #13:

$$-\epsilon_{FR}(\boldsymbol{p})_N = \epsilon^{C}_{\rho}(\boldsymbol{p}) - {}^{\rho}\epsilon_{\mu}(\boldsymbol{p},\boldsymbol{m})$$

substitute ϵ^{C}_{ρ}
$$= \epsilon_{\rho}(\boldsymbol{p}) - \epsilon_{CT}(\boldsymbol{p},\boldsymbol{m}) - {}^{\rho}\epsilon_{\mu}(\boldsymbol{p},\boldsymbol{m}) \qquad \textit{see \#10}$$

re-arrange
$$\epsilon_{CT}(\boldsymbol{p},\boldsymbol{m}) = \epsilon_{\rho}(\boldsymbol{p}) - {}^{\rho}\epsilon_{\mu}(\boldsymbol{p},\boldsymbol{m}) + \epsilon_{FR}(\boldsymbol{p})_N$$

substitute ϵ_{CT}
$$\chi_{CT} \cdot (!\delta_{CT}(\boldsymbol{p},\boldsymbol{m}) - {}^{m}\delta_{CT}(\boldsymbol{p})) \cdot \Phi(\boldsymbol{p}) = \qquad \textit{see Table A.2.3-b}$$

divide by $\chi_{CT} \cdot \Phi(\boldsymbol{p})$
$$!\delta_{CT}(\boldsymbol{p},\boldsymbol{m}) - {}^{m}\delta_{CT}(\boldsymbol{p}) = [\ \epsilon_{\rho}(\boldsymbol{p}) - {}^{\rho}\epsilon_{\mu}(\boldsymbol{p},\boldsymbol{m}) + \epsilon_{FR}(\boldsymbol{p})_N\ ] / \Phi(\boldsymbol{p}) \cdot \chi_{CT}$$

The 'zero' point, ${}^{0}\delta_{CT}(\boldsymbol{p},\boldsymbol{m})$, is then found by substituting $\epsilon_{FR}(\boldsymbol{p})_N = 0$ into #15:

$${}^{0}\delta_{CT}(\boldsymbol{p},\boldsymbol{m}) = {}^{m}\delta_{CT}(\boldsymbol{p}) + [\ \epsilon_{\rho}(\boldsymbol{p}) - {}^{\rho}\epsilon_{\mu}(\boldsymbol{p},\boldsymbol{m})\ ]\ /\ \Phi(\boldsymbol{p}) \cdot \chi_{CT} \qquad (\#16)$$

Finally, recall that the graph in Figure A.3.2-*a* assumed *i*) a constant ${}^{\rho}\epsilon_{\mu}(\boldsymbol{p},\boldsymbol{m})$ with *ii*) a positive value less than or equal to the maximum profit $\epsilon_{\rho}(\boldsymbol{p}) + \epsilon_{FR}(\boldsymbol{p})_N$. Relaxing these assumptions:

- By definition, ${}^{\rho}\epsilon_{\mu}(\boldsymbol{p},\boldsymbol{m})$ is not constant, but instead increases step-wise with $\delta_{CT}(\boldsymbol{p},\boldsymbol{m})$ (see #14 & ${}^{t}\epsilon_{\mu}$ in Table A.1.3-*e*), with corresponding discontinuities in $\mathcal{U}_{\rho}(\boldsymbol{p},\boldsymbol{m})$ (as indicated in Figure A.3.2-*b*), and thus a decrease in the durations ${}^{0}\delta_{CT}$ and $!\delta_{CT}$. Equations #15 and #16, however, remain valid.

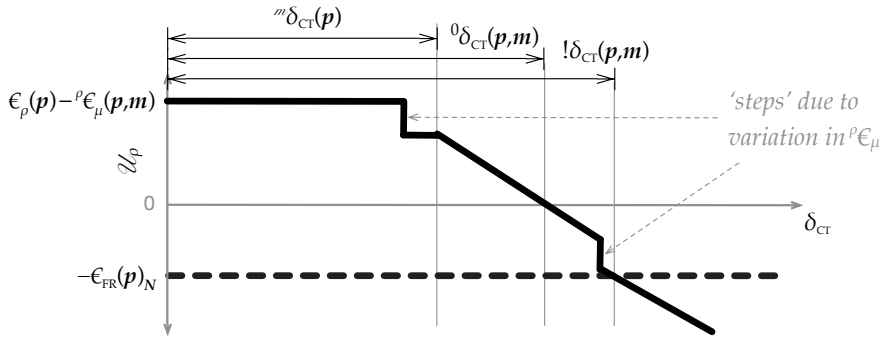

Figure A.3.2-b: Plot of Profit vs Completion-Time
solid line = $\mathcal{U}_{\rho}$ for a completed request; dotted line = $\mathcal{U}_{\rho}$ for a rejected request

- The second assumption, ${}^{\rho}\epsilon_{\mu}(\boldsymbol{p},\boldsymbol{m}) \le \epsilon_{\rho}(\boldsymbol{p}) + \epsilon_{FR}(\boldsymbol{p})_N$, is only broken when ${}^{\rho}\epsilon_{\mu}(\boldsymbol{p},\boldsymbol{m}) > \epsilon_{\rho}(\boldsymbol{p}) + \epsilon_{FR}(\boldsymbol{p})_N$, or by rearrangement, $\epsilon_{\rho}(\boldsymbol{p}) - {}^{\rho}\epsilon_{\mu}(\boldsymbol{p},\boldsymbol{m}) < -\epsilon_{FR}(\boldsymbol{p})_N$, which means that $\mathcal{U}_{\rho}(\boldsymbol{p},\boldsymbol{m}) < \mathcal{U}_{\rho}(\boldsymbol{p})$ for all values of $\delta_{CT}(\boldsymbol{p},\boldsymbol{m})$ - in which case it is always best to reject the request.

A.3.3. Expected Utility

The definition of profit/loss given above, $\mathcal{U}_{\rho}$ (#13), is an *after-the-fact* the definition, in that it can only be calculated *after* a request has been either rejected or fully processed. The RPM, however, has to decide what to do with requests *in advance* (as they arrive), and must therefore base its decision on the *expected* rather than *actual* profit. The *expected* profit, $\langle \mathcal{U}_{\rho} \rangle$, has essentially the same definition as actual profit, $\mathcal{U}_{\rho}$, but with all *actual* valued terms, $\boldsymbol{t}$, replaced by corresponding *expectation* terms, $\langle \boldsymbol{t} \rangle$. It turns out that the only effective difference is the *execution-time*, δ_{XT} (#8), for whose expectation value we shall assume the worse case (with $\varepsilon_{XT} = 0.1 \cdot \rho_{SIZE}(\boldsymbol{p})$), hence:

$$\langle \delta_{XT}(\boldsymbol{p},\boldsymbol{m}) \rangle = 1.1 \cdot \rho_{SIZE}(\boldsymbol{p}) / \mu_{POW}(\boldsymbol{m}) \qquad (\#17)$$

Note that the *time-out* delay, δ_{TO} (see §A.2.1), is *per force* an 'expected' value. We define δ_{TO} such that the request completion-time, δ_{CT} (#2), should never exceed the *cut-off*, $!\delta_{CT}$ (#15), as follows:

$$\delta_{TO}(\boldsymbol{p},\boldsymbol{m}) = \langle !\delta_{CT}(\boldsymbol{p},\boldsymbol{m}) \rangle - \langle \delta_{AT}(\boldsymbol{p}) \rangle - 1.5 \cdot \langle \delta_{XT}(\boldsymbol{p},\boldsymbol{m}) \rangle \qquad (\#18)$$

- In other words: before the time-out it is more profitable to complete the request, while after the time-out it is more profitable to reject the request. Timed-out request should therefore be rejected.

A.4. The Assignment Task

The basic task of the RPM is to decide, on the basis of expected utility, $\langle \mathcal{U}_\rho \rangle$ (§A.3.3), which sequences of requests to assign to which VMs. This section formally defines this *assignment task*. To begin:

- Let $\boldsymbol{P}$ = a set of requests;
- Let $\boldsymbol{M}$ = a set of VMs;

Next, recall from #7 that requests can onlybe assigned to VMs with sufficient disk capacity. We define the predicate, $C(\boldsymbol{p},\boldsymbol{m})$, to be true *iff* request $\boldsymbol{p}$ can be assigned to machine $\boldsymbol{m}$:

$$C(\boldsymbol{p},\boldsymbol{m}) \Leftrightarrow \ \mu_{\text{DISK}}(\boldsymbol{m}) \geq \rho_{\text{SIZE}}(\boldsymbol{p}) \tag{\#19}$$

Then, for a given request, $\boldsymbol{p}$:

$$\begin{gathered}\text{Let } {}^{\text{C}}\mu(\boldsymbol{p}) = \{\, \boldsymbol{m} : C(\boldsymbol{p},\boldsymbol{m}),\ \forall\, \boldsymbol{m}\in\boldsymbol{M} \,\} \\ = \textit{the set of VMs to which request } \boldsymbol{p} \textit{ can be assigned} \\ \text{If } {}^{\text{C}}\mu(\boldsymbol{p}) \neq \varnothing \text{ then } \boldsymbol{p} \text{ is ``}\textit{assignable}\text{''}, \text{ otherwise } \boldsymbol{p} \text{ must be rejected.}\end{gathered} \tag{\#20}$$

And:

- Let $^{\circ}2^{P}$ = the *largest* set of ordered sub-sets of $\boldsymbol{P}$ such that: $\forall\ \boldsymbol{S}\in {}^{\circ}2^{P} : \forall\ \boldsymbol{p},\boldsymbol{q}\in\boldsymbol{S} : {}^{\text{C}}\mu(\boldsymbol{p}) \cap {}^{\text{C}}\mu(\boldsymbol{q}) \neq \varnothing$, i.e. the set of *all possible* sequences of requests, $\langle \boldsymbol{p}_1, \boldsymbol{p}_2, \boldsymbol{p}_3, \ldots \rangle$, such that for every sequence there is at least one machine to which all its elements can be assigned.

We now define an 'assignment' as a mapping from *machines* to *sequences of requests*:

$$\begin{gathered}\text{Let } \alpha{:}\boldsymbol{M}\rightarrow {}^{\circ}2^{P} = \text{an ``assignment''} \\ = \textit{a total function sending a VM to a sequence of requests such that:}\end{gathered} \tag{\#21}$$

$$\begin{array}{ll} \forall\, \boldsymbol{m},\boldsymbol{n}\in\boldsymbol{M} : (\boldsymbol{m}\neq\boldsymbol{n} \Rightarrow \alpha(\boldsymbol{m})\cap\alpha(\boldsymbol{n}) = \varnothing) & \textit{No two sequences contain the same request, \&} \\ \qquad\qquad \wedge & \textit{Every request in a sequence can be assigned to} \\ \forall\, \boldsymbol{m}\in\boldsymbol{M} : (\forall\, \boldsymbol{p}\in\alpha(\boldsymbol{m}) : \boldsymbol{m}\in {}^{\text{C}}\mu(\boldsymbol{p})) & \textit{the machine mapped to that sequence.} \end{array}$$

Now from #13 (§A.3.2):

- $\mathcal{U}_\rho(\boldsymbol{p},\boldsymbol{m})$ is the profit/loss for executing request $\boldsymbol{p}$ on machine $\boldsymbol{m}$, hence:
  - $\sum_{\forall \boldsymbol{p}\in\alpha(\boldsymbol{m})} \mathcal{U}_\rho(\boldsymbol{p},\boldsymbol{m})$ is the profit/loss for executing a *sequence* of requests, $\alpha(\boldsymbol{m}) = \langle \boldsymbol{p}_1, \boldsymbol{p}_2, \boldsymbol{p}_3, \ldots \rangle$, on a given machine $\boldsymbol{m}$[23]; and
  - $\sum_{\forall \boldsymbol{m}\in\boldsymbol{M}} (\sum_{\forall \boldsymbol{p}\in\alpha(\boldsymbol{m})} \mathcal{U}_\rho(\boldsymbol{p},\boldsymbol{m}))$ is the profit/loss for executing all the sequences defined by α.
- $\mathcal{U}_\rho(\boldsymbol{p})$ is the loss from rejecting request $\boldsymbol{p}$:
  - if $\boldsymbol{F} = \cup_{\forall \boldsymbol{m}\in\boldsymbol{M}} \alpha(\boldsymbol{m})$ is the set of all requests assigned by α, then the difference $\boldsymbol{P}\setminus\boldsymbol{F}$ is the set of requests that are rejected by α, and $\sum_{\forall \boldsymbol{p}\in\boldsymbol{P}\setminus\boldsymbol{S}} \mathcal{U}_\rho(\boldsymbol{p})$ is the total loss[24] due to rejected requests.

The total profit/loss, $\mathcal{U}(\alpha)$, for a given assignment α is then defined as follows:

$$\begin{gathered}\text{Let } \mathcal{U}(\alpha) = \textstyle\sum_{\forall \boldsymbol{m}\in\boldsymbol{M}} (\sum_{\forall \boldsymbol{p}\in\alpha(\boldsymbol{m})} \mathcal{U}_\rho(\boldsymbol{p},\boldsymbol{m})) + \sum_{\forall \boldsymbol{p}\in\boldsymbol{P}\setminus\boldsymbol{F}} \mathcal{U}_\rho(\boldsymbol{p}) \\ = \textit{the expected profit for an assignment } \alpha, \textit{ where: } \boldsymbol{F} = \cup_{\forall \boldsymbol{m}\in\boldsymbol{M}} \alpha(\boldsymbol{m})\end{gathered} \tag{\#22}$$

Finally, if $\boldsymbol{A}$ is the set of all possible assignments for a given $\boldsymbol{P}$ & $\boldsymbol{M}$, then the assignment task can be formally defined as determining the assignment, $\alpha\in\boldsymbol{A}$, with maximal $\mathcal{U}(\alpha)$.

The problem is compounded, however, by the fact that:

- $\boldsymbol{P}$ changes as new requests are received from users, and as existing requests are assigned to VMs;
- $\boldsymbol{M}$ changes as the RPM adds/removes VMs to/from the Resource Pool.

To cope with this changeability, we adopt the strategy outlined in §4.2 ('RPM') of the main document: solving the assignment problem for small fixed sets of request and VMs at regular intervals ~ whose period then determines the action-time, δ_{AT}.

---

[23] Note that each request in $\alpha(\boldsymbol{m})$ is part of the *prior* queue, $Q(\boldsymbol{p})$ (#4), for the subsequent request, i.e. $Q(\boldsymbol{p}_{i+1}) = Q(\boldsymbol{p}_i) \cup \{\, \boldsymbol{p}_i \,\}$.

[24] Note that, $\mathcal{U}_\rho(\boldsymbol{p})$, and hence also any summation, $\sum$, over $\mathcal{U}_\rho(\boldsymbol{p})$ values, is by definition a negative value (≤ 0).

A.5. The Default VM for a Request

As described in §2 of the main document, it is possible, given a request, to determine an optimal VM configuration for processing that request. This section details how we arrive at this optimal VM.

Consider the case that a request, $\boldsymbol{p}$, is received for which no suitable VM currently exists (i.e. ${}^{\mathrm{C}}\mu(\boldsymbol{p}) = \varnothing$; see §A.4), such that a new VM, $\boldsymbol{m}$, must be launched in order to process $\boldsymbol{p}$. We can maximise the profit $\langle \mathscr{U}_p(\boldsymbol{p},\boldsymbol{m}) \rangle$ for satisfying $\boldsymbol{p}$ by minimising its completion-time, $\delta_{\mathrm{CT}}(\boldsymbol{p},\boldsymbol{m})$, where:

$$\langle \delta_{\mathrm{CT}}(\boldsymbol{p},\boldsymbol{m}) \rangle = \langle \delta_{\mathrm{AT}}(\boldsymbol{p}) \rangle + \langle \delta_{\mathrm{QT}}(\boldsymbol{p},\boldsymbol{m}) \rangle + 1.5 \cdot \langle \delta_{\mathrm{XT}}(\boldsymbol{p},\boldsymbol{m}) \rangle$$

Since $\boldsymbol{m}$ does not yet exist, the queuing-time, $\langle \delta_{\mathrm{QT}}(\boldsymbol{p},\boldsymbol{m}) \rangle$, is just the deploy-time, $\langle \delta_{\mathrm{DT}}(\boldsymbol{m}) \rangle$ (#5), which from Table A.1.3-*a* is equal to $0.2 \cdot \mu_{\mathrm{SPEC}}(\boldsymbol{m})$. We will also assume a worse case execution-time, $\langle \delta_{\mathrm{XT}}(\boldsymbol{p},\boldsymbol{m}) \rangle = 1.1 \cdot \rho_{\mathrm{SIZE}}(\boldsymbol{p}) / \mu_{\mathrm{POW}}(\boldsymbol{m})$ (from #17), hence:

$$\begin{aligned}
\langle \delta_{\mathrm{CT}}(\boldsymbol{p},\boldsymbol{m}) \rangle &= \langle \delta_{\mathrm{AT}}(\boldsymbol{p}) \rangle + 0.2 \cdot \mu_{\mathrm{SPEC}}(\boldsymbol{m}) + 1.65 \cdot \rho_{\mathrm{SIZE}}(\boldsymbol{p}) / \mu_{\mathrm{POW}}(\boldsymbol{m}) \\
&= \langle \delta_{\mathrm{AT}}(\boldsymbol{p}) \rangle + 0.2 \cdot (\mu_{\mathrm{POW}}(\boldsymbol{m}) + \mu_{\mathrm{DISK}}(\boldsymbol{m})) + 1.65 \cdot \rho_{\mathrm{SIZE}}(\boldsymbol{p}) / \mu_{\mathrm{POW}}(\boldsymbol{m}) \text{ (Table A.1.2-}a\text{)} \\
&= 0.2 \cdot \mu_{\mathrm{POW}}(\boldsymbol{m}) + 1.65 \cdot \rho_{\mathrm{SIZE}}(\boldsymbol{p}) / \mu_{\mathrm{POW}}(\boldsymbol{m}) + \langle \delta_{\mathrm{AT}}(\boldsymbol{p}) \rangle + 0.2 \cdot \mu_{\mathrm{DISK}}(\boldsymbol{m})
\end{aligned}$$

Now $\rho_{\mathrm{SIZE}}(\boldsymbol{p})$ is given, and $\mu_{\mathrm{DISK}}(\boldsymbol{m}) \geq \rho_{\mathrm{SIZE}}(\boldsymbol{p})$ (#7). Since we are looking to maximise profit, we also want to ensure that $\mu_{\mathrm{DISK}}(\boldsymbol{m}) - \rho_{\mathrm{SIZE}}(\boldsymbol{p})$ is minimised[25], the upshot of which is that $\mu_{\mathrm{DISK}}(\boldsymbol{m})$ is determined just by $\rho_{\mathrm{SIZE}}(\boldsymbol{p})$ ~ specifically, given the permitted values of $\rho_{\mathrm{SIZE}}(\boldsymbol{p})$ & $\mu_{\mathrm{DISK}}(\boldsymbol{m})$ (Tables A.2.2-*a* & A.1.2-*a resp.*):

$$\mu_{\mathrm{DISK}}(\boldsymbol{m}) = 2^{\lceil \log_2(\rho_{\mathrm{SIZE}}(\boldsymbol{p})) \rceil} \qquad (\#22)$$

We will also assume that the action time, $\langle \delta_{\mathrm{AT}}(\boldsymbol{p}) \rangle$, is either a constant or a function over $\rho_{\mathrm{SIZE}}(\boldsymbol{p})$, which means that we can only attempt to minimise $\langle \delta_{\mathrm{CT}}(\boldsymbol{p},\boldsymbol{m}) \rangle$ by changing $\mu_{\mathrm{POW}}(\boldsymbol{m})$. Taking the differential of $\langle \delta_{\mathrm{CT}}(\boldsymbol{p},\boldsymbol{m}) \rangle$ with respect to $\mu_{\mathrm{POW}}(\boldsymbol{m})$ ..

$$d\langle \delta_{\mathrm{CT}}(\boldsymbol{p},\boldsymbol{m}) \rangle / d\mu_{\mathrm{POW}}(\boldsymbol{m}) = 0.2 - 1.65 \cdot \rho_{\mathrm{SIZE}}(\boldsymbol{p}) / \mu_{\mathrm{POW}}(\boldsymbol{m})^2$$

.. the minima occur when:

$$\begin{aligned}
0.2 - 1.65 \cdot \rho_{\mathrm{SIZE}}(\boldsymbol{p}) / \mu_{\mathrm{POW}}(\boldsymbol{m})^2 &= 0 \\
\textit{rearranging:} \qquad \mu_{\mathrm{POW}}(\boldsymbol{m})^2 &= 1.65 \cdot \rho_{\mathrm{SIZE}}(\boldsymbol{p}) / 0.2 \\
\mu_{\mathrm{POW}}(\boldsymbol{m}) &= \sqrt{(8.25 \cdot \rho_{\mathrm{SIZE}}(\boldsymbol{p}))} \qquad (\#23)
\end{aligned}$$

Equation #23 thus gives *the unique*[26] value of $\mu_{\mathrm{POW}}(\boldsymbol{m})$, call it ${}^{\circ}\mu_{\mathrm{POW}}(\boldsymbol{m})$, for which $\langle \delta_{\mathrm{CT}}(\boldsymbol{p},\boldsymbol{m}) \rangle$ is minimal. From Table A.1.2-*a*, however, $\mu_{\mathrm{POW}}(\boldsymbol{m})$ is constrained to take only certain values[27], so:

- Let ${}^{-}\mu_{\mathrm{POW}}(\boldsymbol{m})$ = the largest permitted value of $\mu_{\mathrm{POW}}(\boldsymbol{m}) \leq {}^{\circ}\mu_{\mathrm{POW}}(\boldsymbol{m})$, if any, else ${}^{-}\mu_{\mathrm{POW}}(\boldsymbol{m}) = 1.6$;
- Let ${}^{+}\mu_{\mathrm{POW}}(\boldsymbol{m})$ = the smallest permitted value of $\mu_{\mathrm{POW}}(\boldsymbol{m}) > {}^{\circ}\mu_{\mathrm{POW}}(\boldsymbol{m})$, if any, else ${}^{+}\mu_{\mathrm{POW}}(\boldsymbol{m}) = 16.8$;
- We then choose either ${}^{-}\mu_{\mathrm{POW}}(\boldsymbol{m})$ or ${}^{+}\mu_{\mathrm{POW}}(\boldsymbol{m})$ based on which gives the greatest utility according to the following formula (derived in Box A.5-*a* next page):

$$\langle \mathscr{U}_p(\boldsymbol{p},\boldsymbol{m}) \rangle = \Phi(\boldsymbol{p}) \cdot (\chi_\rho \cdot \rho_{\mathrm{SIZE}}(\boldsymbol{p}) - \chi_{\mathrm{CT}} \cdot \langle \Delta_{\mathrm{CT}}(\boldsymbol{p},\boldsymbol{m}) \rangle) - \chi_\mu(\boldsymbol{m}) \cdot \lceil \boldsymbol{\alpha} / \delta_\mu \rceil \qquad (\#24)$$

$$\textit{where: } \boldsymbol{\alpha} = 0.2 \cdot (\mu_{\mathrm{POW}}(\boldsymbol{m}) + \mu_{\mathrm{DISK}}(\boldsymbol{m})) + 2.2 \cdot \rho_{\mathrm{SIZE}}(\boldsymbol{p}) / \mu_{\mathrm{POW}}(\boldsymbol{m})$$

In sum, for a request, $\boldsymbol{p}$, for which no suitable machines currently exist (${}^{\mathrm{C}}\mu(\boldsymbol{p}) = \varnothing$), equations #22 and #23 give a unique VM specification, $\mu_{\mathrm{SPEC}}(\boldsymbol{m}) = \mu_{\mathrm{POW}}(\boldsymbol{m}) + \mu_{\mathrm{DISK}}(\boldsymbol{m})$, for which the expected completion-time, $\langle \delta_{\mathrm{CT}}(\boldsymbol{p},\boldsymbol{m}) \rangle$ is minimised. We refer to this as the 'default' VM for a request. Generalising, even when suitable machines *are* available to process a request, there is no guarantee that any of them will give higher utility than the default. Hence, in all cases it is worthwhile to assess the utility of assigning the request to its default VM.

---

[25] Since $\rho_{\mathrm{SIZE}}(\boldsymbol{p})$ is fixed, increasing $\mu_{\mathrm{DISK}}(\boldsymbol{m}) - \rho_{\mathrm{SIZE}}(\boldsymbol{p})$ just means increasing $\mu_{\mathrm{DISK}}(\boldsymbol{m})$, hence also $\mu_{\mathrm{SPEC}}(\boldsymbol{m})$ and so *i*) the cost ${}^{\rho}\text{€}_\mu(\boldsymbol{p},\boldsymbol{m})$, of using $\boldsymbol{m}$ to process $\boldsymbol{p}$, and *ii*) the deploy-time $\delta_{\mathrm{DT}}(\boldsymbol{m})$, and hence also the completion-time, $\delta_{\mathrm{CT}}(\boldsymbol{p},\boldsymbol{m})$.

[26] We can ignore $-ve$ roots, since by definition $\mu_{\mathrm{POW}}(\mathbf{m}) > 0$.

[27] More precisely: $\mu_{\mathrm{POW}}(\boldsymbol{m}) = \mu_{\mathrm{CLK}}(\boldsymbol{m}) \cdot \mu_{\mathrm{CORES}}(\boldsymbol{m}) + 0.6 \cdot \mu_{\mathrm{MEM}}(\boldsymbol{m})$ where $\mu_{\mathrm{CLK}}(\boldsymbol{m})$, $\mu_{\mathrm{CORES}}(\boldsymbol{m})$ and $\mu_{\mathrm{MEM}}(\boldsymbol{m})$ have constrained values.

Box A.5-a: Expected Profit w.r.t. the Default VM

From #13, the expected profit is: $\langle \mathcal{U}_\rho(\boldsymbol{p},\boldsymbol{m})\rangle = \mathbb{E}^{\mathrm{C}}_\rho(\boldsymbol{p}) - \langle {}^{\rho}\mathbb{E}_\mu(\boldsymbol{p},\boldsymbol{m})\rangle$:

From #10 and Table A.2.3-*b*: $\mathbb{E}^{\mathrm{C}}_\rho(\boldsymbol{p}) = \mathbb{E}_\rho(\boldsymbol{p}) - \langle \mathbb{E}_{\mathrm{CT}}(\boldsymbol{p},\boldsymbol{m})\rangle$

$$= \chi_\rho \cdot \rho_{\mathrm{SIZE}}(\boldsymbol{p}) \cdot \Phi(\boldsymbol{p}) - \chi_{\mathrm{CT}} \cdot \langle \Delta_{\mathrm{CT}}(\boldsymbol{p},\boldsymbol{m})\rangle \cdot \Phi(\boldsymbol{p})$$

$$= \Phi(\boldsymbol{p}) \cdot (\chi_\rho \cdot \rho_{\mathrm{SIZE}}(\boldsymbol{p}) - \chi_{\mathrm{CT}} \cdot \langle \Delta_{\mathrm{CT}}(\boldsymbol{p},\boldsymbol{m})\rangle)$$

where: $\langle \Delta_{\mathrm{CT}}(\boldsymbol{p},\boldsymbol{m})\rangle = \langle \delta_{\mathrm{CT}}(\boldsymbol{p},\boldsymbol{m})\rangle - {}^{m}\delta_{\mathrm{CT}}(\boldsymbol{p})$, if $\langle \delta_{\mathrm{CT}}(\boldsymbol{p},\boldsymbol{m})\rangle > {}^{m}\delta_{\mathrm{CT}}(\boldsymbol{p})$, else 0

and: ${}^{m}\delta_{\mathrm{CT}}(\boldsymbol{p}) = \kappa_{\mathrm{CT}} \cdot \rho_{\mathrm{SIZE}}(\boldsymbol{p}) / \Phi(\boldsymbol{p})$

From #14: $\langle {}^{\rho}\mathbb{E}_\mu(\boldsymbol{p},\boldsymbol{m})\rangle = {}^{t}\mathbb{E}_\mu(\boldsymbol{m},\langle \tau_{\mathrm{END}}(\boldsymbol{p},\boldsymbol{m})\rangle)$

$$= \chi_\mu(\boldsymbol{m}) \cdot \lceil \boldsymbol{\alpha} / \delta_\mu \rceil, \textit{ where}: \boldsymbol{\alpha} = \langle \tau_{\mathrm{END}}(\boldsymbol{p},\boldsymbol{m})\rangle - \langle \tau_{\mathrm{LAUNCH}}(\boldsymbol{m})\rangle$$

$$= \delta_{\mathrm{DT}}(\boldsymbol{m}) + \langle \delta_{\mathrm{PT}}(\boldsymbol{p},\boldsymbol{m})\rangle$$

$$= 0.2 \cdot \mu_{\mathrm{SPEC}}(\boldsymbol{m}) + 2 \cdot \langle \delta_{\mathrm{XT}}(\boldsymbol{p},\boldsymbol{m})\rangle$$

(*taking the worse case* δ_{XT}) $= 0.2 \cdot \mu_{\mathrm{SPEC}}(\boldsymbol{m}) + 2.2 \cdot \rho_{\mathrm{SIZE}}(\boldsymbol{p}) / \mu_{\mathrm{POW}}(\boldsymbol{m})$

$$= 0.2 \cdot (\mu_{\mathrm{POW}}(\boldsymbol{m}) + \mu_{\mathrm{DISK}}(\boldsymbol{m})) + 2.2 \cdot \rho_{\mathrm{SIZE}}(\boldsymbol{p}) / \mu_{\mathrm{POW}}(\boldsymbol{m})$$

Hence, the expected profit:

$$\langle \mathcal{U}_\rho(\boldsymbol{p},\boldsymbol{m})\rangle = \Phi(\boldsymbol{p}) \cdot (\chi_\rho \cdot \rho_{\mathrm{SIZE}}(\boldsymbol{p}) - \chi_{\mathrm{CT}} \cdot \langle \Delta_{\mathrm{CT}}(\boldsymbol{p},\boldsymbol{m})\rangle) - \chi_\mu(\boldsymbol{m}) \cdot \lceil \alpha / \delta_\mu \rceil$$

B. Key Differences *w.r.t.* D4.4.1

This appendix briefly outlines the key differences between the model presented in this deliverable and the original version presented in D4.4.1. In terms of code structure, the changes are considerable. The ABS implementation described in D4.4.1 (§2.4), for example, includes 24 distinct interfaces/classes (not counting interface implementations), while the present model comprises just 6. The underlying reasons for these structural changes are as follows:

- *Simplification of the architecture*:
  - The architecture of the ETICS service as originally presented consisted of 3 main components: the *RPM*, the *Resource Pool* (the set of deployed VMs), and an intermediary *Execution Engine*, whose task was to distribute requests (received from the RPM), one at a time, to their assigned VMs (as those VMs became available). With the introduction of the distributed GA (see §3), however, and in particular the decision to execute GA processes on pooled VMs, we decided that the Execution Engine *per se* was an unnecessary complication[28].
  - The original model also included a *ResourceFactory* interface, which has since been superseded by the built-in ABS *CloudProvider* & *DeploymentComponent* APIs.
- *Move towards functional (as opposed to OO) modelling*:
  - As noted in §4, for the present deliverable we have tried to restrict the use of OO constructs for modelling only the active components of the system, with the information exchanged between them captured, instead, by ABS datatypes. Accordingly, many of the interfaces/classes defined in D4.4.1 have been replaced by functional counterparts. In particular:
    - *ResourceInfo*/*ResourceConfig* interfaces have been replaced by *VMData*/*VMInfo* datatypes[29];
    - The *Request* & *SLA* interfaces have been replaced by corresponding *Request* & *SLA* datatypes;
- *Simplified User Model*:
  - The original ABS model incorporated a more complex model of end users. Specifically, each user belonged to a 'team', operating within a particular international time-zone, with request frequencies (probabilistically) determined by typical 'working hours' (e.g. 9-5, Mon-Fri). In the present deliverable we decided that this level of detail was an unnecessary complication, and opted instead for a minimal stochastic model (generating requests with random parameters at random times). Arbitrarily more complex/realistic scenarios can be added later if needed.

With fewer building blocks, we can also simplify the overall interface/class hierarchy (in particular by collapsing the original high-level *Mutable*, *Monitored* & *Queriable* Resource Pool distinctions), and eliminate some of the observer interactions (i.e. *RequestListener* & *ResourceListener* interfaces). Progress and state monitoring in the original model was distributed and piecemeal, but is now (as noted in §4) centralised in a singleton *Tally* object.

Finally, in D4.4.1 (§3.5) we noted an intention to investigate diverse RPM implementations (namely: 'Baseline', 'Drive-Based' & 'Envisage' approaches), collectively packaged under a configurable feature model. Following feedback from other partners, we opted instead just for the distributed GA approach described in this deliverable.

[28] But note that the original model remains the more generic, since it makes no assumptions about how the RPM's distributed decision mechanism is implemented.

[29] The name change from 'Resource' to 'VM' is due to the adoption in ABS of the term 'resource' to signify CPU-speed, memory, disk space, band-width, and the like. We decided, however, that it was best to stick with the terms 'Resource Pool' and 'Resource Pool Manager' (RPM) for the ETICS components.