Economic Aspects of a Utility Computing Service

Jean Paul Degabriele, David Pym
Trusted Systems Laboratory
HP Laboratories Bristol
HPL-2007-101
July 3, 2007*

Utility Computing, modeling, DEMOS 2000, pricing, security, flexing

This paper presents a case study of business and systems modelling for a Utility Computing service. Our analysis is focused mainly on service pricing, resource flexing, and costs related to preventive security measures. We further present a discrete event model of a Utility Computing service, and show how the information obtained from such a model can aid business and design decisions.

Economic Aspects of a Utility Computing Service

Jean Paul Degabriele David Pym

Trusted Systems Laboratory, Hewlett-Packard Laboratories,
Filton Road, Stoke Gifford, Bristol BS34 8QZ, UK

{jeanpaul.degabriele, david.pym}@hp.com

Abstract

This paper presents a case study of business and systems modelling for a Utility Computing service. Our analysis is focused mainly on service pricing, resource flexing, and costs related to preventive security measures. We further present a discrete event model of a Utility Computing service, and show how the information obtained from such a model can aid business and design decisions.

Keywords: Utility Computing, Modelling, DEMOS 2000, Pricing, Security, Flexing.

1. Introduction

Utility Computing is expected by many technologists to be one of the next major sources of income in the IT services market. While a considerable number of providers already hit the market, others are still studying the full potential of utility computing services. Most of the research is focused on the possible business models and their pricing, architecture of the infrastructure, and ways of securing utility computing services. Because of the many possible alternatives in designing and offering a utility computing service, we propose a modelling methodology to explore these possibilities and their interaction. We make use of Demos 2000 [2] as our modelling and simulation platform. Our analysis is mainly focused on system scrubbing, flexing, and utility computing service pricing.

2. Utility Computing

Utility Computing is more of a different approach to computing resource than a new computing technology. The main idea is to offer computing resource as a utility on a pay-per-use basis, similarly to electricity and gas. Thus one no longer needs to invest in infrastructure, run it, maintain it, and secure it, in order to have computer resource at one's disposal. A number of services are possible that fall under the Utility Computing paradigm. *Data-oriented* services offer bulk storage and bandwidth — ideal for backup purposes for instance. *Computation-oriented* services are, however, more common. These offer computational power normally on a per CPU hour basis, well-suited for movie rendering and other computationally intensive applications. At a higher level stand *Application-oriented* services. Here the service provider offers some proprietary software together with the necessary computer hardware on which to run it. Common examples of the applications offered are Customer Relationship Management Software (CRM), Database Management Systems, and e-Accounting software.

Generally the infrastructure is located in a *Data Centre* where it can be managed and maintained easily by the service provider, and the client can access the resource remotely. In

Computation-oriented services (on which we focus mainly in this text) the client can have different degrees of remote access. At one extreme is the *Farm Renting* model, in which the client is allocated a network of machines (alternatively referred to as a Farm) over which he has complete control. At the other extreme is the *Job Submission* model, in which the client is presented with a web interface with which he submits his application together with a control script. The client can then retrieve the results of the computation from the web interface. The Farm Renting model allows the client to debug his applications before and during the job execution, whereas in the Job Submission model the application needs to be free of any bugs. On the other hand, the Job Submission model presents less exposure and hence better security. Moreover, it allows for better use of the infrastructure, because farms need not be allocated in fixed sizes. A *Resource Flexing* scheme can be used to allocate idle resources for job computation during periods of low load. In general this should increase the amount of available resources for when the next job request arrives, thereby maximizing the overall throughput of the infrastructure.

3. Securing the Utility Computing Infrastructure

The Utility Computing market is already a rather competitive market with a number of providers that offer a range of different services. Security is not yet, however, a major issue. Many providers claim that their services are secure but none of them specifies in detail the security measures that they employ. It is almost certain that when utility computing becomes more ubiquitous, a number of security incidents will occur in which clients will experience significant financial losses and service providers will suffer damage to their reputations. So, at a time when the utility computing market will be even more competitive, trust might be the discriminating factor between service providers. A number of security measures are possible in order to secure a utility computing infrastructure. Amongst others, there are: scanning of uploads against malware, encryption and authentication services between machines (through IPsec, TLS, or SSH for instance), farm separation (through routers/firewalls and/or vlans), IDS/IPS/AIS, and system scrubbing. In our analysis, we have focused mainly on system scrubbing, and categorized it on four different levels.

- *System Reuse*: This offers the least amount of protection. Here the machine is presented to the client with a used system where the client is allocated a new user account on the system with the previous users' accounts disabled. Vulnerabilities in the operating system could allow previous users to infect the system with malware, or allow the current user to access previous users' data. In the former case, the risk can be mitigated by scanning the uploaded data. Apart from confidentiality breaches, malware can also degrade the machines' performance. It is worth mentioning that if previous users delete (without overwriting) their data, then it becomes accessible to subsequent users. While if they retain it, it can't be accessed directly by another user.

- *System Refreshing*: Here the client is offered a machine with a freshly installed system. This in principle should eliminate the risk that the system is infected with malware and ensures that the system operates at its maximum performance. However the client may still recover data stored by the previous clients.

- *Clearing*: This refers to the removal of data so that it may not be reconstructed using normal system capabilities (i.e., not through physical access to the media). For this purpose, a single overwrite of all the memory space is normally enough. This process is more intensive in comparison to a system refresh where only the *File Allocation Table* is erased before installing the operating system.

- *Sanitization*: This is intended to protect against data recovery even in the event that the attacker has physical access to the media. This is normally accomplished through multiple overwrites or degaussing.

Clearing and Sanitization do not refer exclusively to secondary storage. Main memory and other memory buffers (such as the network card's memory buffer) may contain sensitive information as well. In [1], the United States Department of Defence defines a Sanitization Matrix which lists procedures suitable for clearing and sanitizing several storage media.

4. Pricing a Utility Computing Service

As in any other utility service, the business model employed and the service pricing play a pivotal role in its success. The business model has to suit the customers' needs in terms of accessibility, scalability, and usability amongst others. Most prominent are the *Subscription* and the *Metered Usage* business models. Hybrids of these models are also common – *Metered Subscription* is commonly employed by Internet Service Providers and Mobile Network Providers. A more detailed discussion of business models for utility computing services can be found in [3].

In [5], Low and Byde propose a pricing strategy based on auction. It is claimed that this scheme should keep a balance between supply and demand. There are, however, some subtle differences between traditional auction-based markets (such as a vegetable market) and the utility computing market. In utility computing, the consumption of goods may stretch over a considerable time span. Thus the present demand would affect the future supply and not just the present one. An auction based market is comparable to a feedback control system. A delay in the feedback loop reduces the system's stability resulting in an oscillatory behaviour which deviates from the equilibrium point. Another difference is that buyers may arrive sparsely in which case a buyer is unlikely to have any competitors or else the bidding stage has to be prolonged thereby degrading the service accessibility. Additionally, an auction scheme fails to take into account the risk incurred in accepting a relatively small job request, and then be unable to fulfil a more sizeable (and more profitable) job request at a later stage. In [4], Paleologo points out the inadequacy of a traditional *Cost-Plus* pricing methodology for utility services. Paleologo suggests a *Price-at-Risk* pricing methodology which takes into account the uncertainty in the pricing decision.

Our contribution is to demonstrate how computer simulation, based on executable models [2], can aid the pricing decision stage. We have built a model of a utility computing service. The model can be used for instance to calculate parameters such as the *Capacity* and the *Multiplexing Gains* used in the Price-at-Risk methodology. It can be used to quantify the gain obtained from a flexing strategy, or the costs of including one of the system scrubbing schemes discussed earlier. In our model, we associate a *Service Level* with each job request. This determines the amount of resources to be allocated for the job. In general the faster a job is computed the better for the client. Thus clients will demand services that require large amounts of resource for relatively short periods. Such a demand profile tends to reduce the effective capacity of the infrastructure. In view of this we recognise some appropriate properties for a just pricing scheme.

- It should encourage a constant load and discourage an erratic load pattern.
- The cost to compute a job should be proportionate to its size.

- The cost to compute a job should increase as the quality of service improves.

We have implemented two similar pricing schemes that follow these guidelines.

Scheme #1

A price is paid per CPU-hour which depends on the Service Level. The Service Level in turn determines how many machines (on average) will be allocated for the job. A possible flaw of this scheme is that a client can split his job into smaller ones, submit them simultaneously as separate jobs, and process them at a lower quality of service. Hence he effectively gets the same Service Level at a lower price. For instance assume that:

Service Level	Machines Allocated	Cost per CPU-hour
1	10	£1.00
2	50	£1.20

Table 1.

Client A submits a 1000 CPU-hour job at Service Level 2. His job takes 20 hours and is charged 1200 Pounds.

Client B splits his 1000 CPU-hour job into five 200 CPU-hour jobs and submits them simultaneously at Service Level 1. This takes 20 hours and he is charged only 1000 Pounds.

Scheme #2

A price is paid per CPU-hour which depends on the Service Level. The Service Level determines the amount of time taken to compute the job. So the number of allocated machines depends also on the job size. Of course, not every job request can be supplied with the highest Service Level. Bigger jobs get better value for money. This might seem unfair but is reasonable if we consider that the client should always get a better service than he would get if he were to invest in infrastructure of his own.

5. The Model

The model was implemented using the Demos 2000 modelling language [2]. The code for the model is given in the Appendix. Demos 2000 is a semantically justified modelling language developed by Birtwistle, Christodolou, Taylor, and Tofts [2]. Our model consists of four main entities: Clients, Farms, Scrubbing Processes, and the Allocator.

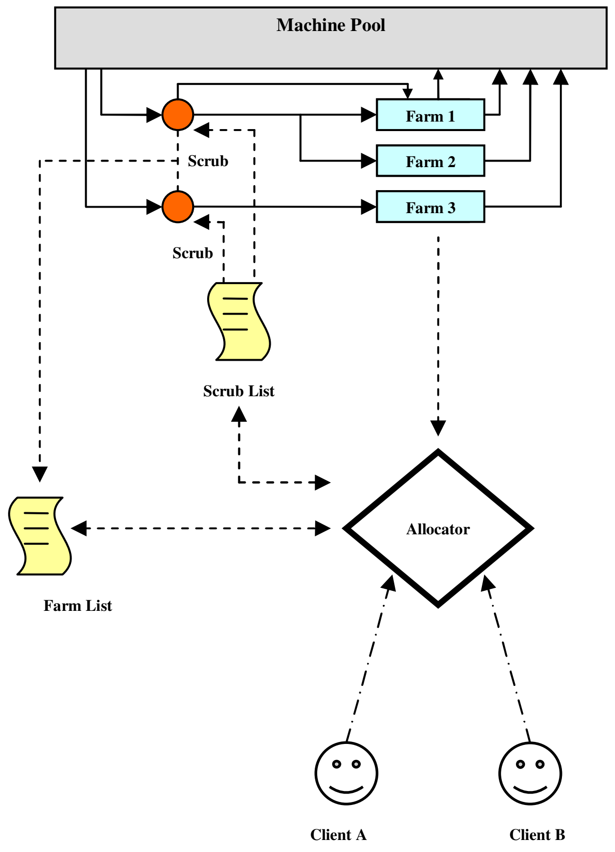

Figure 1.

The client's sole purpose is to generate job requests. Each client makes requests independently, where the time interval between each request is governed by an exponential distribution. A job request consists of two parameters: a job size W in CPU-hours and a Service Level QoS. The QoS is interpreted in accordance with the pricing scheme employed, but in general the higher the QoS the faster the job computation.

Farms are created by the Allocator entity in response to successful job requests from clients. Each farm corresponds to a single job, after which the farm is released on job completion. The farm process is implemented as a loop which continuously waits for instructions (in the form of syncs) mainly from the Allocator. Five instructions are defined.

1. *Lease Expired* signals that the time allocated for the job has come to an end. In response to this, the farm releases its machines and halts.
2. *Work Completed* signals the event that the job has been completed, in which case the farm releases its machines but continues to exist until a Lease Expired message is received. A state parameter is included to indicate the freshness of the message. The signal is fresh if the state parameter matches the current state of the farm.
3. *Resource Instruction* instructs the farm to take or release a number of machines. A Resource Instruction increments the current state of a farm and reschedules a new Work Completed instruction.
4. *Flexing Query* interrogates the farm for any flexing machines that it currently holds and can return back to the Allocator in order to fulfill job requests.
5. *Work Query* interrogates the farm for the amount of work in CPU hours needed to complete the job. This information is used by the Allocator to determine to which farm the idle machines should be allocated for flexing.

Scrubbing Processes are created by the Allocator in order to scrub a set of machines, before they are allocated to a farm. Machines are grabbed from the resource pool and held by a Scrubbing Process for a period of time defined by the model parameter *scrubTime*. Each of the four scrubbing schemes described earlier can be modeled by varying this parameter. On completion the Scrubbing Process would consult the *Scrub List* to determine to which farm(s) it should forward the machines. Finally the Scrubbing Process would release the machines, send the corresponding *Resource Instruction* signals to the intended farms, and terminate.

The *Allocator* is the central entity in the model, and controls most of the remaining entities. Its main responsibilities are: processing job requests, managing and flexing resources, and maintain records (such as the *Farm List* and the *Scrub List*). The flexing strategy adopted in the model uses a single-farm flexing algorithm. In the event that a set of machines are idle, the Allocator consults the Farm List and interrogates every farm with a Work Query. The idle machines are all allocated to the farm with the longest life expectancy. The life expectancy L of a farm F having a workload W_F and M_F machines is given by:

$$L = (W_F - M_F \times \mathrm{T_S}) / (M_F + M_I)$$

where M_I = Idle Machines and T_s = Scrub Time. The scheme is zero-risk as each farm is always allocated enough machines to complete the job in the allotted time span.

The basic operation of the model is depicted in Figure 1, showing the flow of information between entities and how machines progress through their cycle. Clients submit job requests (W, QoS) to the Allocator. The Allocator determines the number of machines required to fulfill the request and checks how many machines are available. It starts by checking how many machines are idle. If these are not enough it consults the Scrub List for any machines destined for flexing. If the machines destined for flexing together with the idle machines do not add up to the required amount, the Allocator consults the Farm List for farms which have flexing machines and sends them a Flexing Query. The farms reply with the amount of machines that they can release (and still complete their job in time). If the total number of machines is enough to fulfill the request a farm is allocated, otherwise the job request is denied. A Scrubbing Process is created to clean the idle machines and the machines retrieved from other farms, while the machines retrieved from other Scrubbing Processes are redirected to the new farm by amending the Scrub List. The Allocator loops indefinitely waiting for job requests and other messages (such as job completion notifications). Each time it is inquired it checks for any idle machines and goes through the flexing subroutine.

6. Results

The model has various potential applications. Some of these are:

- To determine the probability that an amount of machines will be enough to cater for a certain demand distribution;
- To calculate the gain in capacity that can be attained by a particular flexing strategy;
- To determine how distinct Service Levels should be priced;
- To quantify the impact of implementing a scrubbing scheme in a flexed architecture;
- To estimate parameters necessary for the Price-at-Risk methodology;
- To provide insight for SLA design.

QoS	Job Duration	Probability	Price
1	6 Months	0.05	£ 0.10
2	4 Months	0.15	£ 0.14
3	2 Months	0.15	£ 0.20
4	1 Month	0.20	£ 0.30
5	2 Weeks	0.25	£ 0.40
6	1 Week	0.10	£ 0.60
7	3 Days	0.06	£ 0.90
8	1 Day	0.04	£ 1.50

Table 2.

The sample model found in the Appendix was run a number of times, each with a different amount of machines. The Pricing scheme used was that described in scheme #2 where the price of each Service Level is listed in Table 2. The probability associated with each Service Level is the probability that a job request demands that Service Level. Figure 2 depicts a plot of the revenue attained by each run as the number of machines is increased. The experiment was repeated for a Scrub Time value of 1 hour instead of 5 hours, and once again with no flexing. Figure 3 represents the same three experiments but the abscissa represents the percentage of jobs denied.

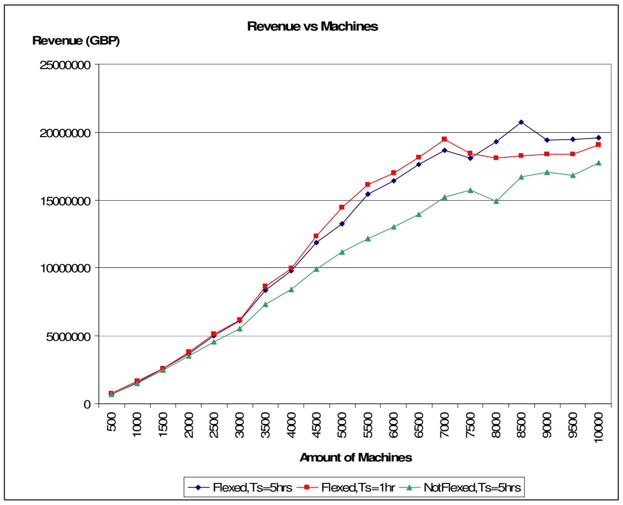

Figure 2.

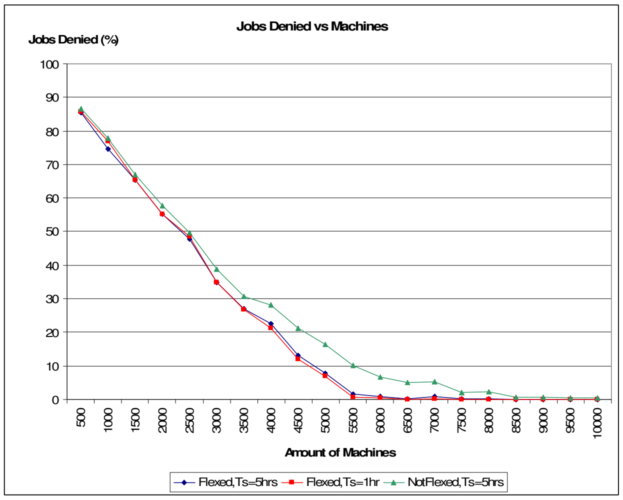

Figure 3.

It can be seen from Figure 2 that the red and blue plots attain the maximum possible revenue while the green plot approaches this maximum more slowly. A small increase in revenue due to a reduced Scrub Time is also evident. Instead of contrasting the revenue attained by each configuration, we can make a comparison in terms of investment. In particular we can compare the amount of infrastructure required to fulfill the same amount of requests and hence attain the same revenue. For instance from Figure 3 we can see that for a flexed configuration with 5000 machines the job denial ratio is 7.9 %. On the other hand the non-flexed configuration requires 6000 machines to attain an almost equivalent job denial ratio of 6.7 %. Hence in this scenario an increase in investment of 20% is required to attain the same revenue as a flexed architecture.

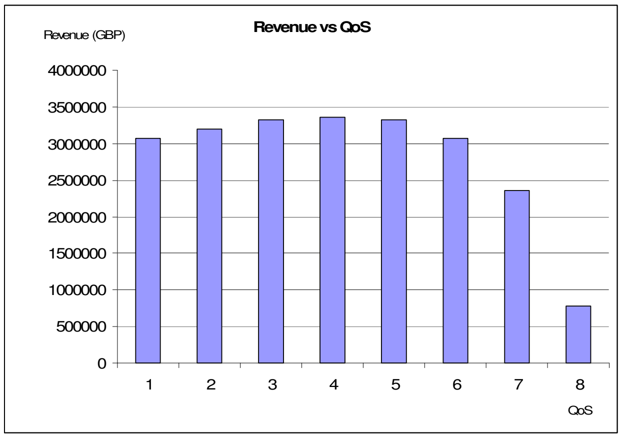

Figure 4.

As mentioned earlier, high QoS values diminish the effective capacity of the infrastructure. This, of course, should be reflected in the service pricing. Our model can be used to quantify the effective capacities attained by each QoS, and hence determine a fair price for each Service Level. The model was adjusted such that each QoS is priced at 10p/CPUhr. Then it was run eight times, where in each run all job requests were processed at one particular Service Level. Figure 4 shows the revenue attained for each Service Level. As a starting point for a fair pricing scheme, we could adjust the price of each Service Level such that all bars attain the same revenue. For instance according to these results, Service Level 8 should cost three times as much as Service Level 7.

A rather unexpected effect, portrayed in Figure 4, is that in going from Service Level 4 down to Level 1, the revenue is seen to decrease. In our model the client pays for the service on job completion. The simulation runtime here is two years, and the job processing times vary from 1 day to 6 months, depending on the Service Level. Thus, at lower Service Levels there is a greater amount of work which is not yet paid for at the end of the run. This situation might amount to a serious cash-flow problem. On top of this, one can add inflation and maintenance costs, which also increase as computation intervals get longer. So, if clients are to pay on job completion, it may not be viable to offer Service Levels between 1 and 4. Alternatively, one could circumvent this problem by changing to a business model in which the client pays either in advance or periodically until the job is completed.

7. Related Work

A good summary of the various pricing and business models that have been proposed for utility computing can be found in [7]. Oceano, described in [8], is a utility computing infrastructure prototype developed at IBM. Including features such as scrubbing and flexing, Oceano proves to be quite in line with our utility computing model, although it is intended for web services rather than computation services. Yu, Buyya, and Tham [6] suggest a cost-based scheduling scheme in order to improve the internal workflow of a job. Potentially there might be scope to augment this scheme with a flexing strategy.

8. Conclusions and Future Work

We demonstrated some of the potential that discrete event modelling holds for economic studies in relation to Information Technology and Information Security. In this paper we examined the case of Utility Computing, and showed how our simple model can help in business decisions, as well as exploring a bigger fraction of the possibility space.

In our analysis we aimed to keep the model as simple as possible. Two issues that are overlooked in this model are *Farm Fragmentation* and the *Atomic Transaction* nature of distributed jobs. The flexing algorithm moves machines from farm to farm without any notion of machine location, resulting in Farm Fragmentation. Mainly this is due to the difficulty in associating location with resource elements in DEMOS 2000. Thus our model does not represent the network traffic overhead which may result from a particular flexing algorithm. Secondly, distributed jobs are normally composed of smaller atomic transactions. However in the model it is assumed that job computation can be split into arbitrarily small transactions. By stepping the Allocator entity in discrete time and some other modifications it should be easy to include this in the model. Another limitation is that the implemented flexing strategy assumes jobs are arbitrarily distributable. Unfortunately this applies only to a limited set of problems, such as an exhaustive search of a cryptographic key. Therefore the 20% increase in infrastructure utilization mentioned in section 6 is essentially an upper bound of the flexing gain that the implemented flexing algorithm can attain. Thus if the cost of implementing such a flexing scheme is more than the cost of increasing the computing infrastructure by 20%, then such a scheme is obviously not viable.

Possible directions for future work could be to amend the model to portray these factors. The flexing strategy could be upgraded to a multiple-farm flexing scheme where idle machines are distributed among multiple farms rather than one farm. Risky flexing, where some jobs may not be completed on time, can also be investigated. The model could be amended to include maintenance and upgrading costs for a more thorough business continuity study. Jobs could be portrayed by a more general model, such as the one presented in [6]. Finally, it should be noted that the model can be easily adapted to portray other utility computing services, such as Web Services.

Acknowledgements

We are grateful to Chris Dalton, Brian Monahan, Chris Tofts, and Mike Yearworth for their advice on various aspects of this paper.

References

[1] DoD 5220.22-M *National Industrial Security Program Operating Manual (NISPOM)* available from: http://www.usaid.gov/policy/ads/500/d522022m.pdf

[2] A. Christodolou, R. Taylor, and C. Tofts, 2000. Demos 2000. http://www.demos2k.org

[3] M. A. Rappa, "The utility business model and the future of computing services", *IBM Systems Journal*, Vol. 43, No 1. 2004, available from: http://www.research.ibm.com/journal/sj/431/rappa.html

[4] G. A. Paleologo, "Price-at-Risk: A methodology for pricing utility computing services", *IBM Systems Journal*, Vol. 43, No 1. 2004, available from: http://www.research.ibm.com/journal/sj/431/paleologo.html

[5] C. Low and A. Byde, "Market-Based Approaches to Utility Computing", *Technical Report HPL-2006-23, HP Laboratories, Bristol*, available from:

http://www.hpl.hp.com/techreports/2006/HPL-2006-23.pdf

[6] Jia Yu, Rajkumar Buyya, and Chen Khong Tham, “Cost-based Scheduling of Scientific Workflow Applications on Utility Grids”, *Proceedings of the 1st International Conference on e-Science and Grid Computing* (e-Science 2005), December 2005, pp. 140-147.

[7] Buyya R, Abramson D, Giddy J, Stockinger H. “Economic models for resource management and scheduling in Grid computing”, *Concurrency and Computation: Practice and Experience* 2002; 14(13–15):1507–1542.

[8] K. Appleby, S. Fakhouri, L. Fong, G. Goldszmidt, M. Kalantar, S. Krishnakumar, D. Pazel, J. Pershing, and B. Rochwerger, “Oceano – SLA Based Management of A Computing Utility,” in *7th IFIP/IEEE Intl. Symp. on Integrated Network Management*, May. 2001, pp. 855–868.

Appendix

```
//=====================================================================
// Utility Computing Model: Allocator Version 2.5c (Single Farm Flexing
// + Revenue + Pricing Scheme #2)
//=====================================================================
//
// Jean Paul Degabriele David Pym
//
// 7th March 2007



// Basic Model Framework
// =====================
//
// Clients submit job requests of a certain size in CPU-hours,
// and a maximum computation time, determined by the QoS.
// The Resource Allocator evaluates the request and if it can
// be satisfied a farm is allocated to compute the job. In order
// to maximise resource utilisation, the Resource Allocator is
// allowed to flex the resources between farms.
//
//
// Notes specific to this version
// ==============================
//
// The flexing algorithm is very crude. The free machines are allocated
// as a chunk to one farm only. Each client submits job requests of a
// certain size in CPU-hours, and specifies a maximum computation time.
// The allocator flexes the resources so as to maximise their
// utilisation.



// ========================
// Parameter Initialisation
// ========================


// Scaling Constants (hrs = timing unit)
// =====================================

   cons hrs    = 1;
   cons mins   = hrs/60;
   cons secs   = mins/60;
   cons msecs  = secs/1000;

   cons days   = 24 * hrs;
   cons weeks  = 7 * days;
   cons months = 28 * days;
   cons years  = 365 * days;
```

```
// Parameter Constants
// ===================

   cons null = 0;

   cons instrLeaseExpired = 1;
   cons instrWorkCompleted = 2;

   cons takeMachines = 3;
   cons releaseMachines = 4;


// Model Parameters
// ================

   cons runtime      = 1 * years;
   cons numMachines  = 5000;
   cons numClients   = 10;

   cons scrubTime    = 5 * hrs;


// Stochastic Parameters
// =====================

   cons requestInterval = negexp(100 * hrs);
   cons requestSize = puni(10000, 100000); // Job Size (work in CPUhrs)
   cons serviceLevel = pud[(0.05,1),(0.15,2),(0.15,3),(0.2,4),(0.25,5),
   (0.1,6),(0.06,7),(0.04,8)];      // Quality of Service


// Pricing Parameters
// ==================

   cons QoSPrice[1] = 0.10;
   cons QoSPrice[2] = 0.14;
   cons QoSPrice[3] = 0.20;
   cons QoSPrice[4] = 0.30;
   cons QoSPrice[5] = 0.40;
   cons QoSPrice[6] = 0.60;
   cons QoSPrice[7] = 0.90;
   cons QoSPrice[8] = 1.50;

   cons QoSTime[1] = 6 * months;
   cons QoSTime[2] = 4 * months;
   cons QoSTime[3] = 2 * months;
   cons QoSTime[4] = 1 * months;
   cons QoSTime[5] = 2 * weeks;
   cons QoSTime[6] = 1 * weeks;
   cons QoSTime[7] = 3 * days;
   cons QoSTime[8] = 1 * days;


// Universal Variables
// ===================

   var t = 0;
   var clock = 0;
   var idleMachines = numMachines;
   var scrubbingMachines = 0;
   var requestsSubmitted = 0;
   var jobsCompleted = 0;
   var jobsDenied = 0;
   var revenue = 0;


// Resources
// =========

   res(machines, numMachines);    // current pool of unassigned machines
                                  // available for work.
   res(lockIM, 1);
   res(lockSM, 1);
   res(lockRS, 1);
   res(lockJC, 1);
   res(lockJD, 1);
   res(lockA, 1);      // group of mutually exclusive transactions


// Bins
// ====

   bin(farmList, 0);
   bin(scrubList, 0);


// =================
// Class Definitions
// =================


class client (cid) = {

  local var work    = 0;          // Amount of CPU Hrs
  local var QoS     = 0;

  repeat {

    hold(requestInterval);

    work := requestSize;
    QoS := serviceLevel;

    syncV(jobRequest, [cid, work, QoS], []);
  }
}



class farmProcess(owner, argFid, work, argTmax) = {

// After job completion the farm is kept alive with state stop == 1
// in order to cater for any 'takeMachine' instructions from ScrubProcesses
```

```
  local var startTime = DEMOS_TIME;
  local var lastTime = startTime;
  local var workRemaining = work;
  local var lifeExpectancy  = 0;
  local var farmMachines  = 0;
  local var flexMachines = 0;
  local var stop = 0;
  local var currentState = 1;

  local var recipient = 0;
  local var state = 0;
  local var action = 0;
  local var amount = 0;

  local var fid = argFid;
  local var Tmax = argTmax;


  entity(LEASEEXPIRED, scheduleInstr(#fid, instrLeaseExpired, null, #Tmax), 0);
//schedule lease expiry

  while [stop < 2]
  {


    try[getSv(leaseExpired, [recipient], recipient == fid)] then
    {
      try [stop < 1] then
      {
        putR(machines, farmMachines);
        putSV(leaseExpired, [farmMachines]);
      }
      etry[] then {putSV(leaseExpired, [0]);} //farm was already released

      stop := 2;
    }



    etry[getSv(workCompleted, [recipient, state], recipient == fid)] then
    {
      try[currentState == state] then // check instruction is valid
      {
        getR(lockJC, 1); jobsCompleted := jobsCompleted + 1; putR(lockJC, 1);
        putR(machines, farmMachines);
        putSV(workCompleted, [1, farmMachines]);
        farmMachines := 0;
        stop := 1;
      }
      etry[] then {putSV(workCompleted, [0, 0]);}
    }



    etry[getsV(resInstr, [recipient, action, amount], recipient == fid)] then
    {
      workRemaining := workRemaining - (farmMachines * (DEMOS_TIME - lastTime));
//recalculate remaining work
      lastTime := DEMOS_TIME;


      try [action == takeMachines] then
      {
        try [stop == 0] then
        {getR(machines, amount); farmMachines := farmMachines + amount;}

        // If farm has been released, then forward machines to pool and start
Allocator's flexing routine
        etry [] then {getR(lockIM, 1); idleMachines := idleMachines + amount;
putR(lockIM, 1); syncV(flex, [], []);}
      }

      etry [] then
      {
        putR(machines, amount);
        farmMachines := farmMachines - amount;
      }

      currentState := currentState + 1;


      try [(farmMachines > 0) && (stop < 1) ] then
      {
        lifeExpectancy := workRemaining / farmMachines;
        entity(WORKCOMPLETED, scheduleInstr(#fid, instrWorkCompleted,
#currentState, #lifeExpectancy), 0);
      }
      etry [] then {}

      putSV(resInstr, []);

    }



    etry[getsV(flexingQuery, [recipient], recipient == fid)] then
    {
      workRemaining := workRemaining - (farmMachines * (DEMOS_TIME - lastTime));
//recalculate remaining work
      lastTime := DEMOS_TIME;

      flexMachines := farmMachines - (workRemaining / (startTime + Tmax -
DEMOS_TIME));
      try [flexMachines - rnd(flexMachines) > 0] then {flexMachines :=
rnd(flexMachines);} etry [] then {flexMachines := rnd(flexMachines) - 1;}
      putSV(flexingQuery, [flexMachines]);
    }



    etry[getsV(workQuery, [recipient], recipient == fid)] then
    {
      workRemaining := workRemaining - (farmMachines * (DEMOS_TIME - lastTime));
//recalculate remaining work
      lastTime := DEMOS_TIME;

      putSV(workQuery, [workRemaining, farmMachines]);
    }
  }
}



class scrubMachines(id, amount) = {

  local var stop = 0;
  local var sid = 0;
  local var fid = 0;
  local var fidList = 0;
  local var flexing = 0;
  local var flexList = 0;
  local var cost = 0;
  local var amt = 0;

  getR(lockSM, 1); scrubbingMachines := scrubbingMachines + amount;
      putR(lockSM, 1);
  getR(machines, amount);
  hold(scrubTime);
  getR(lockSM, 1); scrubbingMachines := scrubbingMachines - amount;
      putR(lockSM, 1);
  putR(machines, amount);

  getR(lockA, 1);

  while [getVB(scrubList, [sid, fid, flexing, amt], sid == id)]
  {
    syncV(resInstr, [fid, takeMachines, amt], []);

    try [getVB(farmList, [fidList, flexList, cost], fidList == fid)] then
{flexList := (flexList + flexing) - (flexList * flexing); putVB(farmList,
[fidList, flexList, cost]);}
    etry [] then {} // farm does not exist anymore, machines are forwarded to the
machine pool
  }

  putR(lockA, 1);
}



class scheduleInstr(recipient, instruction, parameter, delay) = {              //
schedule farm instructions

  local var reply = 0;

  hold(delay);

  try [instruction == instrLeaseExpired] then
  {syncV(aLeaseExpired, [recipient], []);}
  etry [instruction == instrWorkCompleted] then
  {syncV(aWorkCompleted, [recipient, parameter], []);}

  etry [] then {}
}



class numberService = {
  local var num = 100 + numClients;

  repeat {
    getSV(nextID, [], true);
      num := num + 1;
    putSV(nextID, [num]);
  }
}



class monitor = {

  local var lastIdleMachines = idleMachines;
  local var idleWork = 0;

  repeat {

    hold(1);
    clock := clock + 1;

    trace("Monitoring at time instant %v", clock);
    trace("idleMachines=%v", idleMachines);
    trace("scrubbingMachines=%v", scrubbingMachines);
    trace("requestsSubmitted=%v", requestsSubmitted);
    trace("jobsDenied=%v", jobsDenied);
    trace("jobsCompleted=%v", jobsCompleted);
    trace("revenue=%v", revenue);

    idleWork := idleWork + (idleMachines + lastIdleMachines)/2;
    lastIdleMachines := idleMachines;
    trace("idleFraction =%v", idleWork/(numMachines * clock));
  }
}



class allocator = {

  local var Tmax = 0;
  local var requestedMachines = 1;

  local var id = 0;
  local var state = 0;
  local var valid = 0;
  local var work = 0;
  local var QoS = 0;
  local var sid = 0;
  local var fid = 0;
  local var fidNew = 0;
  local var fidFlex = 0;

  local var flexing = 0;
  local var cost = 0;
  local var amount = 0;
  local var farmMachines = 0;
  local var farmFlexMachines = 0;
  local var farmWorkLoad = 0;
  local var lifeExpectancy = 0;
  local var maxLifeExpectancy = 0;

  local var flexMachines = 0;
  local var scrubMachines = 0;


  putVB(farmList, [0, 1, 0]);           // dummy farm => list is never empty &
                                        // acts as a marker; fid = 0, flexing = 1
  putVB(scrubList, [0, 0, 1, 0]);       // dummy scrub process; sid = 0, fid = 0,
                                        // flexing = 1, amount = 0



  repeat
  {

  // Check for Job Requests
  // =====================


    try [getSV(jobRequest, [id, work, QoS], true)] then
    {
      getR(lockRS, 1); requestsSubmitted := requestsSubmitted + 1;
      putR(lockRS, 1);

      // INITIALISE VARIABLES

      Tmax := QoSTime[QoS] + scrubTime;
      requestedMachines := rnd(work / QoSTime[QoS]) + 1;
      flexMachines := 0;
      scrubMachines := 0;


      // CHECK MACHINE AVAILABILITY

      getR(lockA, 1);
      try [idleMachines < requestedMachines] then
      {
      getVB(scrubList, [sid, fid, flexing, amount], sid == 0); //put marker at the
end
      putVB(scrubList, [sid, fid, flexing, amount]);

      sid := 1;
        while [sid != 0]
        {
        getVB(scrubList, [sid, fid, flexing, amount], flexing == 1);
        scrubMachines := scrubMachines + amount;
        putVB(scrubList, [sid, fid, flexing, amount]);
      }
      }
      etry [] then {}


      try [idleMachines + scrubMachines < requestedMachines] then
      {
      getVB(farmList, [fid, flexing, cost], fid == 0); //put marker at the end
        putVB(farmList, [fid, flexing, cost]);

      fid := 1;
        while [fid != 0]
        {
        getVB(farmList, [fid, flexing, cost], flexing == 1);
          putVB(farmList, [fid, flexing, cost]);

        try [fid != 0] then
          {
            syncV(flexingQuery, [fid], [farmFlexMachines]);
          flexMachines := flexMachines + farmFlexMachines;
          }
          etry [] then {}
        }
      }
      etry[] then {}



  // IF REQUEST CAN BE FULFILLED THEN CREATE FARM & ALLOCATE MACHINES

      try [idleMachines + scrubMachines + flexMachines >= requestedMachines] then
      {
      syncV(nextID, [], [fidNew]);
      entity(FARMPROCESS, farmProcess(#id, #fidNew, #work, #Tmax), 0); // id = cid
      putVB(farmList, [fidNew, 0, (work * QoSPrice[QoS])]);


      try [idleMachines >= requestedMachines] then
      {syncV(nextID, [], [sid]); entity(SCRUBMACHINES, scrubMachines(#sid,
#requestedMachines), 0); putVB(scrubList, [sid, fidNew, 0, requestedMachines]);
getR(lockIM, 1); idleMachines := idleMachines - requestedMachines; putR(lockIM,
1); requestedMachines := 0;}

      etry [idleMachines < requestedMachines && idleMachines > 0] then
      {syncV(nextID, [], [sid]); entity(SCRUBMACHINES, scrubMachines(#sid,
#idleMachines), 0); putVB(scrubList, [sid, fidNew, 0, idleMachines]); getR(lockIM,
1); idleMachines := 0; putR(lockIM, 1); requestedMachines := requestedMachines -
idleMachines;}

      etry [] then {}


      try [requestedMachines > 0 && scrubMachines > 0] then
      {
        getVB(scrubList, [sid, fid, flexing, amount], sid == 0); //put marker at
the end
        putVB(scrubList, [sid, fid, flexing, amount]);

        sid := 1;
          while [sid != 0 && requestedMachines > 0]
          {
          getVB(scrubList, [sid, fid, flexing, amount], flexing == 1);

          try [sid != 0] then
            {
              try [amount <= requestedMachines] then
            {putVB(scrubList, [sid, fidNew, 0, amount]); requestedMachines :=
requestedMachines - amount;} // delete original record

            etry [] then
            {putVB(scrubList, [sid, fidNew, 0, requestedMachines]);
putVB(scrubList, [sid, fid, 1, amount - requestedMachines]); requestedMachines :=
0;}
            }
            etry [] then {putVB(scrubList, [sid, fid, flexing, amount]);}
        }
      }
      etry [] then{}


      try [requestedMachines > 0] then
      {
        getVB(farmList, [fid, flexing, cost], fid == 0); //put marker at the end
          putVB(farmList, [fid, flexing, cost]);

        fid := 1;
          while [fid != 0 && requestedMachines > 0]
          {
          getVB(farmList, [fid, flexing, cost], flexing == 1);

          try [fid != 0] then
            {
              syncV(flexingQuery, [fid], [farmFlexMachines]);

            try [farmFlexMachines <= requestedMachines && farmFlexMachines > 0]
then
            {syncV(resInstr, [fid, releaseMachines, farmFlexMachines], []);
syncV(nextID, [], [sid]); entity(SCRUBMACHINES, scrubMachines(#sid,
#farmFlexMachines), 0); putVB(scrubList, [sid, fidNew, 0, farmFlexMachines]);
requestedMachines := requestedMachines - farmFlexMachines; putVB(farmList, [fid,
0, cost]);}

            etry [farmFlexMachines > requestedMachines] then
            {syncV(resInstr, [fid, releaseMachines, requestedMachines], []);
syncV(nextID, [], [sid]); entity(SCRUBMACHINES, scrubMachines(#sid,
#requestedMachines), 0); putVB(scrubList, [sid, fidNew, 0, requestedMachines]);
requestedMachines := 0; putVB(farmList, [fid, 1, cost]);}

              etry [] then {putVB(farmList, [fid, 0, cost]);}
            }
            etry [] then {putVB(farmList, [fid, flexing, cost]);}
          }
      }
      etry [] then {}

      }
      etry[] then{getR(lockJD, 1); jobsDenied := jobsDenied + 1; putR(lockJD, 1);}

      putSV(jobRequest, []);

      putR(lockA, 1);
    }



  // Check for Farm Expiry
  // =====================


    etry [getSv(aLeaseExpired, [id], true)] then
    {
      getR(lockA, 1);

      try [getVB(farmList, [fid, flexing, cost], fid == id)] then {}
      //remove farmList entry unless already removed
      etry [] then {}

      syncV(leaseExpired, [id], [amount]);
      getR(lockIM, 1); idleMachines := idleMachines + amount; putR(lockIM, 1);

      putR(lockA, 1);

      putSV(aLeaseExpired, []);
    }



  // Check for Farm Work Completion
  // ==============================


    etry [getSv(aWorkCompleted, [id, state], true)] then
    {
      getR(lockA, 1);

      try[getVB(farmList, [fid, flexing, cost], fid == id)] then
      {
        syncV(workCompleted, [id, state], [valid, amount]);
        try [valid == 1] then
        {
          getR(lockIM, 1); idleMachines := idleMachines + amount; putR(lockIM, 1);
          revenue := revenue + cost;                  //record revenue
                                                     // & delete farmList entry
        }
        etry [] then {putVB(farmList, [fid, flexing, cost]);}   //invalid msg
      }
      etry [] then {} // farm has been released already!
      putR(lockA, 1);
      putSV(aWorkCompleted, []);
    }




  // Check for Farm Liveness Queries
  // ===============================


    etry [getSV(farmAlive, [id], true)] then
    {
      getR(lockA, 1);

      try [getVB(farmList, [fid, flexing, cost], fid == id)] then
      {putVB(farmList, [fid, flexing, cost]); putSV(farmAlive, [1]);}

      etry [] then
      {putSV(farmAlive, [0]);}

      putR(lockA, 1);
    }



  // Dummy Sync to start Flexing Routine
  // ===================================


    etry [getSV(flex, [], true)] then {putSV(flex, []);}



  // After that an Allocator msg has been processed, allocate any free machines
  // for flexing. (This can be optimised further by allocating the machines to
  // more than one farm.)
  // ==========================================================================


    try [idleMachines > 0] then
    {
      getR(lockA, 1);

      getVB(farmList, [fid, flexing, cost], fid == 0); //put marker at the end
      putVB(farmList, [fid, flexing, cost]);

      // CHOOSE THE FARM WITH THE LONGEST FLEXING TIME ESTIMATE

      lifeExpectancy := 0;
      maxLifeExpectancy := 0;
      fidFlex := 0;

      fid := 1;
      while [fid != 0]
      {
        getVB(farmList, [fid, flexing, cost], true);
        putVB(farmList, [fid, flexing, cost]);

        try [fid != 0] then
        {
          syncV(workQuery, [fid], [farmWorkLoad, farmMachines]);

          try [(farmMachines > 0)&&(farmWorkLoad/farmMachines > scrubTime)] then
// check farm is still alive after scrubbing
          {
            lifeExpectancy := (farmWorkLoad - farmMachines * scrubTime) /
(farmMachines + idleMachines);
            try [lifeExpectancy > maxLifeExpectancy] then {fidFlex := fid;
maxLifeExpectancy := lifeExpectancy;}
            etry [] then {}
          }
          etry [] then {}
        }
        etry [] then {}
      }

      putR(lockA, 1);

      // SCRUB MACHINES AND FLEX

      try [fidFlex != 0] then
      {
        syncV(nextID, [], [sid]);
        entity(SCRUBMACHINES, scrubMachines(#sid, #idleMachines), 0);
        putVB(scrubList, [sid, fidFlex, 1, idleMachines]);
        getR(lockIM, 1); idleMachines := 0; putR(lockIM, 1);
      }
      etry [] then {}
    }
    etry [] then {}
  }
}



// ==============
// Run Simulation
// ==============

entity(MONITOR, monitor, 0);
entity(ALLOCATOR, allocator, 0);
t := 101; do numClients {entity(CLIENT, client(#t), 0); t := t + 1;}
entity(NUMBERSERVICE, numberService, 0);

hold(runtime);

close;
```