

Dynamic Allocation of Servers in a Grid Hosting Environment

Mike Fisher¹, Charles Kubicek², Paul McKee¹, Isi Mitrani², Jennie Palmer² and Rob Smith²

Abstract—A grid hosting environment is described, where servers may be reconfigured dynamically from one type of work to another in response to changes in demand. The problem of carrying out these reconfigurations in the most efficient manner is addressed by means of stochastic modelling and optimization. A heuristic policy which is close to optimal over a wide range of parameters is introduced. A prototype system based on existing resource management software has been developed to demonstrate the concepts described.

Index Terms—Reconfigurable servers, Optimal resource allocation, Grid computing, Dynamic programming, Heuristic policies.

I. INTRODUCTION

In a Grid environment, heterogeneous pools of servers provide a variety of services to widely distributed user communities. Users submit jobs without necessarily knowing, or caring, where they will be executed. The system distributes those jobs among the servers, attempting to make the best possible use of the available resources and provide the best possible quality of service.

The random nature of user demand, and also changes of demand patterns over time, can lead to temporary oversubscription of some services, and underutilization of others. In such situations, it could be advantageous to reallocate servers from one type of provision to another, even at the cost of switching overheads. The question that arises in that context is how to decide whether, and if so when, to perform such reconfigurations.

Consider a system consisting of N servers (possibly geographically distributed), split into M heterogeneous pools of sizes K_1, K_2, \dots, K_M , where $\sum_{i=1}^M K_i = N$. Pool i is dedicated to a queue of jobs of type i ($i = 1, \dots, M$). Examples of job types may include short web accesses, long database searches, streamed media, etc. Different job types may have different Quality of Service (QoS) requirements; e.g., some may be less tolerant of delays than others). It is possible to reassign any server from one queue to another, but the process is generally not instantaneous and during it the server becomes unavailable. In those circumstances, a dynamic reconfiguration policy would specify, for any given parameter set (including costs), and current system state, whether or not to switch a server from queue i to queue j .

This paper addresses both the realization of the above system, and the determination of optimal, or near-optimal, dynamic reallocation policies. To our knowledge, the problem

has not been posed and handled in this way before. There are some commercial products ([7], [14]) which attempt to increase server utilization through policy driven resource management; however, they are not concerned with job types and QoS requirements. Chase et al. [3] consider the possibility of switching servers between services to cope with demand. They propose an ad-hoc policy which does not take either costs or QoS requirements into account.

There is a body of work on optimal server allocation, but it is mostly in the context of polling systems, where one server visits several queues in a fixed or variable order, with or without switching overheads (see [5], [6], [8], [9], [11]). Even in those cases of a single server, it has been observed by both Duenyas and Van Oyen [5], [6], and Koole [8], [9], that the presence of non-zero switching times makes the optimal policy very difficult to characterize explicitly. This necessitates the consideration of heuristic policies. The only general result available for multiprocessor systems applies when the switching times and costs are zero: then the $c\mu$ -rule is optimal, i.e. the best policy is to give absolute preemptive priority to the job type for which the product of holding cost and service rate is largest (Buyukkoc et al [2]). A preliminary report on the mathematical formulation of the problem discussed here, without the architectural and implementation aspects, was given by two of the present authors in [12].

The proposed architectural design, and the implemented prototype, are described in section 2. The optimization problem, its solution, and the construction of the heuristic policy are presented in section 3, while section 4 contains some simulation results comparing different policies, including the optimal and the heuristic.

II. POOL PARTITIONING AND SWITCHING

The aim of the system architecture is to allow servers, partitioned into conceptual pools, to switch from one pool to another. The switching decisions are governed by a switching policy, which in turn uses system metrics gathered over time. Networking technology allows a Resource Management System (RMS) to control servers over a wide area network, so a conceptual pool may contain servers in different geographical locations, provided that they all run the same reconfiguration software. The main components needed to build the pool partitioning architecture are described below. One machine may play any, or all, roles at any given time.

- **Server manager.** Each individual computational server runs the RMS software, and a daemon that allows the server to be instructed to switch to another pool.
- **Pool manager.** This component manages a conceptual pool. It runs an instance of the RMS in a central manager

¹ BT Exact, Adastral Park, Ipswich, IP5 3RE, UK

² School of Computing Science, University of Newcastle, NE1 7RU, UK
Contact e-mail: jennie.palmer@ncl.ac.uk

mode, queueing and scheduling jobs on servers in its control. The pool manager keeps track of the servers in its pool, helps to coordinate switches of servers to and from other pools, and obtains data such as queue length to send to the cluster manager.

- **Cluster manager.** This dispatches submitted jobs to the appropriate pool according to job type, collects demand statistics and implements the reallocation policy. The cluster manager obtains queue size information from each pool manager; this is used, together with the collected job data, in making reallocation decisions. The cluster manager acts as a coordinator between two pools during a server switch, and does not need to run an instance of the RMS.

Figure 1 illustrates the organization of these components.

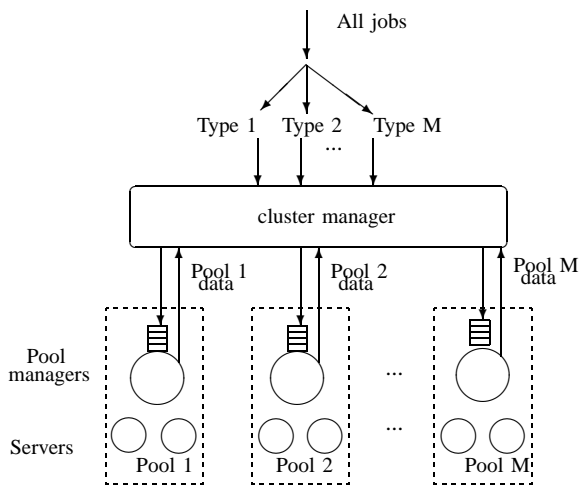


Fig. 1. Architecture of a cluster partitioned into pools

The different job types are registered with the cluster manager during initialization. Information about the resources required for each job type must be specified, so that each server in a pool is configured correctly. The cluster manager has a list of addresses which correspond to pool managers to which jobs are to be sent.

Dispatching a job to a pool does not require a significant amount of computation, so a queue is not likely to develop at the cluster manager. The pool manager communicates with the RMS through a proxy, allowing different RMSs to be used in different pools. Together the cluster manager and pool managers constantly monitor events in the system, including job arrival and processing times. That information allows the cluster manager to implement the desired server reallocation policy.

To switch a server from pool i to pool j , pool manager i is informed by the cluster manager of the decision to remove one of its servers, and it is also told that the destination is pool j . Pool manager i then chooses a server to be removed (e.g., an idle machine) and initiates its reconfiguration. The latter may include installing new programs downloaded from a software server, and also perhaps changing the RMS to match the one in pool j . If the target server is not idle, the jobs on it are returned to the queue. Upon successful reconfiguration of the

server, pool manager i sends its details to the cluster manager, which forwards them to pool manager j . Both pool managers make appropriate adjustments, such as updating their lists of current pool members and setting RMS parameters.

We have developed a prototype system to demonstrate the functionality of the server manager, pool manager and cluster manager components described above, and to show the operation of a heuristic switching policy. It is not intended to support a real hosting environment, and therefore does not use a complex job description language such as RSL or JDSL. Instead, it relies on Condor (see [10]), whose class-ads have been wrapped in an XML document containing file transfer information and job type. A simple file transfer mechanism was developed, and Condor is used as a resource manager. The switching that occurs involves only a server being taken from one pool and added to another; the ‘reconfiguration’ of the server consists of making adjustments to instances of Condor. A graphical interface allows a user to set system parameters and to observe changes in state. The components represent interfaces which allow different implementations to be plugged-in, so the simple file transfer protocol may be replaced by a system such as GridFTP.

During the initialization of the cluster manager, each pool is registered, and servers are added to it, using an administrator interface. The cluster manager then starts processing job submissions. The prototype cluster manager employs a round-robin polling pattern to obtain data from each pool. A call is made twice a second to each pool manager, which returns its current state (i.e., the size of its queue). The pool manager communicates with both the cluster manager and each server in its pool.

The Condor RMS was used in each pool to provide queueing and scheduling facilities, and to dispatch jobs to servers, together with any files they may need. Condor allows servers to switch from one pool to another simply by rewriting a configuration file. A job which is interrupted by a switch may have to re-start from the beginning when a server becomes available, or in some cases it may be check-pointed and continue from the point of interruption. Another reason for using Condor is that it is fairly straightforward to partition a cluster of servers running the Condor daemons into multiple pools by restarting some servers as pool managers.

The protocol invoked by the prototype when reallocating a server from a source pool to a destination pool is illustrated in figure 2. The reallocation decisions are controlled by a policy, whose development is described next.

III. OPTIMIZATION AND HEURISTIC

The optimization problem can be formulated as a Markov decision process, by making appropriate assumptions. Then, given that there is a stationary optimal policy, the latter can be computed numerically by truncating the state space to make it finite.

The following assumptions ensure the Markov property:

- Jobs of type i arrive according to an independent Poisson process with rate λ_i , and join a separate unbounded queue ($i = 1, 2, \dots, M$).

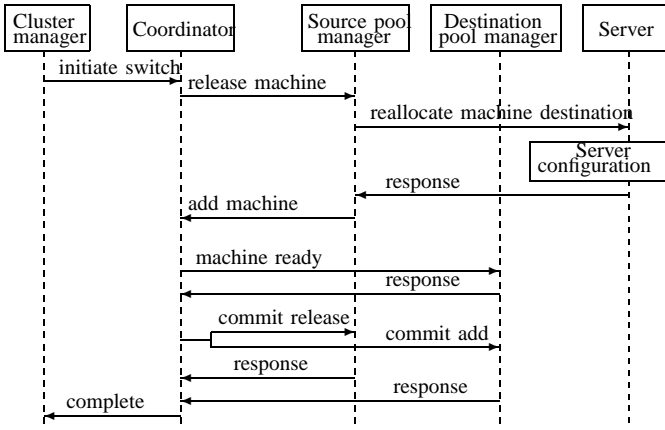


Fig. 2. A sequence diagram representing the switching protocol

- The required service times of type i are distributed exponentially with mean $1/\mu_i$.
- The time taken to switch a server from pool i to pool j is distributed exponentially with mean $1/\zeta_{i,j}$ ($i, j = 1, 2, \dots, M$).
- Switching decisions are made at job arrival or departure instants, and depend only on the current state.

For the purpose of this model, it is assumed that a service interrupted by a switch is resumed from the point of interruption when a server becomes available.

The primary QoS objective is to minimize appropriately weighted job response times. To that effect, it is assumed that a cost of c_i is incurred for each job of type i , per unit time that it spends in the system ($i = 1, 2, \dots, M$). These ‘holding’ costs reflect the relative importance, or willingness to wait, of the different job types. In addition to the holding costs, the switching of a server from pool i to pool j may incur a monetary cost of $c_{i,j}$.

The system state at any time is described by the triple, $S = (\mathbf{j}, \mathbf{k}, \mathbf{m})$ where $\mathbf{j} = (j_1, j_2, \dots, j_M)$ is the vector of current queue sizes (j_i is the number of jobs in queue i , including those being served), $\mathbf{k} = (k_1, k_2, \dots, k_M)$ is the vector of current server allocations (k_i servers allocated to queue i) and $\mathbf{m} = (m_{i,j})_{i,j=1}^M$ is the matrix of switches currently in progress ($m_{i,j}$ servers being switched from queue i to queue j , $m_{i,i} = 0$). The valid states satisfy $\sum_{i=1}^M k_i + \sum_{i,j=1}^M m_{i,j} = N$.

The instantaneous transition rates of the Markov process depend on the switching policy, i.e. on the decisions (actions) taken in various states. Denote by $r_d(S, S')$ the transition rate from state S to state S' ($S \neq S'$), given that action d is taken. The possible actions are (a) do nothing, or (b) initiate a switch from queue i to queue j (if $k_i > 0$ and $i \neq j$). These actions are numbered from $d = 0$ (do nothing) through $d = 1, 2, \dots, M(M-1)$. The values of $r_d(S, S')$ are easily expressed in terms of the parameters λ_i , μ_i and $\zeta_{i,j}$. The total transition rate out of state S , given that action d is taken, $r_d(S)$, is equal to:

$$r_d(S) = \sum_{S'} r_d(S, S').$$

The optimization may be carried out for a finite planning

horizon (minimize costs over a finite period of operation), or an infinite one (minimize total future discounted costs). In both cases, it is convenient to discretize the problem by applying the technique of uniformization to the Markov process (e.g., see [13]). This entails the introduction of ‘fictitious’ transitions which do not change the system state, so that the average interval between consecutive transitions ceases to depend on the state, and then embedding a discrete-time Markov chain at transition instants. First, we find a constant, Λ , such that $r_d(S) \leq \Lambda$ for all S and d . A suitable value for Λ is

$$\Lambda = \sum_{i=1}^M \lambda_i + N\mu + N\zeta, \quad (1)$$

where $\mu = \max(\mu_i)$ is the largest service rate and $\zeta = \max(\zeta_{i,j})$ is the largest switching rate.

Next, construct a Markov chain whose one-step transition probabilities when action d is taken, $q_d(S, S')$, are given by

$$q_d(S, S') = \begin{cases} r_d(S, S')/\Lambda & \text{if } S' \neq d(S) \\ 1 - r_d(S)/\Lambda & \text{if } S' = d(S) \end{cases},$$

where $d(S)$ is the state resulting from the immediate application of action d in state S . This Markov chain is, for all practical purposes, equivalent to the original Markov process.

Without loss of generality, the unit of time can be scaled so that the uniformization constant becomes $\Lambda = 1$.

The finite-horizon optimization problem can be formulated as follows. Denote by $V_n(S)$ the minimal expected total cost incurred during n consecutive steps of the Markov chain, given that the current system state is S . The cost incurred at step l in the future is discounted by a factor α^l ($l = 1, 2, \dots, n-1$; $0 \leq \alpha \leq 1$). Setting $\alpha = 0$ implies that all future costs are disregarded; only the current step is important. When $\alpha = 1$, the cost of a future step, no matter how distant, carries the same weight as the current one.

Any sequence of actions which achieves the minimal cost $V_n(S)$, constitutes an ‘optimal policy’ with respect to the initial state S , cost parameters, event horizon n , and discount factor α .

Suppose that the action taken in state S is d . This incurs an immediate cost of $c(d)$, equal to $c_{i,j}$ if the action taken is to switch a server from queue i to queue j . In addition, since the average interval between transitions is 1, each type i job in the system incurs a holding cost c_i . The next state will be S' , with probability $q_d(S, S')$, and the minimal cost of the subsequent $n-1$ steps will be $\alpha V_{n-1}(S')$. Hence, the quantities $V_n(S)$ satisfy the following recurrence relations:

$$V_n(S) = \sum_{i=1}^M j_i c_i + \min_d \left[c(d) + \alpha \sum_{S'} q_d(S, S') V_{n-1}(S') \right]. \quad (2)$$

Thus, starting with the initial values $V_0(S) = 0$ for all S , one can compute $V_n(S)$ in n iterations. In order to make the state space finite, the queue sizes are bounded at some level, $j_i < J$ ($i = 1, \dots, M$). Then, if $V_{n-1}(S)$ has already been computed for some n and for all S , the complexity of computing $V_n(S)$, for a particular state S , is roughly constant. There are no more than $2M + M(M-1)$ states S' reachable

from state S , and $M(M-1) + 1$ actions to be compared (corresponding to the $M(M-1)$ possible switches from queue i to queue j and action $d = 0$ to do nothing). The best action to take in that state, and for that n , is indicated by the value of d that achieves the minimum in the right-hand side of (2). Since there are on the order of $O(J^M N^{M-1+M(M-1)})$ states altogether, the computational complexity of one iteration is on the order of $O(J^M N^{M-1+M(M-1)})$, and hence the overall complexity of solving (2) and determining the optimal switching policy over a finite event horizon of size n , is on the order of $O(nJ^M N^{M-1+M(M-1)})$.

If the discount factor α is strictly less than 1, it is reasonable to consider the infinite-horizon optimization, i.e. the total minimal expected cost, $V(S)$, of all future steps, given that the current state is S . That cost is of course infinite when $\alpha = 1$, but it is finite when $\alpha < 1$. Indeed, in the latter case it is known (see [1]), that under certain rather weak conditions, $V_n(S) \rightarrow V(S)$ when $n \rightarrow \infty$. When the optimal actions depend only on the current state, S , and not on n , the policy is said to be ‘stationary’.

An argument similar to the one preceding (2) leads to the following equation for $V(S)$:

$$V(S) = \sum_{i=1}^M j_i c_i + \min_d \left[c(d) + \alpha \sum_{S'} q_d(S, S') V(S') \right]. \quad (3)$$

The optimal policy (i.e. the best action in any given state) is specified by the value of d that achieves the minimum in the right-hand side of (3).

Equation (3) can be solved by the ‘policy improvement’ algorithm (see Dreyfus and Law [4]). This iterative algorithm can be applied to the present optimization problem as follows.

Step 1. Start by making an initial guess about the optimal policy, i.e. construct an initial mapping, $d = f(S)$, from system states to action indices. This could be a simple heuristic such as the $c\mu$ -rule (see [2]).

Step 2. Treat this guess as the optimal stationary policy, and compute the corresponding discounted costs, V^f , by solving the large set of simultaneous linear equations:

$$V^f(S) = \sum_{i=1}^M j_i c_i + \left[c(f(S)) + \alpha \sum_{S'} q_{f(S)}(S, S') V^f(S') \right]. \quad (4)$$

Step 3. Now try to ‘improve’ policy f . For every state S , find the action $d^*(S)$ which achieves the minimum value in:

$$\sum_{i=1}^M j_i c_i + \min_d \left[c(d) + \alpha \sum_{S'} q_d(S, S') V^f(S') \right]. \quad (5)$$

In other words, minimize the total cost in state S , assuming that *after the current operation*, policy f will be used.

Step 4. If action $d^*(S) = f(S)$ for all states S , then the policy f cannot be improved; it is optimal. Otherwise, the next guess for the optimal policy is $f(S) = d^*(S)$; repeat from step 2.

The computational complexity of this algorithm is determined by the complexity of each iteration, which is dominated by step 2, and by the number of iterations. The simultaneous equations can be represented in matrix and vector form as:

$$V = C + \alpha QF(V) \quad (6)$$

where V is the matrix of unknowns, C is the vector of holding and switching costs, Q is the matrix of transition probabilities from state S to state S' , and $F(V)$ is an appropriate rearrangement of the elements of V .

An iterative method has been used to solve the set of simultaneous linear equations given in 6. Start with an initial approximation to V , such as the holding cost in the current state, $V_0(S) = \sum_{i=1}^M j_i c_i$, then at the n th iteration compute

$$V_n = C + \alpha QF(V_{n-1}) \quad (7)$$

Since Q is a stochastic matrix and $\alpha < 1$, this schema converges geometrically. This iterative solution is more efficient than Gaussian elimination for such a large state space, unless α is very close to 1.

A. Heuristic policy

The idea behind our dynamic heuristic policy is to attempt to balance the total holding costs of the different job types. That is, the policy tries to prevent the quantities $j_i c_i$ ($i = 1, \dots, M$) from diverging. The following rule is applied:

1. Evaluate the following quantity for each of the $M(M-1)$ possible switches from queue a to queue b ($a \neq b$ and $k_a > 0$):

$$c_b \left\{ j_b + \frac{1}{\zeta_{a,b}} [\lambda_b - \mu_b \min(k_b, j_b)] \right\} - K c_a \left\{ j_a + \frac{1}{\zeta_{a,b}} [\lambda_a - \mu_a \min(k_a - 1, j_a)] \right\},$$

where K is a constant used to discourage too many switches from being initiated. The best value of K depends on the total load. For heavily loaded systems, $K = 5$ has been used.

2. Find the maximum of all quantities calculated in 1; if it is strictly positive, this will be the most advantageous switch to initiate. Take the action $d \neq 0$ corresponding to this switch. Otherwise, take action $d = 0$.

This rule is based on approximating the effects of a switch. If j_b jobs of type b are present and k_b servers are available for them, then the average queue b increment during an interval of length x may be estimated as $x[\lambda_b - \mu_b \min(k_b, j_b)]$. Similarly for queue a , except that if a server is switched from queue a then the available servers for this queue drops to $k_a - 1$. Thus, a server is switched if that switch would help to balance the holding costs, after taking account of its effect on the M queues. The above policy will be referred to as the ‘heuristic’.

IV. EXPERIMENTAL RESULTS

We start by examining the optimal switching decisions for a modest model with 3 servers and 3 pools. In this example, the arrival and service parameters of the three job types are the same, but waiting times for type 1 are twice as expensive as those for types 2 and 3. The discount factor is $\alpha = 0.95$. The stationary optimal policy for states where $k_1 = k_2 = k_3 = 1$ and $j_1 = 0$ is shown in table 1. The truncation level used in the computation was $J = 15$, but the table stops at $j_2 = j_3 = 9$; the actions do not change beyond that level. Actions d are numbered as follows:

- d=0, do nothing;
- d=1, switch a server from queue 1 to queue 2;
- d=2, switch a server from queue 2 to queue 1;
- d=3, switch a server from queue 1 to queue 3;
- d=4, switch a server from queue 3 to queue 1;
- d=5, switch a server from queue 2 to queue 3;
- d=6, switch a server from queue 3 to queue 2.

	0	1	2	3	4	j_3 5	6	7	8	9
0	0	0	5	5	5	5	5	5	5	5
1	0	0	0	3	3	3	3	3	3	3
2	1	0	0	3	3	3	3	3	3	3
3	1	1	1	1	3	3	3	3	3	3
4	1	1	1	1	3	3	3	3	3	3
j_2 5	1	1	1	1	3	3	3	3	3	3
6	1	1	1	1	3	3	3	3	3	3
7	1	1	1	1	3	3	3	3	3	3
8	1	1	1	1	3	3	3	3	3	3
9	1	1	1	1	3	3	3	3	3	3

TABLE I

OPTIMAL ACTIONS: NON-ZERO SWITCHING TIMES, $N = 3$, $M = 3$,
 $j_1 = 0$, $\lambda_1 = \lambda_2 = \lambda_3 = 0.111$, $\mu_1 = \mu_2 = \mu_3 = 0.111$, $c_1 = 2$,
 $c_2 = c_3 = 1$, $\zeta_{i,j} = 0.0909 (i \neq j)$

We observe that switching is discouraged, compared to the $c\mu$ -rule. For example, when $(j_1, j_2, j_3) = (0, 2, 1)$ the optimal decision is to do nothing, even though a job of type 2 is not being served whilst a server at queue 1 remains idle. Only when $(j_1, j_2, j_3) = (0, 3, 1)$ is the decision made to switch a server from queue 1 to queue 2.

Next, compare the total average cost under three stationary policies: the optimal policy obtained by the policy improvement algorithm, the heuristic policy described in the previous section and a 'static' policy which allocates servers roughly in proportion to the average offered loads and changes the allocation only when those averages change.

The comparisons are carried out by simulation. The performance measure in all cases is the total average holding cost, i.e. the simulation estimate of $E(\sum_{i=1}^M c_i j_i)$. In each run, a total of 200000 job completions were simulated.

The optimal policy was pre-computed for each set of parameters and stored in the form of look-up tables. At every decision instant, the simulation program consults the table corresponding to the current state in order to find out what action to take. For that reason, the simulation of the optimal policy took the longest to run.

In all experiments, the parameters given below are renormalized to make the uniformization constant, Λ , equal to 1.

In figure 3, the average cost is plotted against the number of servers, N when $M = 3$. The following parameters are used: $1000\lambda_1 = \lambda_2 = \lambda_3$; $\mu_2 = \mu_3 = 1000\mu_1$; $c_1 = 2$, $c_2 = 1$, $c_3 = 1$. Switching rates are equal and given by $\zeta_{i,j} = \mu_2/10 = \mu_3/10$. Arrival rates are increased with N so that the total offered load, $\rho_1 + \rho_2 + \rho_3$, is equal to $4N/5$ (i.e., the system is quite heavily loaded). This models a system where type 1 jobs are much longer, and also more expensive to hold, than types 2 and 3. Requests of type 1 arrive at a much slower rate than for types 2 and 3, although the total load for each job type is the same.

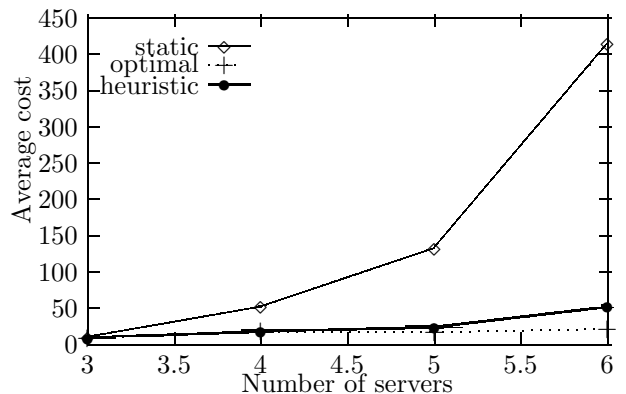


Fig. 3. Policy comparisons: $M = 3$ and increasing N

The figure has two features which should be pointed out. First, dynamic reallocation of servers definitely pays off. The static policy performs poorly compared to the dynamic ones, and the disparity becomes worse when N increases (for $N = 6$, the static policy is almost ten times worse than the heuristic). Second, the heuristic policy performs almost as well as the optimal policy. The difference between the two does appear to increase with N , but the rate of increase is low.

The next experiment compares the static, optimal and heuristic policies for a fixed number of servers ($N = 4$) and number of pools ($M = 3$), but varying offered load. The results are shown in figure 4. Here, all arrival rates are equal and increase together so that the total offered load varies between 2.6 and 3.6 (the system saturates and becomes unstable when that load reaches 4). The following parameters are used: $\mu_1 = \mu_2 = \mu_3 = 1$, $\zeta_{i,j} = 0.1$. Holding costs for type 1 are double those for types 2 and 3: $c_1 = 2$, $c_2 = 1$, $c_3 = 1$.

The features emphasized before are present again, and are even more pronounced. The static policy collapses and becomes unacceptable when the load exceeds 3.2, while the heuristic is almost indistinguishable from the optimal policy over the entire range.

We have also performed experiments where, in order to model changes in demand, the simulation runs include a sequence of phases, with λ_i changing values from one phase to the next. There are two possible values for each λ_i : a high rate and a low rate. In any one phase, a particular λ_i is set at the high rate, while the remaining λ_i are at the low rate.

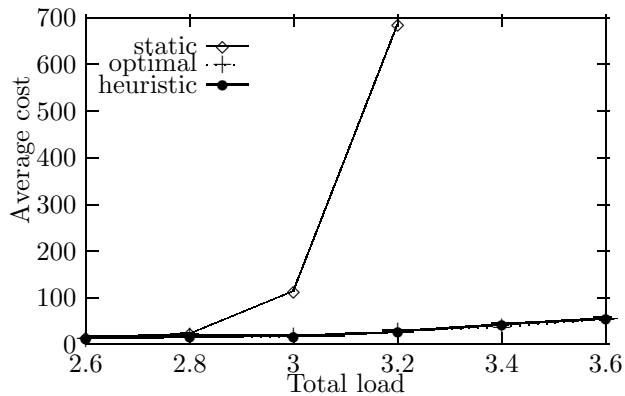


Fig. 4. Policy comparisons: $M = 3$ and increasing loads

This models demand peaking for a particular job type over a period of time. There were approximately 1000 phase changes in such simulation runs. Those results are not included here, because they exhibit the same general behaviour as figures 3 and 4.

V. CONCLUSION

A problem of interest in the area of distributed processing and dynamic Grid provision has been examined. The appropriate system architecture is discussed and an implemented prototype is described. The optimal reconfiguration policy can be computed and tabulated, subject to complexity constraints imposed by the size of the state space and the ranges of parameter values. However, for practical purposes, an easily implementable heuristic policy is available. The encouraging experimental results suggest that its performance compares quite favourably with that of the optimal policy.

Acknowledgement

This work was carried out as part of the collaborative project GridSHED (Grid Scheduling and Hosting Environment Development), funded by British Telecommunications plc and the North-East Regional e-Science centre.

REFERENCES

- [1] D. Blackwell, "Discounted dynamic programming", *Annals of Mathematical Statistics*, 26, pp 226-235, 1965
- [2] C. Buyukkoc, P. Varaiya and J. Walrand, "The $c\mu$ -rule revisited", *Advances in Applied Probability*, 17, pp 237-238, 1985
- [3] J.S. Chase, D.E. Irwin, L.E. Grit, J.D. Moore and S.E. Sprenkle, "Dynamic Virtual Clusters in a Grid Site Manager", 12th IEEE International Symposium on High Performance Distributed Computing, 2003
- [4] S.E. Dreyfus and A.M. Law, "The Art and Theory of Dynamic Programming", Academic Press, New York, 1977
- [5] I. Duenyas and M.P. Van Oyen, "Heuristic Scheduling of Parallel Heterogeneous Queues with Set-Ups", *Technical Report 92-60*, Department of Industrial and Operations Engineering, University of Michigan, 1992
- [6] I. Duenyas and M.P. Van Oyen, "Stochastic Scheduling of Parallel Queues with Set-Up Costs", *Queueing Systems Theory and Applications*, 19, pp 421-444, 1995
- [7] IBM on-demand business, <http://www.ibm.com/e-business>
- [8] G. Koole, "Assigning a Single Server to Inhomogeneous Queues with Switching Costs", *Theoretical Computer Science*, 182, pp 203-216, 1997
- [9] G. Koole, "Structural Results for the Control of Queueing Systems using Event-Based Dynamic Programming", *Queueing Systems Theory and Applications*, 30, pp 323-339, 1998
- [10] M. Litzkow, M. Livny and M. Mutka, "Condor - A Hunter of Idle Workstations", *Procs. 8th International Conference of Distributed Computing Systems*, 104-111, 1988.
- [11] Z. Liu, P. Nain, and D. Towsley, "On Optimal Polling Policies", *Queueing Systems Theory and Applications*, 11, pp 59-83, 1992
- [12] I. Mitrani and J. Palmer, "Optimal Server Allocation in Reconfigurable Clusters with Multiple Job Types", *Procs. International Conference of Computational Science and its Applications, Part II*, pp 76-86, 2004
- [13] E. de Souza e Silva and H.R. Gail, "The Uniformization Method in Performability Analysis", in *Performability Modelling* (eds B.R. Haverkort, R. Marie, G. Rubino and K. Trivedi), Wiley, 2001
- [14] Sychron <http://www.sychron.com/>