Fast Scalable Optimization to Configure Service Systems having Cost and Quality of Service Constraints

Jim (Zhanwen) Li, John Chinneck, Murray Woodside
Dept. of Systems and Computer Engineering
Carleton University, Ottawa Canada
{zwli | chinneck | cmw}@sce.carleton.ca

Marin Litoiu
School of Information Technology,
York University, Toronto, Canada
mlitoiu@yorku.ca

ABSTRACT
Large complex service centers must provide many services to many users with separate service contracts, while managing their overall costs. A scalable hybrid optimization procedure is described for a minimum-cost deployment of services on nodes, taking into account processing requirements and resource contention. This is a heuristic for a problem which is in general NP-hard. It iterates between a fast linear programming (LP) sub-problem, and a nonlinear performance model, both of which scale easily to thousands of services. The approach can be adapted to minimize cost subject to performance constraints, or to optimize a combined quality of service measure subject to cost constraints. It can be combined with tracked performance models to periodically re-optimize deployment for autonomic QOS management.

Keywords
Allocation, optimal deployment, autonomic control, cloud computing, performance, service systems, performance management.

1. INTRODUCTION
Service systems including web applications, legacy client-server applications, platforms (i.e. PAAS [5]), infrastructure (i.e. IAAS [19]), and information services are increasingly hosted in large processing complexes sometimes called *clouds* [11][21]. These give the advantages of flexible deployment as needs change, hide management details from the user and the service provider, and require payment only for resources used. Clouds use virtualization to achieve controlled sharing of resources, rapid redeployment of application images, and isolation of different applications and instances from each other (when they share a host).

The economics of clouds require efficient sharing of the resources between large numbers of applications, beginning with efficient deployment of applications on the hosts of the cloud. We seek deployments which minimize the overall cost of the hosts used, subject to meeting average delay and throughput constraints for each application as posed by its service level agreement (SLA). A deployment method must be able to scale up to thousands of services running on thousands of hosts, and should be cheap enough to re-run frequently as loads and requirements change. A performance model is essential to account for software contention (e.g. thread or buffer-related delays) and its effect on delay. The solution proposed here combines a rapid linear optimization of execution flows, with a scalable approximate layered performance model. The present work will require further extension to address memory requirements of deployments, which are also important.

ICAC-09, June 15-19, 2009, Barcelona, Spain.

Figure 1 illustrates the deployment of application processes in our experimental cloud for CERAS [4]. Virtualization of processors makes it possible for separate applications with separate virtual machines (VMs) to safely share a physical node, and a virtual machine monitor can control the rate of processing provided to each VM. Deployment issues include (i) the number of replicas of each service, (ii) the selection of processors, (iii) computing power consolidation, (iv) allocation of service replicas, and (v) workload balancing and distribution. The system should meet performance targets described in service contracts, (e.g. response time, number of users, capacity given as arrival rates), and economic targets (e.g. cost budgets, power constraints, profit targets).

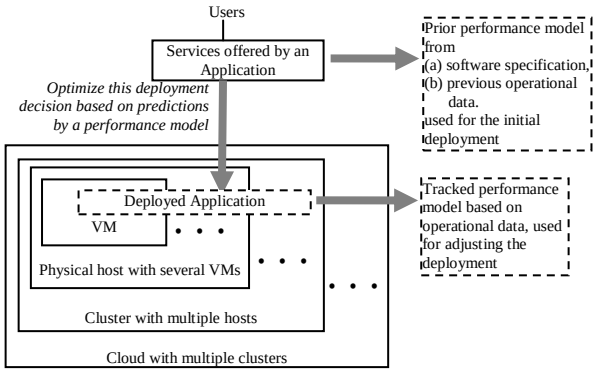


Figure 1. Application Processes in a Cloud

The deployment is based on the existence of a performance model of each application we deploy. This model can be built from software specification for the initial deployment and from runtime performance tracking for periodical redeployments. Figure 2 shows two deployment scenarios: (a) *incremental deployment* in red, with a new application deployed in a busy cloud (b) *full optimal redeployment* in blue where all applications are optimally redeployed to adapt to changing conditions. The deployment/

optimization module computes the deployment plans and forwards them to a deployment engine (such as IBM Tivoli Provisioning Manager) which executes them. The optimization decisions are based on the *state* of the cloud which includes information about the applications and resources already allocated and also, in scenario (b), a tracked performance model for each application in the cloud.

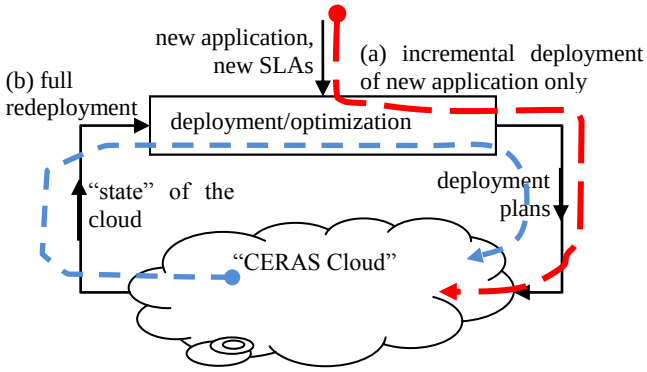


Figure 2. Two Deployment Scenarios

1.1 The Model

We view a service system as comprising UserClasses, Services, ServerTasks, Resources and Hosts, related as sketched in [23] and illustrated through a UML class diagram in Figure 3. UserClasses request services from outside, and these services request other services inside or outside the system (exploiting the concepts of Service-Oriented Architecture), forming a web of inter-service traffic. Services are implemented by Applications which run as system tasks or thread pools (ServerTasks), which may have limited capacity. UserClasses have throughput and delay requirements expressed by their SLAs. Hosts have flow constraints due to limited capacity. Additional Resources such as memory are important but will not enter the present analysis.

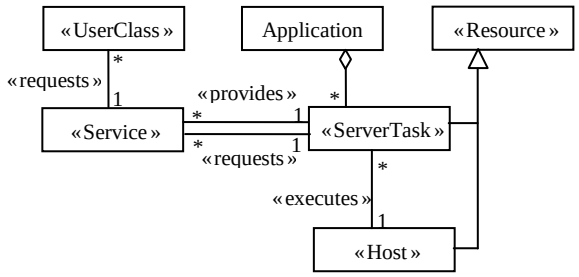


Figure 3. Service System Concepts

The problem addressed here is to determine the deployment and sharing of hosts which minimizes the cost of processing, subject to *mean throughput constraints*, and *taking into account resource contention*. Response time constraints can be rewritten as equivalent throughput constraints, based on finite user populations, as described below.

A finite population (closed workload) is preferable for the performance calculations because the results are never unstable due to overloaded hosts, which can happen when throughput is fixed (open workload). Suppose UserClass c has a fixed population of N_c users (called a *closed* workload situation), and has throughput f_c and mean response time RT_c, then Little's identity states that

$$f_c = N_c / (RT_c + Z_c) \quad (1)$$

where Z_c is the average time the user spends between receiving one response and making the next request (sometimes called a think time). If N_c is specified in the SLA, and a target response time $RT_{c,SLA}$ is given, then the target throughput is given by:

$$f_{c,SLA} = N_c /(RT_{c,SLA} + Z_c) \quad (2)$$

When Z_c is unknown the worst-case value of 0 can be taken. If both $f_{c,SLA}$ and $RT_{c,SLA}$ are specified, a throughput is computed from the latter using Eq. (2) and the larger throughput is used.

If on the other hand the throughput is assumed fixed (called an *open* workload situation) and the SLA specifies response time, then arbitrary large values of N_c and Z_c are chosen to approximate the open situation by a closed one. The target $f_{c,SLA}$ is computed using (2), and the chosen values of N_c and Z_c are used in the performance model.

In summary, the approach taken here finds the minimum cost deployment subject to processing capacity and user throughput constraints. We use a network flow model to find a deployment and then apply the deployment to a closed performance model and iteratively adjust the flow model for the contention delays. As will be seen, both parts of the iteration are fast and scale well.

1.2 Related Research

Research on deployment optimization *without* considering resource contention includes graph partitioning to minimize communications [1][2], bin packing for time or memory [6][22], and hill-climbing [12]. Recently attention has shifted to time-varying situations with periodic re-deployment. Karve [14] and Steinder et al. [17] used heuristics to distribute workloads across virtual nodes in IBM Websphere XD, and Tang et al. [20] presented a combination of max-flow algorithms and heuristics to allocate varying workloads across large scale systems.

Contention may be included in the optimization, using a performance model to calculate the queueing delays, and giving a difficult non-linear integer programming problem. Heuristics are common. For example, Beaty et al. [1] proposed a heuristic approach to optimizing server migration and consolidation in terms of performance level. Menasce et al. [15][16] describe heuristics for good combinations of QoS-aware components or service information providers across networks. Their work is limited by using queueing models which ignore contention for software resources. Limited thread pool and buffer pool sizes are examples of software resources which can be bottlenecks [8].

In [28] a performance model was used to track system changes, and the deployment (and some other system parameters) were optimized periodically using a simple hill-climbing algorithm. However the algorithm does not scale to very large systems, which prompted the present effort to find a more scalable optimization algorithm to use with the tracking technique.

2. PERFORMANCE PREDICTION FOR SERVICE SYSTEMS IN CLOUDS

Resource contention increases delays. If software resources are ignored, then all waiting occurs at processors, and where throughputs are fixed (open workloads) contention may in principle be controlled by limiting processor utilizations to some chosen amount such as 80%. However this does not provide an estimate of delay so one cannot address the SLA for delay using this solution. This is why, in e.g. [15][16], a performance model for the processors is introduced. However, there is increasing evidence that software resources are also important, and for this a more structured performance model is required. This work uses a layered queueing network (LQN) model [9][24][18] because it models important software contention effects.

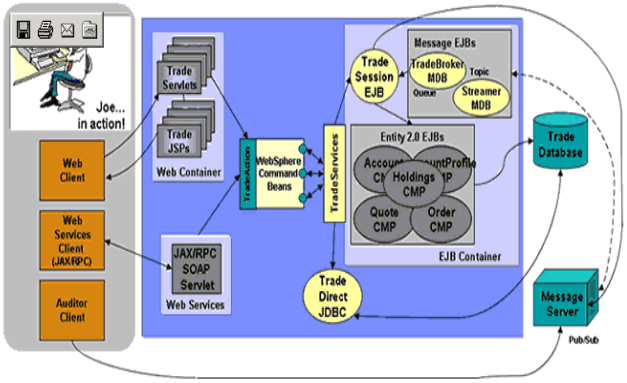
2.1 LQNs and the Model of a Service Center

A *Layered Queueing Network* (LQN) model of a service system is a simplified view of its structure, emphasizing its use of resources. This is illustrated by a small system shown in Figure 4. The users (UserClasses) are represented in the LQN by *userTasks*, in which userTask c has population N_c. A userTask does not receive any requests, but rather cycles forever, waiting for a think time Z_c given as their demand (e.g. [1000 ms]), and then making a set of requests for service shown by directed arcs to the services. The arcs or arrows are labeled with mean counts of requests, per operation of the requester, e.g. (1). Services are represented by *entries*, which have processing demands D and make requests to other entries. Where a Service is provided by a ServerTask, the entry forms part of a corresponding resource called a *task*, and is deployed on a *processor*. Tasks and processors have a multiplicity $\{m\}$ (e.g. {50}, modeling multiple threads or multiprocessor). As discussed in [9], other software resources such as buffer pools may also be modeled as *tasks*.

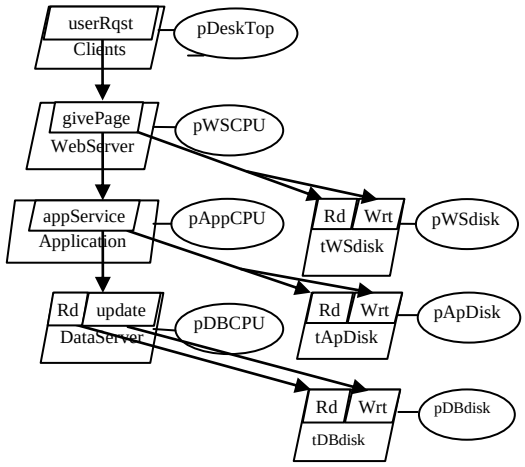
LQN models are special extended queueing networks [9][18][24] which incorporate services with nested requests for other services, blocking of tasks making synchronous requests, other request types (asynchronous, forwarding, parallel), and multiphase types of software service with synchronous (in-rendezvous) and asynchronous (post-rendezvous) phases. They have been successfully applied to many applications (see, e.g. [9] for references).

An LQN solution determines throughputs at users, entries, tasks and processors, delays including queueing for requests, and resource utilizations. Number the userTasks as $c = 1...C$; entries as $s = 1...S$, tasks as $t = 1...T$, and host processors as $h = 1,...H$. Then throughputs at these entities are f_c, $f_{ENTRY,s}$, $f_{TASK,t}$, and $f_{HOST,h}$ respectively. Task and processor utilizations are $u_{TASK,t}$ and $u_{HOST,h}$ respectively, and for a *multiple* resource, full utilization makes the utilization equal to the multiplicity m.

The more complex LQN in Figure 6 indicates the potential of the LQN framework, with the main features of a shopping service application. The two topmost user tasks represent the two classes of users, with 250 and 100 users. The pUsers processor represents the user desktops. Arrows represent requests for services (labeled by the mean number of calls, e.g. (2)), with a filled arrowhead indicating a synchronous request (the requester waits for a reply), and an open arrowhead, an asynchronous request. There are databases for inventory and customer information. Entries are named beginning with "e" in Figure 6 and carry labels (e.g. [1]) for the mean execution demand D on the host. Processors are shown as ovals, linked to tasks deployed on them; a processor entity may represent a multiprocessor. Processors and tasks are labeled by a resource multiplicity (e.g. {100}). For a user task the multiplicity is the number of users in the class. Pure delays without contention are represented by infinite-multiplicity tasks and processors. Some additional details: a device like a disk is modeled by a task with entries to describe its services, and a processor representing the physical resource. Delay for an external service not modeled in detail can be represented by a surrogate task with a pure delay (infinite task) and entries for its services, as for the Payment Server and Shipping Server.



(a) The service system



(b) The LQN performance model

Figure 4. A Lab-Scale System with the Trade 6 Benchmark

We will assume that a performance model has already been constructed. In practice it is derived from ordinary measurements on the running system. The structure of tasks and entries participating in each service is found either from the system design or by tracing some representative requests as described in [29]. The parameters are determined by profiling, by regression techniques [27] or by using a tracking filter [26][28]. In practice these models are not perfect, because of statistical estimation errors, and delays in computing the parameters (during which the system may change). The references above discuss how this inaccuracy may itself be estimated and controlled.

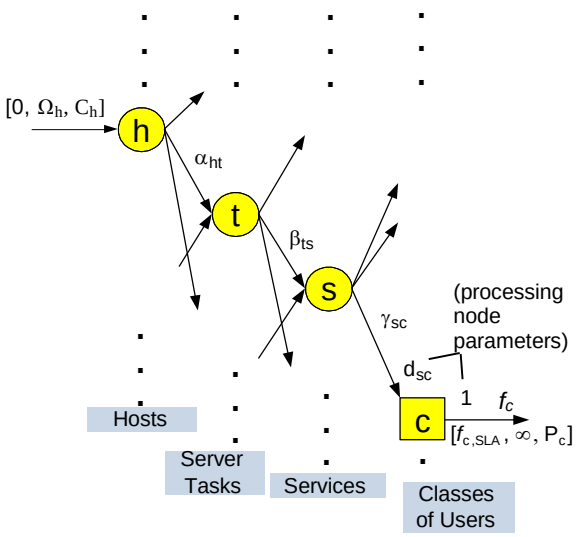


Figure 5. A Network Flow Model

3. OPTIMAL DEPLOYMENT BY A NETWORK FLOW MODEL

The deployment problem is often formalized in terms of assignment variables a_{ht} such that $a_{ht} = 1$ if task t is deployed on host h, or 0 otherwise. Here we consider instead the flow of execution of services of task t by host h, as part of the solution of a network flow model (NFM). A NFM is a graph with arcs which carry flows and nodes which operate on the flows, as illustrated in Figure 5. Each node in Figure 5 is representative of a set of nodes, with H nodes in the Host column, T nodes in the Task column, S nodes in the Services column, and C nodes in the Class column.

The unknown flows α, β, γ comprise the variables in the model. Each arc is labeled with a triple of parameters $[l,u,c]$: the lower flow bound l (default 0), the upper flow bound u (default infinity), and the cost per unit of flow c (default 0). The parameters are not shown where all take the default values. Ordinary nodes are of three types, and are shown as circles in a network diagram. *Source nodes* introduce flow into the network and *sink nodes* remove flow from the network, at rates given by the input and output arcs attached to them, called *phantom arcs*. Ordinary nodes simply balance flow between their input and output arcs (total input = total output) [10]. In addition, a special type of NFM called a *processing network* [10] has at least one *processing node* which has fixed ratios of the flows in its incident arcs. Processing nodes are shown as squares labeled with the fixed proportion of flow at the attachment point of each incident arc. The resulting model

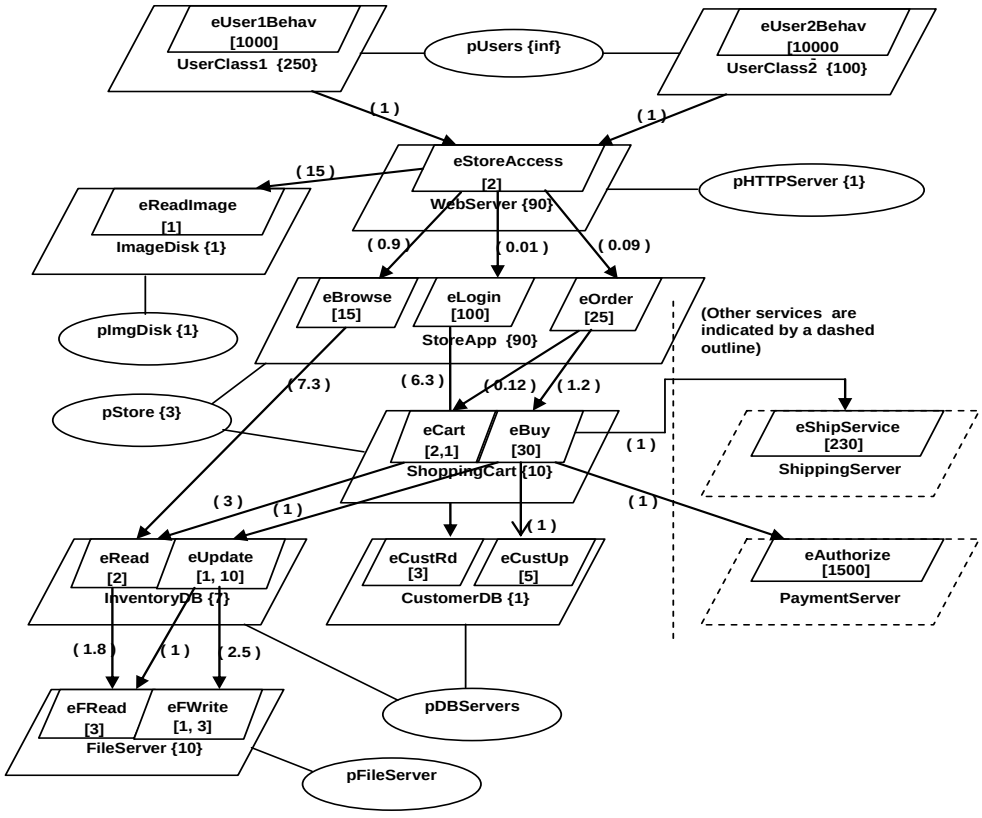


Figure 6. An Example of a Layered Queuing Network Model of a Service System

consists entirely of linear relationships, and with a linear objective function it forms a linear programming optimization problem.

An NFM is derived from the LQN performance model by considering the flow of demands for CPU work implied by the request arcs in the LQN. An NFM host node h =1...H is created for LQN processor h; a task node t=1...T is created for non-user task t; a service node s=1...S is created for entry s. These are *ordinary nodes* which relate demand flow on each host to demand flows by services. The NFM may include additional processors which are not used but which could be used in an optimal deployment. For each *userTask c* there is a *processing node* for user class c in the NFM, which converts a flow of user requests into CPU demands by services. Table 1 summarizes the entities defined for service systems in general, with their corresponding representations in the LQN and NFM models.

Table 1 Corresponding Entities in Different Views

Service System	Network Flow Model (NFM)	Layered Queueing Network (LQN)
Processor h	Host h	Processor h
UserClass c	User class c	UserTask c
Service s	Service node s	Entry s
ServerTask t	Task node t	Task t
Resource	...	A Task or Processor
Activity	...	Activity (within an entry)

The arcs show which flows between nodes may be non-zero, and by the conventions of modeling with the NFM they flow into the hosts, and out at the user classes. Each arc has a flow quantity, defined as

flow quantity = demands for CPU-sec of processing, transferred per second between nodes

and initially the CPU-sec for any operation are assumed to be the same on all hosts (hosts are of uniform speed).

The input arcs on the left in Figure 5 represent the total flow $f_{HOST,h}$ at host h, and are labelled by $[0, \Omega_h, C_h]$ meaning that flow ≥ 0, the host capacity limit is flow $\leq \Omega_h$, and the cost is C_h per unit of flow. For a set of processors of equal speed, and flows given in CPU-sec/sec, the capacities are all 1.0. There is also an arc:

- from host h to each task t which is permitted to be deployed on h, with flow α_{ht} (the demand rate executed on host h, to satisfy the needs of task t). If multiple replicas of a task are deployed, it will have non-zero flows from multiple processors, which will optimally divide the execution flow between them.
- from task t to each service s offered by task t, with flow β_{ts} (the demand rate from the service). In the LQN each service (entry) is associated with just one task.
- from service s and each user class c which causes s to be executed, with flow γ_{ts}. γ_{ts} is the total CPU demand triggered at service s by requests made by class c.

These arcs relate demands at processors to demands from user requests, and express the software structure and the constraints on deployment of tasks. Omitted arc labels default to $[0, \infty, 0]$.

The output arcs at the right have a flow which is the requested throughput of the user classes. The user class node c is a processing node with flow ratio parameters which convert the class flow f_c at the right in Figure 5, in units of user requests/sec, to demand flows γ_{sc} for services. For each single user request by class c, a demand of d_{sc} CPU-sec is required for service s, giving this flow proportionality:

$$\gamma_{sc} = d_{sc} f_c$$

The value of d_{sc} can be determined by profiling the system for each user class request type, or from the LQN model. In the LQN, let Y_{cs} be the total direct and indirect mean requests to entry s for one request from user class c, and let y_{es} be the mean requests made directly from any entry e to entry s. For this purpose user class c will be defined to have an entry numbered $S + c$, and $y_{S+c,\,s}$ is the mean number of requests made directly to entry s for one user response (in 0, there is exactly one request to a particular service entry point, but it can be more general). Then assuming there are no request cycles, Y_{cs} can be computed by setting $Y_{c,S+c} = 1$ for all c, and using:

$$Y_{cs} = \sum_{e=1}^{S+C} Y_{ce} y_{es} \quad , \quad s = 1..S$$

Using the parameter D_s from the LQN, for the CPU demand per execution of entry s, we obtain $d_{sc}=Y_{cs}D_s$.

We can now state the decision problem. It is a linear program (LP) to find the flows which minimize total cost subject to flow constraints:

$$\min_{\alpha_{ht},\beta_{ts},\gamma_{sc}} COST = \sum_{h=1}^{H} C_h \sum_{t=1}^{T} \alpha_{ht} \tag{3}$$

subject to:

- Service level agreement: for each c, $f_c \geq f_{c,SLA}$.
- Host capacity: for each h, $\sum_{t \in T} \alpha_{ht} \leq \Omega_h$. To limit the maximum processor utilization to $\varphi_h < 1$ replace Ω_h by $\varphi_h \Omega_h$.
- Flow balance: $\sum_{h \in H} \alpha_{ht} = \sum_{s \in S} \beta_{ts}$ (for all t); $\sum_{t \in T} \beta_{ts} = \sum_{c \in C} \gamma_{sc}$ (for all s); $\gamma_{sc} = f_c d_{sc}$ (for all s and c)
- Nonnegative flows: for all h, t, s, $\alpha_{ht} \geq 0$, $\beta_{ts} \geq 0$, $\gamma_{sc} \geq 0$.

As pointed out in the previous section, for a suitable closed workload population the satisfaction of the user throughput requirement implies satisfaction of the response time requirement.

The solution of the NFM gives the optimal flow rate in each arc, which shows how processing demands should be distributed from hosts to services. The allocation of demands includes computing power consolidations and isolations, the number of replicas of each task or service and the allocation of these services onto the virtualized nodes as well as the transaction flow rates etc.

The generalization to a set of processors of different speeds is trivially made through the host capacities. In place of Ω_h = host multiplicity, we have Ω_h = host multiplicity × speed factor of each element. The speed factor is relative to the type of processor for which the CPU demands are defined. If processor types are such

that simple speed scaling is not possible, then the linearity of the problem is lost, and the NFM cannot be applied.

In summary, the parameters are:

- Ω_h: the maximum operation demand rate available at host *h*, seen as the capacity of host *h*.
- φ_h: maximum host usage fraction of host h (safety factor),
- C_h: the cost per unit of operation demand of host *h*,
- P_c: profit per transaction of class *c*,
- d_{sc}: the amount of operation demand rate needed from service *s* by a request in class *c*,
- $f_{c,SLA}$: the required throughput for class *c* in the service level agreement,
- $RT_{c,SLA}$: the required response time of class *c* in the service level agreement.

The variables are:

- α_{ht}, the operation demand rate from host *h* assigned to task *t*,
- β_{ts}, the operation demand rate from task *t* assigned to service *s*,
- γ_{sc} the operation demand rate in service *s* assigned to user class *c*, and
- f_c, throughput of class *c*, $f_c = \gamma_{sc} / d_{sc}$.

The solution of this NFM ignores contention for resources in computing the throughputs, and this makes it optimistic. Contention delays reduce the flow rates and increase the response times. Contention delays are estimated by an LQN which represents the first NFM solution, and the NFM is then adjusted to account for those delays. These steps are the core of this paper, and are described below.

Queueing for processors is only part of the contention. Software server interactions often introduce blocking delays either for limited server threads or for critical sections, which reduce the servers' capacity in *software bottlenecks* [8]. The LQN solution includes the blocking delays at the requesting entry. These delays are converted into equivalent demand rates for processing by that service, to create a new NFM, which is solved in an iterative loop as shown in Figure 7.

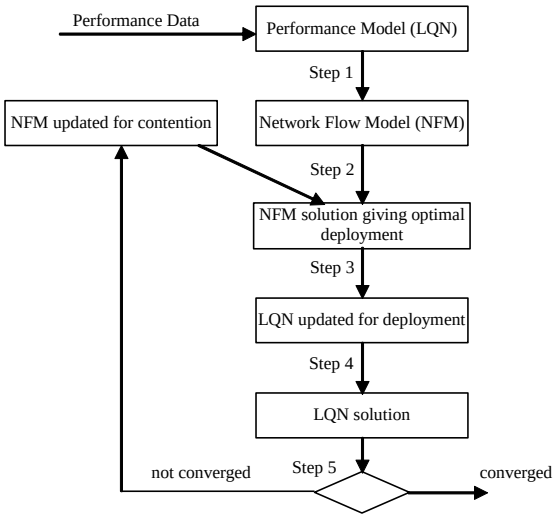


Figure 7. Optimization Loop

The overall optimization loop is made up of 5 main steps and iterates until the optimal configuration to achieve the required QoS is found. The four steps are:

Step 1. construct the NFM as described in Section 3,
Step 2. solve the NFM optimization,
Step 3. reconfigure the performance model to incorporate the deployment decisions in the NFM solution, as described below,
Step 4. solve the performance model
Step 5. test for convergence of the user throughputs. If not converged, incorporate the queueing delays into the NFM as described below, and repeat from (2).

3.1 Steps 1 and 2: Set up and Solve the NFM

The initial LQN and NFM construction were described above. The NFM is solved by any LP package. For example, Figure 8 shows a fragment of the NFM for the LQN in Figure 6. pStore has 5 processors and a cost factor of 3; UserClass1 has the required throughput as shown, and a cost figure of P_1/response which is not used here. Arcs from task ShoppingCart to services eCart and eBuy show that it includes those services.

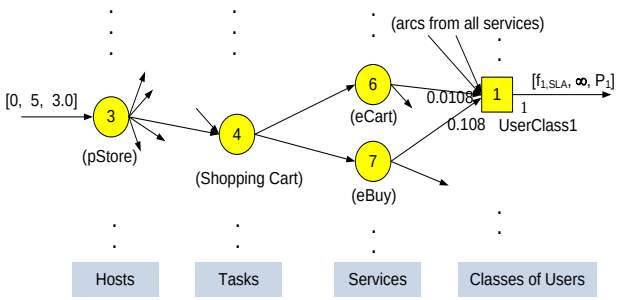


Figure 8. Fragment of NFM for the System in Figure 6

The network flow model gives a corresponding LP for this assignment problem, as described in Section 3. Solving the LP gives the optimal flow rates carried by every arc to reach the optimal objective. The host-to-task arcs with non-zero flow indicate the deployment of the tasks.

3.2 Steps 3 and 4: Insert Deployments and Solve the LQN

The optimal task-to-processor flows in the NFM determine the task deployments in the LQN. Where a task *t* has NFM flow from a single host, this means it is deployed only on that host. However if it has non-zero flow from several hosts then task *t* is replaced by a set of identical replica copies (with the same set of entries), with the replica deployed on host *h* identified as task *t_h* and its replica of entry *s* identified as entry *s_h*. Each request to an entry of task *t* is split among the replicas in the same ratios as the NFM flows α_{ht}. To do this, each request arc to an entry of task *t* (say an arc from entry *e*) is replaced by a set of request arcs. The arc from entry *e* to entry *s*, labeled with y_{es} requests, gives an arc to entry *s_h* in task replica *t_h* labeled with y_{e,s_h} requests, with

$$y_{e,s\_h} = y_{es} (\alpha_{ht} / \Sigma_h \alpha_{ht})$$

and this is repeated for each replica of task *t*. The solution is found using the LQNS solver [7][9].

The NFM only partially represents the use of hosts in the LQN. The NFM allocates a fraction of a host to processing a task, and that fraction enters the cost in Eq. (3). In the LQN however the full processing power of the host is available, and the cost is really the full cost of the processor. Thus the cost in Eq. (3) may be an underestimate. This is why, in Section 5.1, we will see constrained (an thus sub-optimal) solutions with a lower cost value than unconstrained solutions.

3.3 Step 5: Convergence Test or Iteration with NFM Adjustment

The NFM approximation is improved by adjusting for the effect of contention as estimated by the performance model (the LQN), and iterating the NFM solution.

At iteration *i*, let the throughput of user class *c* be $f^{i}_{c,LQN}$, and define the shortfall in throughput to be e^{i}_{c}:

$$e^{i}_{c} = f_{c,SLA} - f^{i}_{c,LQN}$$

When e^{i}_{c} is less than the allowanced tolerance rate such as 1% of the $f_{c,SLA}$, it means that the optimal configuration for class *c* has been found, which can converge the throughput and response time to the target value. Iteration stops when every class has converged.

Otherwise a new NFM is created, denoted NFM^{i+1} for the next iteration *i+1*, adjusted to deal with the shortfall. The shortfall in user throughput is attributed proportionately to the demands for services, with an amount

$$e^{i}_{sc} = d_{sc}\, e^{i}_{c}$$

for service *s*. Over all users, the flow adjustment to service *s* is

$$\delta^{i}_{s} = \Sigma_{c}\, e^{i}_{sc} = \Sigma_{c}\, d_{sc}\, e^{i}_{c}.$$

The heuristic used to adapt the NFM is to increase the host flows to provide enough additional processing capacity to make up this shortfall. δ^{i}_{s} is added to the flow through service node *s*. The additional flow is not real processing, but is capacity to process which is provided by the deployment to reduce contention; we may term it a *virtual demand rate*. The virtual demand rate at service *s* is represented in NFM^{i+1} by an output arc with fixed rate Δ^{i+1}_{s}. When NFM^{i+1} is solved, the total demand rate at the hosts will be increased by this amount.

The increment in virtual demand at iteration *i* is δ^{i}_{s}, and the total rate on the output arc at service *s* is the sum over the iterations:

$$\Delta^{i+1}_{s} = \Sigma^{i}_{j=1}\, \delta^{i}_{s}$$

This is indicated by an output arc from service node *s* with the label $[\Delta^{i+1}_{s}, \Delta^{i+1}_{s}, 0]$.

In the new NFM, the replicas of a task (if any) are treated again as a single task node. The iteration continues until enough resources are reserved to compensate for the performance lost due to contention, and the requirement is met (or until the iteration limit).

4. EXAMPLE AND EVALUATION

The decision algorithm was evaluated for convergence, scalability, degree of complexity and accuracy, using a case study of a moderate-sized service system. It is represented by the LQN in Figure 9, showing two classes of users. Class 1 has 250 users and class 2 has 100 users. The objective is to minimize the host computing costs while meeting the multiclass workload response time goals. Figure 9 shows the deployment of a single application.

The response time goal $RT_{c,SLA}$ is chosen to be the value that can be provided with infinite resources in the system. The solution of the LQN with an infinite processor for each task makes $RT_{1,SLA}$ = 0.146 sec and $RT_{2,SLA}$ = 0.267. The corresponding value of *fc* is taken as the goal for the NFM optimization, giving $f_{1,SLA}$ = 250/(1 + 0.146) = 219.3/sec, and $f_{2,SLA}$ = 100/(1 + 0.267) = 78.9/sec.

The NFM optimization will determine what power is needed to provide the best possible service, on a certain set of hosts.

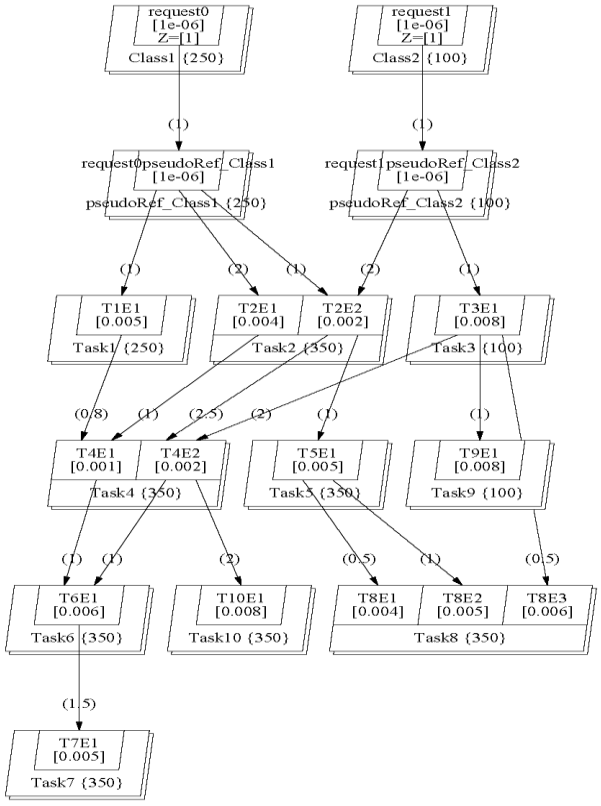


Figure 9. LQN Model of a Service Center

Table 2. Host Resource Attributes in the Example

Host *h*	m_h	φ_h	Speed Ratio	Ω_h	Cost C_h	Hostable Tasks
Host 1	20	80%	1	20	1	1,2,4,7
Host 2	20	80%	1.2	24	1.1	3,4,6,7
Host 3	20	80%	0.9	18	0.9	1,4,5,6
Host 4	20	80%	1.1	22	1.1	3,7,9,10
Host 5	20	80%	0.8	16	0.7	1,2,8,10
Host 6	20	80%	1.2	24	1.2	5,6,8,9

There are six hosts available with constraints as to the tasks that can be deployed to them. Demands are defined in CPU-seconds on a reference processor type, with a relative speed factor for each host. The resources at each node are described in Table 2. The column headed m_h gives the multiplicity of each host.

A fragment of the network flow model is shown in Figure 10.

The optimization did not reach its goal, but it came within 5% in five iterations. In fact the goal is not feasible, and if the iterations are continued the number of processors used will creep up until all are used, without ever quite reaching the SLA goal. The thread pool size of each task in the LQN was set to handle 70% of the possible maximum demand rate at the task, a value found by experience to give good results.

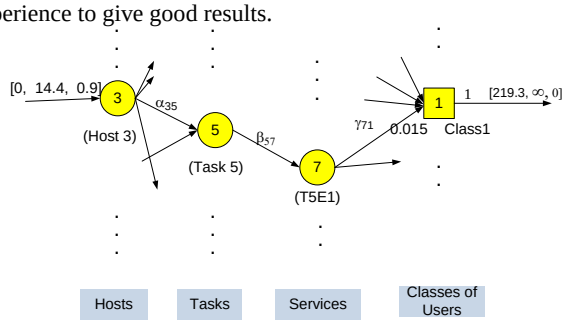


Figure 10. Fragment of Network Flow Model

The performance of each class, the resource utilizations and the service allocation can be seen in Table 3 and Figure 11. Table 3 shows the performance achieved by users in every class in each round. Table 4 shows the computing power consolidation in every node, in which the integer number indicates the required multiplicities of devices. Notice that a multiprocessor is fully utilized when its utilization equals its multiplicity.

Table 3. Response Times of Classes in Each Iteration

Class	Itn 1	Itn 2	Itn 3	Itn 4	Itn 5	Final	Goal	Errors
Class1	0.290	0.168	0.162	0.153	0.149	0.147	0.146	+0.68%
Class2	0.456	0.368	0.311	0.290	0.280	0.272	0.267	+1.87%

Table 4. Host Multiplicity at each Iteration, and Final Utilizations

Host	Itn 1	Itn 2	Itn 3	Itn 4	Itn 5	Final	Final Utilization
1	8	5	7	8	5	5	5 × 0.72
2	17	17	17	16	16	16	16 × 0.66
3	0	0	0	2	11	8	8 × 0.80
4	8	16	16	16	17	17	17 × 0.68
5	16	16	16	16	16	18	18 × 0.79
6	5	5	5	6	11	8	8 × 0.70

Figure 11 shows that tasks 6, 7, and 10 are replicated across multiple hosts, and that every host accommodates at least two tasks. The ratio of request flows divided between replicas has been determined by the relative flows, as described above. These results were obtained by a prototype tool which combines a handmade LP solver with either LQNS or Apera for the LQN solution. The tool has been used in a prototype decision maker for the CERAS cloud.

The calculations above had an infeasible goal, that the response should equal the no-contention lower bound, and the search continued for the maximum iterations. Feasible goals can be provided by increasing the response time requirement. Factors of 1.1, 1.2,...,1.5 were applied to the required values of 0.146 sec for Class 1 and 0.267 sec for Class 2, to give the results in Table 5. The larger response time, the easier the problem. We can see that as the factor increases, the cost of the system required to meet the requirements decreases and the solution is found more quickly.

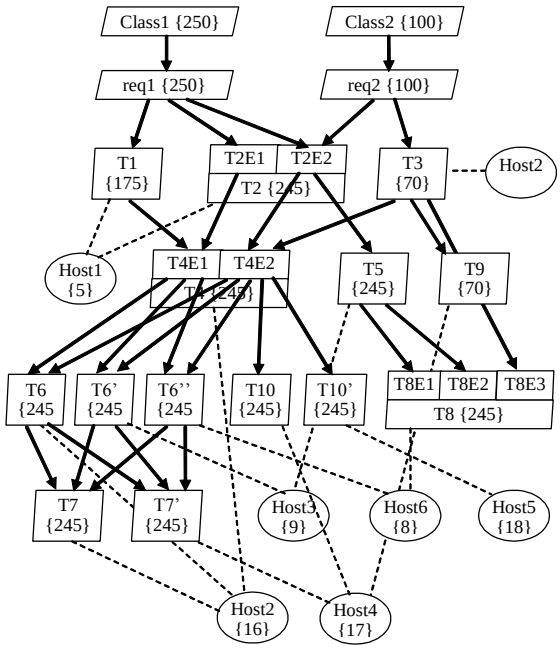


Figure 11. Optimal Deployment for the Service Center in Figure 9

Table 5. Cost and Number of Iterations Corresponding to the Relaxation of the Goals

Factor	1.1	1.2	1.3	1.4	1.5
Cost	61.07	57.31	55.66	52.91	51.69
Number of Iterations	7	4	4	3	3

This is a small-scale example but it demonstrates the approach. The times for these solutions were a few seconds per iteration. The very large systems we would like to handle require a larger test, in the following section.

5. MANY APPLICATIONS, AND SCALABILITY OF THE ANALYSIS

A cloud may host many applications, each one provided by a system perhaps similar in scale to the one modeled in Figure 9. Each application has its own workloads, resources and requirements. The goal of cloud management is to find the least-cost configuration which ensures that the performance of every class meets the requirement with quality and cost constraints.

To evaluate scalability of the computation, a system with 1 to 50 copies of the model of Figure 9 (each with different performance parameters and requirements) was used. The largest has 100 user classes, and over 1000 tasks across hundreds of hosts. Table 6 describes the optimization effort. It shows the iterations and the time to compute a full optimal deployment or re-deployment (Path (b) in Figure 2), vs. the number of copies, on a current-model PC.

Table 6 Optimization Effort with Increasing Scale

N Application Instances	1	10	20	30	40	50
Iterations	2	4	5	7	8	8
Time (sec)	0.968	15.2	38.5	94.5	167.9	258

The longest solution time is just over four minutes. This is a practical range of values for the purpose of computing a deployment, as changes to deployment take on the order of minutes even for just a few machines. The tool is not optimized in any way and can probably be made somewhat faster.

We can see that the time increases rather rapidly for larger systems, and this is due to a cubic term in the number of entries in the model, in the computational complexity of LQNS ([7], sec. 8.2.3). This suggests that for much larger systems than this, it will be necessary to partition the problem and optimize the deployment in batches of about 50 services.

5.1 Incremental Deployment

The same algorithm can be used to incrementally deploy one new application in a system (Path (a) in Figure 2). To fix the prior deployments, the input arcs to the already-deployed tasks are modified as follows:

- ***Fixed-deployment Constraint:*** on input arcs with zero flow, the max-flow constraint is changed from infinity to zero. The constraint label becomes [0,0,0].

To avoid disturbing the prior applications, their hosts are not used. To this end, output arcs from hosts are modified as follows:

- ***Non-interference or "no-sharing" constraint:*** from each host node with non-zero flow through the node (representing processors used by the prior-deployed applications), each output arc with zero flow is given a max-flow constraint of zero. The constraint label becomes [0,0,0].

The resulting optimization problem is equivalent to a reduced problem without the existing applications and their hosts.

The non-interfering case (no sharing of processors) case is equivalent to optimizing a single application alone, over the unused hosts. It was evaluated for incremental deployment of from 1 to 50 applications. The lack of sharing requires 10-20% more processors than a full optimization, as compared in Table 7.

Although all the applications are instances of one template, they have different workload parameters (chosen randomly) so some require more processing resources than others, and this explains why 10 instances require more than 10 times the processors of one instance, in the second column.

Since the hosts all have cost and capacity factors of 1.0, the cost found by Eq (3) is the sum of the host utilizations, and cost/processors is the average utilization. Full optimization increases the host utilizations, so the cost may be higher for a deployment with fewer processors and equal performance. For example row 1, which is suboptimal because sharing is constrained, has lower costs than row 2 but uses more processors. Eq. (3) is only an approximation, required by the NFM.

To control the waste of resources due to lack of sharing, one could periodically do a full redeployment (path (b) in Figure 2). The third row of Table 7 shows the result if a full optimization and redeployment is performed at N = 11, 21, 31, 41, with no-sharing incremental deployment in between.

Table 7 Cost for Incremental vs Full Optimization
(number of processors, with the cost by Eq. (3) in brackets)

N Applic. Instances	1	10	20	30	40	50
Incremental only (no sharing)	10 (7.41)	112 (84.5)	220 (167)	350 (276)	428 (347)	---
full optimizn at every N	10 (7.41)	101 (91.6)	188 (180)	304 (295)	358 (371)	454 (470)
periodic full optimization	10 (7.41)	112 (84.5)	210 (178)	323 (288)	386 (350)	476 (439)

Full optimization at every scale does succeed in deploying onto 10-20% fewer hosts. Lightly loaded applications would show a greater difference, because of greater opportunities for sharing. Periodic full optimization is between the two, and is closer to the full optimization result as the scale increases. In a sequence of incremental deployments, a full optimization provides de-fragmentation. It could be done periodically or when idle hosts become scarce.

6. CONCLUSIONS AND FUTURE WORK

This paper presents a new and effective solution to optimizing resource assignment for very large scale service centers in clouds. Deployment is addressed as a static feasibility/optimization problem captured by a performance model. Dynamic management is provided by solving a sequence of static problems as conditions change, and by incremental deployments, in which a new application is deployed without changing existing applications.

Examples have shown that this approach can solve very tough problems, such as minimizing multi-class response time with minimum cost in very large systems. It is a step towards advanced management tools for cloud computing. A tool has been implemented for our CERAS laboratory cloud.

A key contribution of the combination of NFM and analytical model (LQN) is its effectiveness in solving a non-linear constrained optimization problem by a series of LP solutions.

The approach can be adapted to optimizing other general performance problems, such as searching for the minimum replicas, or maximizing system workloads or the ratio of workloads/cost, etc. It can be adapted to limit the changes to existing deployments during a re-deployment (to avoid thrashing). The performance model can also be extended to represent allocation of a share of a host to a virtual machine, if the shares allocated by the NFM are to be enforced at run-time.

A significant issue which is not considered here is allocation of memory to tasks. Memory limits the number of tasks that can be deployed on a single processor, even if each one runs at low utilization. However even without memory constraints, the present algorithm gives a significant capability. Also, in practice the algorithm tends to allocate just a few tasks per processor, so a simple interim solution is to heuristically redistribute excess tasks away from any overfilled processors.

Acknowledgement

This research was supported by OCE, the Ontario Centres of Excellence, and by the IBM Toronto Centre for Advanced Studies, as part of the program of the Centre for Research in Adaptive Systems (CERAS).

7. REFERENCES

[1] Berger, M. J. and Bokhari, S. H. "A Partitioning Strategy for Nonuniform Problems on Multiprocessors". *IEEE Trans. Comput.* 36, 5 (May. 1987), 570-580.

[2] Bokhari, S. H. "Partitioning Problems in Parallel, Pipeline, and Distributed Computing". *IEEE Trans. Comput.* 37, 1 (Jan. 1988), 48-57.

[3] N. Bobroff, A. Kochut, and K. Beatty. "Dynamic placement of virtual machines for managing SLA violations". In *Proc. Integrated Management 2007*, pp 119-128, Munich, May 2007.

[4] CERAS (Centre of Excellence for Research in Adaptive Systems) https://www.cs.uwaterloo.ca/twiki/view/CERAS

[5] M. Chang, J. He, E. Castro-Leon. "Service-Orientation in the Computing Infrastructure ", *Proc. 2nd IEEE Int. Symp. on Service-Oriented System Engineering (SOSE'06)*

[6] E. Coffman, M. Garey, D. Johnson, "An application of bin-packing to multiprocessor scheduling", *SIAM J. Computing*, vol. 7, pp. 1-17, Feb. 1978

[7] Greg Franks, *Performance Analysis of Distributed Server Systems*, PhD. thesis, Carleton University, Jan. 2000.

[8] G. Franks, D. Petriu, M. Woodside, J. Xu, and P. Tregunno, "Layered bottlenecks and their mitigation," Proc 3rd Int. Conf. on Quantitative Evaluation of Systems QEST'2006, Riverside, CA, Sept 2006, pp. 103-114.

[9] G. Franks, T. Al-Omari, M. Woodside, O. Das, S. Derisavi, "Enhanced Modeling and Solution of Layered Queueing Networks", *IEEE Trans. on Software Eng*. Aug. 2008.

[10] J.W. Chinneck, "Processing Network Models of Energy/Environment Systems", *Computers and Industrial Engineering*, vol. 28, no. 1, pp. 179-189. 1995

[11] IBM, "From Cloud Computing to the New Enterprise Data Center", http://download.boulder.ibm.com/ibmdl/pub/software/dw/wes/hipods/CloudComputingNEDC_wp_28May.pdf, 2008.

[12] M. Litoiu, J. Rolia, G. Serazzi, "Designing Process Replication and Activation: A Quantitative Approach ", *IEEE Trans. Software Engineering*, v.26 n.12, p.1168-1178, Dec 2000

[13] M. Litoiu, APERA (Application Performance Evaluator and Resource Allocation Tool) http://www.alphaworks.ibm.com/tech/apera

[14] A. Karve, T. Kimbrel, G. Pacifici, M. Spreitzer, M. Steinder, M. Sviridenko, and A. Tantawi, "Dynamic placement for clustered web applications", *Proc. 15th Int. Conf. on the World Wide Web* May 2006. ACM, New York.

[15] D. Menascé, H. Ruan, H. Gomaa: "A framework for QoS-aware software components ", *Proc 3rd ACM Int. Workshop on Software and Performance (WOSP 2004*), pp 186-196, Jan 2004.

[16] D. Menascé, E. Casalicchio, A. Dubey, "A heuristic approach to optimal service selection in service oriented architectures ", *Proc 7th Int. Workshop on Software and Performance WOSP '08*, ACM, New York, NY, pp. 13-24.

[17] M. Steinder, I. Whalley, D. Carrera, I. Gaweda D. Chess, "Server virtualization in autonomic management of heterogeneous workloads ". *Proc. Integrated Management (IM 2007)*, Munich, May 2007.

[18] J. Rolia, K. Sevcik, "The Method of Layers ". *IEEE Trans. Softw. Eng.* 21, 8 (Aug. 1995), pp 689-700.

[19] J. Rolia, R. Friedrich, C. Patel, "Service Centric Computing - Next Generation Internet Computing". In *Performance 2002, Tutorial Lectures* eds M. Calzarossa, S. Tucci, LNCS, vol. 2459. Springer, pp 463-479.

[20] Tang, C., Steinder, M., Spreitzer, M., and Pacifici, G. "A scalable application placement controller for enterprise data centers ". In *Proc. 16th Int. Conf. on the World Wide Web*, Banff, May 2007, WWW '07. ACM, New York, pp 331-340.

[21] M. Tim Jones, "Cloud computing with Linux Cloud computing platforms and applications", http://www.ibm.com/developerWorks, 10 Sep 2008.

[22] C.M. Woodside G.G. Monforton, "Fast Allocation of Processes in Distributed and Parallel Systems", *IEEE Trans. on Parallel and Distributed Systems*, V. 4, N. 2, pp. 164-174, 1993.

[23] M. Woodside, "A Composable Performance Model for Service/Resource Systems", *Proc 7th Workshop on Performability Modelling of Computer and Communications Systems (PMCCS7)*, Torino, Italy, Sept 2005, pp 89-92

[24] C.M. Woodside, J.E. Neilson, D.C. Petriu and S. Majumdar, "The Stochastic Rendezvous Network Model for Performance of Synchronous Client-Server-Like Distributed Software ", *IEEE Trans Computers*, Vol. 44, No. 1, January 1995, pp. 20-34

[25] M. Woodside, "Software Resource Architecture ", *Int. Journal on Software Engineering and Knowledge Engineering* (IJSEKE), v 11, no 4, pp 407-429, 2001.

[26] T. Zheng, M. Woodside, M. Litoiu, "Performance Model Estimation and Tracking using Optimal Filters", *IEEE Trans. Software Engineering*, V 34 , no. 3 (May 2008) pp 391-406.

[27] M. Woodside, "The Relationship of Performance Models to Data", in *Performance Evaluation: Metrics, Models and Benchmarks (Proc SIPEW 2008)*, eds S. Kounev, I. Gorton, K. Sachs, Springer Verlag, LNCS vol 5119, pp 9-28, 2008.

[28] T. Zheng, J. Yang, M. Woodside, M. Litoiu, G. Iszlai, "Tracking Time-Varying Parameters in Software Systems with Extended Kalman Filters", *Proc CASCON 2005*, Toronto, Oct. 2005

[29] T. Zheng, "Model-based Dynamic Resource Management for Multi Tier Information Systems", PhD. thesis, Carleton University, Aug. 22, 2007