UNIT IV: Cloud Resource Management and Scheduling:
Policies and Mechanisms for Resource Management, Applications of control theory to task scheduling on a Cloud, Stability of a two-level resource allocation architecture, feedback control based on dynamic thresholds, coordination of specialized automatic performance managers, resource bundling, scheduling algorithms for computing Clouds, fair queuing, start time fair queuing.

1. Policies and Mechanisms for Resource Management:

A policy typically refers to the principal guiding decisions, whereas mechanisms represent the means to implement policies. Separation of policies from mechanisms is a guiding principle in computer science.

Cloud resource management policies can be loosely grouped into five classes:

1. Admission control.
2. Capacity allocation.
3. Load balancing.
4. Energy optimization.
5. Quality-of-service (QoS) guarantees.

The explicit goal of an admission control policy is to prevent the system from accepting workloads in violation of high-level system policies; for example, a system may not accept an additional workload that would prevent it from completing work already in progress or contracted.

Limiting the workload requires some knowledge of the global state of the system. In a dynamic system such knowledge, when available, is at best obsolete. Capacity allocation means to allocate resources for individual instances; an instance is an activation of a service. Locating resources subject to multiple global optimization constraints requires a search of a very large search space when the state of individual systems changes rapidly.

Load balancing and energy optimization can be done locally, but global load-balancing and energy optimization policies encounter the same difficulties as the one we have already discussed.

Load balancing and energy optimization are correlated and affect the cost of providing the services. Indeed, it was predicted that by 2012 up to 40% of the budget for IT enterprise infrastructure would be spent on energy.

The common meaning of the term load balancing is that of evenly distributing the load to a set of servers. For example, consider the case of four identical servers, A B C , , , and D , whose relative loads whose relative loads are 80% 60% 40%, and 20%, respectively, of their capacity. As a result of perfect load balancing, all servers would end with the same load - 50% of each server's capacity.

In cloud computing a critical goal is minimizing the cost of providing the service and, in particular, minimizing the energy consumption. This leads to a different meaning of the term load balancing instead of having the load evenly distributed among all servers, we want to concentrate it and use the smallest number of servers while switching the others to standby mode, a state in which a server uses less energy.

In our example, the load from D will migrate to A and the load from C will migrate to; thus, B A and B will be loaded at full capacity, whereas C and D will be switched to standby mode.

Quality of service is that aspect of resource management that is probably the most difficult to address and, at the same time, possibly the most critical to the future of cloud computing.

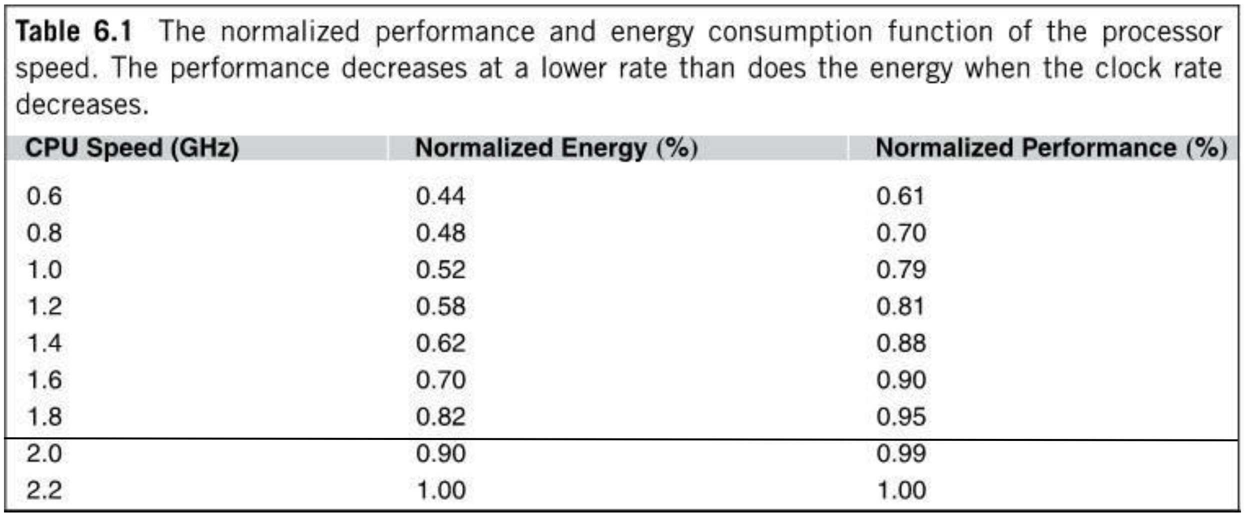


Table 6.1 The normalized performance and energy consumption function of the processor speed. The performance decreases at a lower rate than does the energy when the clock rate decreases.

CPU Speed (GHz)	Normalized Energy (%)	Normalized Performance (%)
0.6	0.44	0.61
0.8	0.48	0.70
1.0	0.52	0.79
1.2	0.58	0.81
1.4	0.62	0.88
1.6	0.70	0.90
1.8	0.82	0.95
2.0	0.90	0.99
2.2	1.00	1.00

<u>→Allocation techniques in computer clouds must be based on a disciplined approach rather than adhoc methods. The four basic mechanisms for the implementation of resource management policies are:</u>

1. Control theory: Control theory uses the feedback to guarantee system stability and predict transient behavior but can be used only to predict local rather than global behavior.

2. Machine learning: A major advantage of machine learning techniques is that they do not need a performance model of the system. This technique could be applied to coordination of several autonomic system managers.
3. Utility-based: Utility-based approaches require a performance model and a mechanism to correlate user-level performance with cost.
4. Market-oriented/economic mechanisms: Such mechanisms do not require a model of the system, e.g., combinatorial auctions for bundles of resources

A distinction should be made between interactive and non-interactive workloads. The management techniques for interactive workloads.

e.g., Web services, involve flow control and dynamic application.

2. Applications of control theory to task scheduling on a cloud:

Control theory has been used to design adaptive resource management for many classes of applications, including power management, task scheduling, QoS adaptation in Web servers ,and load balancing.

The classical feedback control methods are used in all these cases to regulate the key operating parameters of the system based on measurement of the system output; the feedback control in these methods assumes a linear time-invariant system model and a closed-loop controller.

This controller is based on an open-loop system transfer function that satisfies stability and sensitivity constraints.

The technique allows multiple QoS objectives and operating constraints to be expressed as a cost function and can be applied to stand-alone or distributed Web servers, database servers, high-performance application servers, and even mobile/embedded systems.

Control Theory Principles. We start our discussion with a brief overview of control theory principles one could use for optimal resource allocation. Optimal control generates a sequence of control inputs over a look-ahead horizon while estimating changes in operating conditions. A convex cost function has arguments $x(k)$, the state at step k, and $u(k)$, the control vector; this cost function is minimized, subject to the constraints imposed by the system dynamics. The discrete-time optimal control problem is to determine the sequence of control variables $u(i), u(i+1), \ldots, u(n-1)$ to minimize the expression

$$J(i) = \Phi(n, x(n)) + \sum_{k=i}^{n-1} L^k(x(k), u(k)), \qquad (6.1)$$

where $\Phi(n, x(n))$ is the cost function of the final step, n, and $L^k(x(k), u(k))$ is a time-varying cost function at the intermediate step k over the horizon $[i, n]$. The minimization is subject to the constraints

$$x(k+1) = f^k(x(k), u(k)), \qquad (6.2)$$

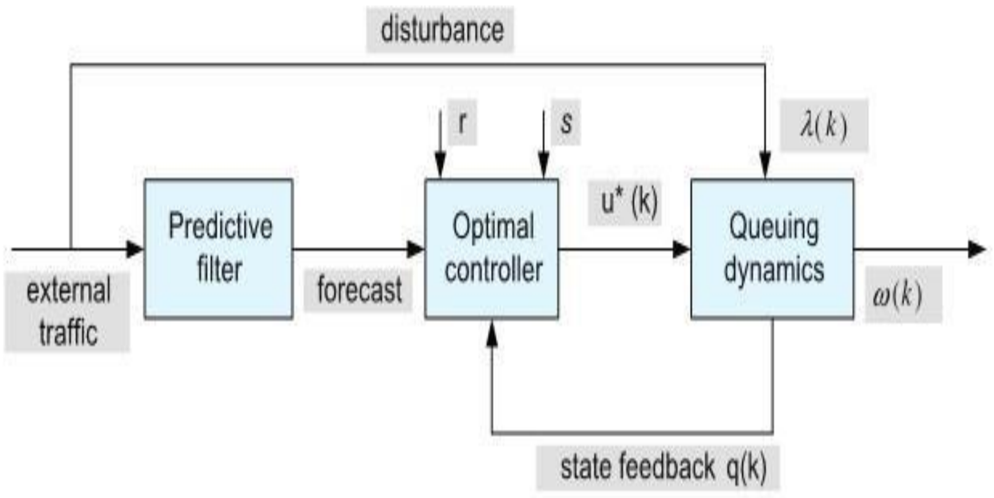


FIGURE 6.1

The structure of an optimal controller described in [369]. The controller uses the feedback regarding the current state as well as the estimation of the future disturbance due to environment to compute the optimal inputs over a finite horizon. The two parameters r and s are the weighting factors of the performance index.

where $x(k+1)$, the system state at time $k+1$, is a function of $x(k)$, the state at time k, and of $u(k)$, the input at time k; in general, the function f^k is time-varying; thus, its superscript.

One of the techniques to solve this problem is based on the *Lagrange multiplier* method of finding the extremes (minima or maxima) of a function subject to constrains. More precisely, if we want to maximize the function $g(x, y)$ subject to the constraint $h(x, y) = k$, we introduce a Lagrange multiplier λ. Then we study the function

$$\Lambda(x, y, \lambda) = g(x, y) + \lambda \times [h(x, y) - k]. \tag{6.3}$$

A necessary condition for the optimality is that (x, y, λ) is a stationary point for $\Lambda(x, y, \lambda)$. In other words,

$$\nabla_{x,y,\lambda}\Lambda(x, y, \lambda) = 0 \text{ or } \left(\frac{\partial\Lambda(x, y, \lambda)}{\partial x}, \frac{\partial\Lambda(x, y, \lambda)}{\partial y}, \frac{\partial\Lambda(x, y, \lambda)}{\partial\lambda}\right) = 0. \tag{6.4}$$

The Lagrange multiplier at time step k is λ_k and we solve Eq. (6.4) as an unconstrained optimization problem. We define an adjoint cost function that includes the original state constraints as the Hamiltonian function H, then we construct the adjoint system consisting of the original state equation and the *costate equation* governing the Lagrange multiplier. Thus, we define a two-point boundary problem[3]; the state x_k develops forward in time whereas the costate occurs backward in time.

A Model Capturing Both QoS and Energy Consumption for a Single-Server System. Now we turn our attention to the case of a single processor serving a stream of input requests. To compute the optimal inputs over a finite horizon, the controller in **Figure 6.1** uses feedback regarding the current state, as well as an estimation of the future disturbance due to the environment. The control task is solved as a state regulation problem updating the initial and final states of the control horizon.

The behavior of a single processor is modeled as a nonlinear, time-varying, discrete-time state equation. If T_s is the sampling period, defined as the time difference between two consecutive observations of the system, e.g., the one at time $(k+1)$ and the one at time k, then the size of the queue at time $(k+1)$ is

$$q(k+1) = \max\left\{\left[q(k) + \left(\hat{\Lambda}(k) - \frac{u(k)}{\hat{c}(k) \times u_{max}}\right) \times T_s\right], 0\right\}. \tag{6.5}$$

The first term, $q(k)$, is the size of the input queue at time k, and the second one is the difference between the number of requests arriving during the sampling period, T_s, and those processed during the same interval.

The response time $\omega(k)$ is the sum of the waiting time and the processing time of the requests

$$\omega(k) = (1 + q(k)) \times \hat{c}(k). \tag{6.6}$$

Indeed, the total number of requests in the system is $(1 + q(k))$ and the departure rate is $1/\hat{c}(k)$.

We want to capture both the QoS and the energy consumption, since both affect the cost of providing the service. A utility function, such as the one depicted in Figure 6.4, captures the rewards as well as the penalties specified by the service-level agreement for the response time. In our queuing model the utility is a function of the size of the queue; it can be expressed as a quadratic function of the response time

$$S(q(k)) = 1/2(s \times (\omega(k) - \omega_0)^2), \tag{6.7}$$

with ω_0, the response time set point and $q(0) = q_0$, the initial value of the queue length. The energy consumption is a quadratic function of the frequency

$$R(u(k)) = 1/2(r \times u(k)^2). \tag{6.8}$$

The two parameters s and r are weights for the two components of the cost, the one derived from the utility function and the second from the energy consumption. We have to pay a penalty for the requests left in the queue at the end of the control horizon, a quadratic function of the queue length

$$\Phi(q(N)) = 1/2(v \times q(n)^2). \tag{6.9}$$

The performance measure of interest is a cost expressed as

$$J = \Phi(q(N)) + \sum_{k=1}^{N-1}[S(q(k)) + R(u(k))]. \tag{6.10}$$

The problem is to find the optimal control u^* and the finite time horizon $[0, N]$ such that the trajectory of the system subject to optimal control is q^*, and the cost J in Eq. (6.10) is minimized subject to the following constraints

$$q(k+1) = \left[q(k) + \left(\hat{\Lambda}(k) - \frac{u(k)}{\hat{c}(k) \times u_{max}} \right) \times T_s \right], \; q(k) \geqslant 0, \text{ and } u_{min} \leqslant u(k) \leqslant u_{max}. \quad (6.11)$$

When the state trajectory $q(\cdot)$ corresponding to the control $u(\cdot)$ satisfies the constraints

$$\Gamma 1 : q(k) > 0, \quad \Gamma 2 : u(k) \geqslant u_{min}, \quad \Gamma 3 : u(k) \leqslant u_{max}, \quad (6.12)$$

then the pair $[q(\cdot), u(\cdot)]$ is called a *feasible state*. If the pair minimizes Eq. (6.10), then the pair is *optimal*.

The Hamiltonian H in our example is

$$H = S(q(k)) + R(u(k)) + \lambda(k+1) \times \left[q(k) + \left(\Lambda(k) - \frac{u(k)}{c \times u_{max}} \right) T_s \right] \\ + \mu_1(k) \times (-q(k)) + \mu_2(k) \times (-u(k) + u_{min}) + \mu_3(k) \times (u(k) - u_{max}). \quad (6.13)$$

According to Pontryagin's minimum principle,[4] the necessary condition for a sequence of feasible pairs to be optimal pairs is the existence of a sequence of costates λ and a Lagrange multiplier $\mu = [\mu_1(k), \mu_2(k), \mu_3(k)]$ such that

$$H(k, q^*, u^*, \lambda^*, \mu^*) \leqslant H(k, q, u^*, \lambda^*, \mu^*), \quad \forall q \geqslant 0 \quad (6.14)$$

where the Lagrange multipliers, $\mu_1(k), \mu_2(k), \mu_3(k)$, reflect the sensitivity of the cost function to the queue length at time k and the boundary constraints and satisfy several conditions

$$\mu_1(k) \geqslant 0, \; \mu_1(k)(-q(k)) = 0, \quad (6.15)$$
$$\mu_2(k) \geqslant 0, \; \mu_2(k)(-u(k) + u_{min}) = 0, \quad (6.16)$$
$$\mu_3(k) \geqslant 0, \; \mu_3(k)(u(k) - u_{max}) = 0. \quad (6.17)$$

3. Stability of a two-level resource allocation architecture:

A server with a closed-loop control system and can apply control theory principles to resource allocation.

Allocation architecture based on control theory concepts for the entire cloud. The automatic resource management is based on **two levels of controllers**, one for the service provider and one for the application, see **Figure 6.2**

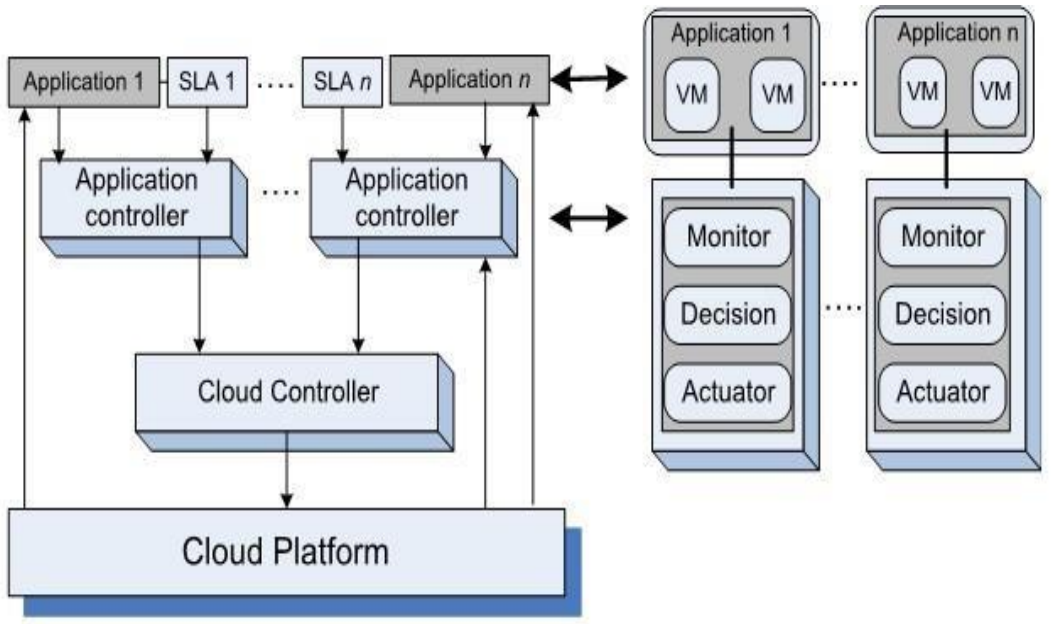


FIGURE 6.2

A two-level control architecture. Application controllers and cloud controllers work in concert.

→The main components of a control system are the inputs, the control system components, and the outputs. The inputs in such models are the offered workload and the policies for admission control, the capacity allocation, the load balancing, the energy optimization, and the QoS guarantees in the cloud.

→The system components are sensors used to estimate relevant measures of performance and controllers that implement various policies; the output is the resource allocations to the individual applications.

The controllers use the feedback provided by sensors to stabilize the system; stability is related to the change of the output. If the change is too large, the system may become unstable.

There are three main sources of instability in any control system:

1. The delay in getting the system reaction after a control action.
2. The granularity of the control, the fact that a small change enacted by the controllers leads to very large changes of the output.
3. Oscillations, which occur when the changes of the input are too large and the control is too weak, such that the changes of the input propagate directly to the output.

Two types of policies are used in autonomic systems:

(i) threshold-based policies
(ii) sequential decision policies based on Markovian decision models.

→In the first case, upper and lower bounds on performance trigger adaptation through resource reallocation. Such policies are simple and intuitive but require setting per-application thresholds.

4. Feedback control based on dynamic thresholds:

The elements involved in a control system are sensors, monitors, and actuators. The sensors measure the parameter(s) of interest, then transmit the measured values to a monitor , which determines whether the system behavior must be changed, and, if so, it requests that the actuators carry out the necessary actions.

The implementation of such a policy is challenging.

- First, due to the very large number of servers and to the fact that the load changes rapidly in time, the estimation of the current system load is likely to be inaccurate.
- Second, the ratio of average to maximal resource requirements of individual users specified in a service-level agreement is typically very high.

Thresholds: A threshold is the value of a parameter related to the state of a system that triggers a change in the system behavior. Thresholds are used in control theory to keep critical parameters of a system in a predefined range.

The threshold could be static , defined once and for all, or it could be dynamic . A dynamic threshold could be based on an average of measurements carried out over a time interval, a so-called integral control.

The dynamic threshold could also be a function of the values of multiple parameters at a given time or a mix of the two. To maintain the system parameters in a given range, **a high and a low** threshold are often defined.

The two thresholds determine different actions; **for example**, a high threshold could force the system to limit its activities and a low threshold could encourage additional activities.

Proportional Thresholding:

There are two types of controllers,

(1) Application controllers that determine whether additional resources are needed and
(2) Cloud controllers that arbitrate requests for resources and allocate the physical resources?
(3) Is it feasible to consider fine control?
(4) Are dynamic thresholds based on time averages better than static ones?
(5) Is it better to have a high and a low threshold, or it is suf?cient to de?ne only a high threshold?

The essence of the proportional thresholding is captured by the following algorithm:

1. Compute the integral value of the high and the low thresholds as averages of the maximum and, respectively, the minimum of the processor utilization over the process history.
2. Request additional VMs when the average value of the CPU utilization over the current time slice exceeds the high threshold.
3. Release a VM when the average value of the CPU utilization over the current time slice falls below the low threshold.

→The conclusions reached based on experiments with three VMs are as follows:
(a) Dynamic thresholds perform better than static ones and
(b) Two thresholds are better than one.

→Though conforming to our intuition, such results have to be justified by experiments in a realistic environment. Moreover, convincing results cannot be based on empirical values for some of the parameters required by integral control equations.

5. Coordination of specialized autonomic performance managers:

It Can specialized autonomic performance managers cooperate to optimize power consumption. The reports on actual experiments carried out on a set of blades mounted on a chassis (see **Figure 6.3** for the experimental setup).

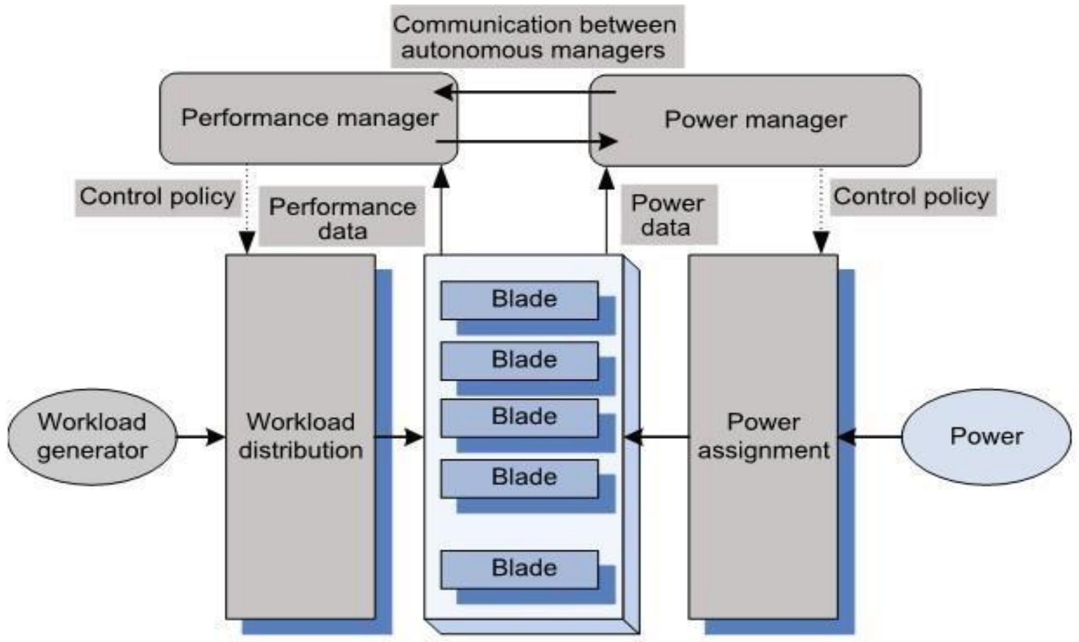


FIGURE 6.3

Autonomous performance and power managers cooperate to ensure SLA prescribed performance and energy optimization. They are fed with performance and power data and implement the performance and power management policies, respectively.

Virtually all modern processors support dynamic voltage scaling (DVS) as a mechanism for energy saving. Indeed, the energy dissipation scales quadratically with the supply voltage.

The management controls the CPU frequency and, thus, the rate of instruction execution. For some compute-intensive workloads the performance decreases linearly with the CPU clock frequency, whereas for others the effect of lower clock frequency is less noticeable or nonexistent. The clock frequency of individual blades/servers is controlled by a power manager, typically implemented in the firmware; it adjusts the clock frequency several times a second.

→The approach to coordinating power and performance management in is based on several ideas:

- Use a joint utility function for power and performance. The joint performance-power utility function, $U_{pp}(R, P)$, is a function of the response time, R, and the power, P, and it can be of the form

$$U_{pp}(R, P) = U(R) - \epsilon \times P \quad \text{or} \quad U_{pp}(R, P) = \frac{U(R)}{P}, \tag{6.18}$$

  with $U(R)$ the utility function based on response time only and ϵ a parameter to weight the influence of the two factors, response time and power.
- Identify a minimal set of parameters to be exchanged between the two managers.
- Set up a power cap for individual systems based on the utility-optimized power management policy.
- Use a standard performance manager modified only to accept input from the power manager regarding the frequency determined according to the power management policy. The power manager consists of Tcl (Tool Command Language) and C programs to compute the per-server (per-blade) power caps and send them via IPMI[5] to the firmware controlling the blade power. The power manager and the performance manager interact, but no negotiation between the two agents is involved.

- Use standard software systems. For example, use the WebSphere Extended Deployment (WXD), middleware that supports setting performance targets for individual Web applications and for the monitor response time, and periodically recompute the resource allocation parameters to meet the targets set. Use the Wide-Spectrum Stress Tool from the IBM Web Services Toolkit as a workload generator.

For practical reasons the utility function was expressed in terms of n_c, the number of clients, and p_κ, the powercap, as in

$$U'(p_\kappa, n_c) = U_{pp}(R(p_\kappa, n_c), P(p_\kappa, n_c)). \tag{6.19}$$

The optimal powercap p_κ^{opt} is a function of the workload intensity expressed by the number of clients, n_c,

$$p_\kappa^{opt}(n_c) = \arg\max U'(p_\kappa, n_c). \tag{6.20}$$

→Three types of experiments were conducted: (i) with the power management turned off; (ii) when the dependence of the power consumption and the response time were determined through a set of exhaustive experiments; and (iii) when the dependency of the powercap *pκ* on *nc* was derived via reinforcement-learning models.

The second type of experiment led to the conclusion that both the response time and the power consumed are nonlinear functions of the powercap, *pκ* , and the number of clients, *nc*; more specifically, the conclusions of these experiments are:

• At a low load the response time is well below the target of 1,000 msec.
• At medium and high loads the response time decreases rapidly when *pk* increases from 80 to 110 watts.
• For a given value of the powercap, the consumed power increases rapidly as the load increases.

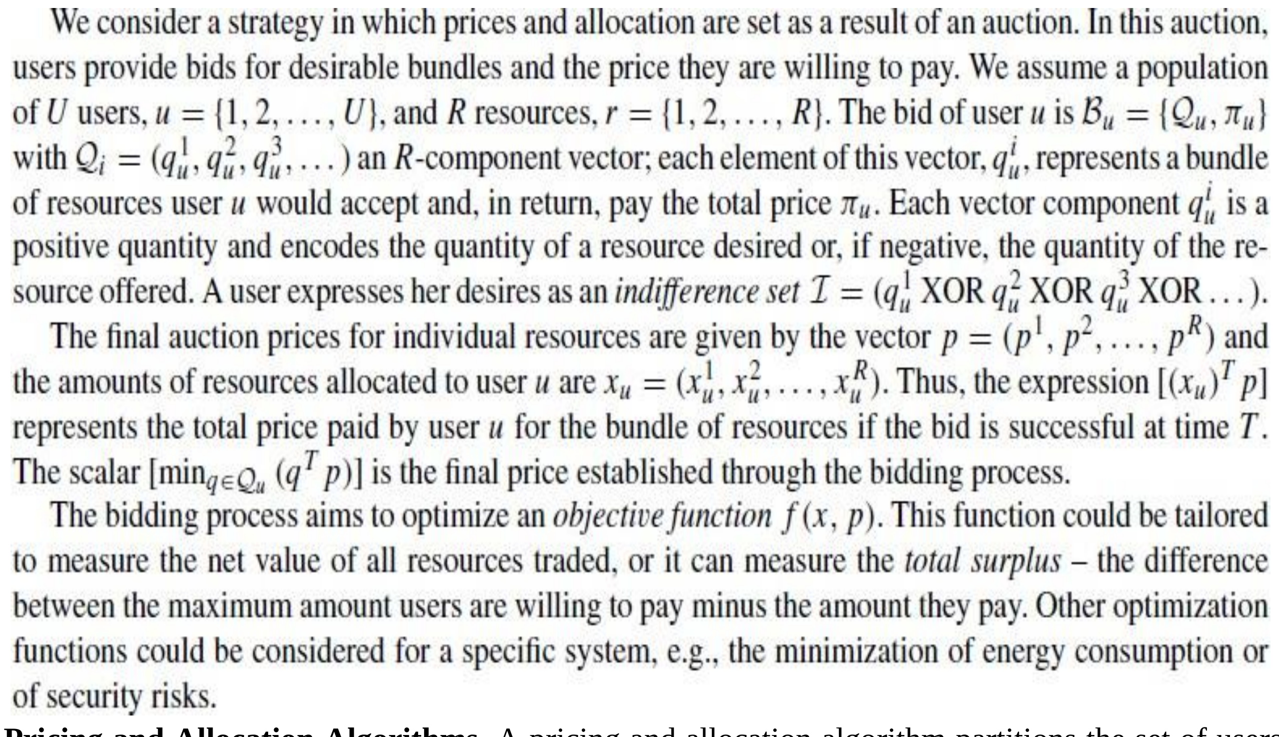
6. Resource bundling: Combinatorial auctions for cloud resources:

Resources in a cloud are allocated in bundles, allowing users get maximum benefit from a specific combination of resources. Indeed, along with CPU cycles, an application needs specific amounts of main memory, disk space, network bandwidth, and so on.

Resource bundling complicates traditional resource allocation models and has generated interest in economic models and, in particular, auction algorithms.

Combinatorial Auctions.: Auctions in which participants can bid on combinations of items, or pack- ages , are called combinatorial auctions. Such auctions provide a relatively simple, scalable, and solution to cloud resource allocation.

→Two recent combinatorial auction algorithms are the **simultaneous clock auction and clockproxy auction.**



We consider a strategy in which prices and allocation are set as a result of an auction. In this auction, users provide bids for desirable bundles and the price they are willing to pay. We assume a population of U users, $u = \{1, 2, \dots, U\}$, and R resources, $r = \{1, 2, \dots, R\}$. The bid of user u is $\mathcal{B}_u = \{\mathcal{Q}_u, \pi_u\}$ with $\mathcal{Q}_i = (q_u^1, q_u^2, q_u^3, \dots)$ an R-component vector; each element of this vector, q_u^i, represents a bundle of resources user u would accept and, in return, pay the total price π_u. Each vector component q_u^i is a positive quantity and encodes the quantity of a resource desired or, if negative, the quantity of the resource offered. A user expresses her desires as an *indifference set* $\mathcal{I} = (q_u^1 \text{ XOR } q_u^2 \text{ XOR } q_u^3 \text{ XOR} \dots)$.

The final auction prices for individual resources are given by the vector $p = (p^1, p^2, \dots, p^R)$ and the amounts of resources allocated to user u are $x_u = (x_u^1, x_u^2, \dots, x_u^R)$. Thus, the expression $[(x_u)^T p]$ represents the total price paid by user u for the bundle of resources if the bid is successful at time T. The scalar $[\min_{q \in \mathcal{Q}_u} (q^T p)]$ is the final price established through the bidding process.

The bidding process aims to optimize an *objective function* $f(x, p)$. This function could be tailored to measure the net value of all resources traded, or it can measure the *total surplus* – the difference between the maximum amount users are willing to pay minus the amount they pay. Other optimization functions could be considered for a specific system, e.g., the minimization of energy consumption or of security risks.

Pricing and Allocation Algorithms. A pricing and allocation algorithm partitions the set of users into two disjoint sets, winners and losers, denoted as *W* and *L*, respectively. The algorithm should:

1. Be computationally tractable. Traditional combinatorial auction algorithms such as Vickey-Clarke- Groves (VLG) fail this criteria, because they are not computationally tractable.
2. Scale well. Given the scale of the system and the number of requests for service, scalability is a necessary condition.
3. Be objective. Partitioning in winners and losers should only be based on the price *πu* of a user's bid. If the price exceeds the threshold, the user is a winner; otherwise the user is a loser.
4. Be fair.Make sure that the prices are *uniform*. All winners within a given resource pool pay the same price.
5. Indicate clearly at the end of the auction the unit prices for each resource pool.
6. Indicate clearly to all participants the relationship between the supply and the demand in the system.

The function to be maximized is

$$\max_{x,p} f(x, p). \tag{6.25}$$

→The constraints in **Table** 6.4 correspond to our intuition:

(a) the first one states that a user either gets one of the bundles it has opted for or nothing; no partial allocation is acceptable.
(b) The second constraint expresses the fact that the system awards only available resources; only offered resources can be allocated.
(c) The third constraint is that the bid of the winners exceeds the final price.
(d) The fourth constraint states that the winners get the least expensive bundles in their indifference set.
(e) The fifth constraint states that losers bid below the final price.
(f) The last constraint states that all prices are positive numbers.

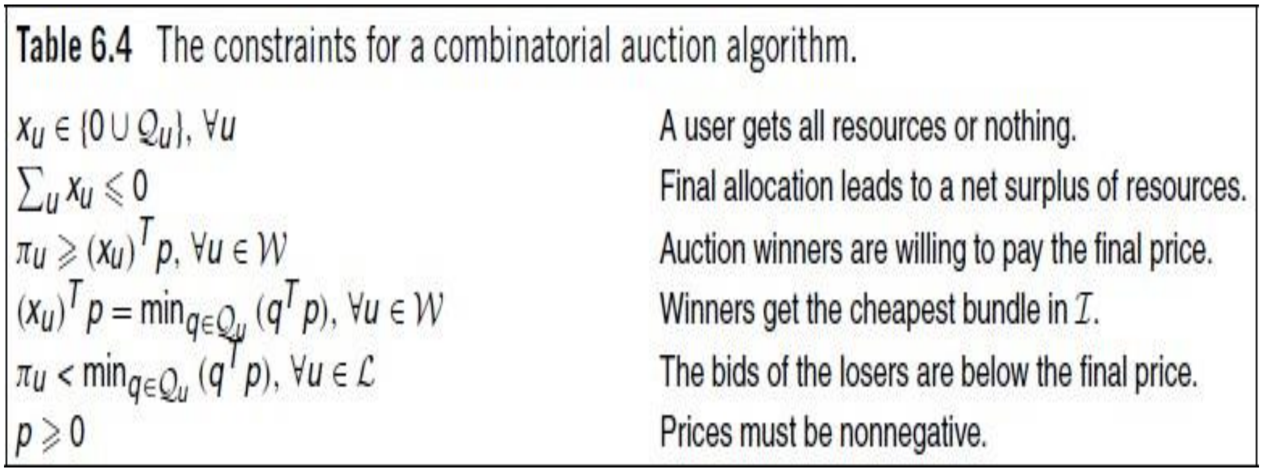


Table 6.4 The constraints for a combinatorial auction algorithm.

$x_u \in \{0 \cup Q_u\}, \forall u$	A user gets all resources or nothing.
$\sum_u x_u \leqslant 0$	Final allocation leads to a net surplus of resources.
$\pi_u \geqslant (x_u)^T p, \forall u \in \mathcal{W}$	Auction winners are willing to pay the final price.
$(x_u)^T p = \min_{q \in Q_u} (q^T p), \forall u \in \mathcal{W}$	Winners get the cheapest bundle in $\mathcal{I}$.
$\pi_u < \min_{q \in Q_u} (q^T p), \forall u \in \mathcal{L}$	The bids of the losers are below the final price.
$p \geqslant 0$	Prices must be nonnegative.

The ASCA Combinatorial Auction Algorithm.
In the ASCA algorithm the participants at the auction specify the resource and the quantities of that resource offered or desired at the price listed for that time slot. Then the excess vector

$$z(t) = \sum_u x_u(t) \tag{6.26}$$

is computed. If all its components are negative, the auction stops; negative components mean that the demand does not exceed the offer. If the demand is larger than the offer, *z(t)* _ 0, the auctioneer increases the price for items with a positive excess demand and solicits bids at the new price.

There is a slight complication as the algorithm involves user bidding in multiple rounds. To address this problem the user proxies automatically adjust their demands on behalf of the actual bidders, as shown in Figure 6.6. These proxies can be modeled as functions that compute the "best bundle" from each $\mathcal{Q}_u$ set given the current price

$$\mathcal{Q}_u = \begin{cases} \hat{q}_u & \text{if } \hat{q}_u^T p \leqslant \pi_u \quad \text{with} \quad \hat{q}_u \in \arg\min (q_u^T p) \\ 0 & \text{otherwise} \end{cases} .$$

The input to the ASCA algorithm: U users, R resources, $\bar{p}$ the starting price, and the update increment function, $g : (x, p) \mapsto \mathbb{R}^{\mathbb{R}}$. The pseudocode of the algorithm is:

```
1: set $t = 0$, $p(0) = \bar{p}$
2: loop
3:   collect bids $x_u(t) = \mathcal{G}_u(p(t))$, $\forall u$
4:   calculate excess demand $z(t) = \sum_u x_u(t)$
5:   if z(t) <0 then
6:     break
7:     else
```

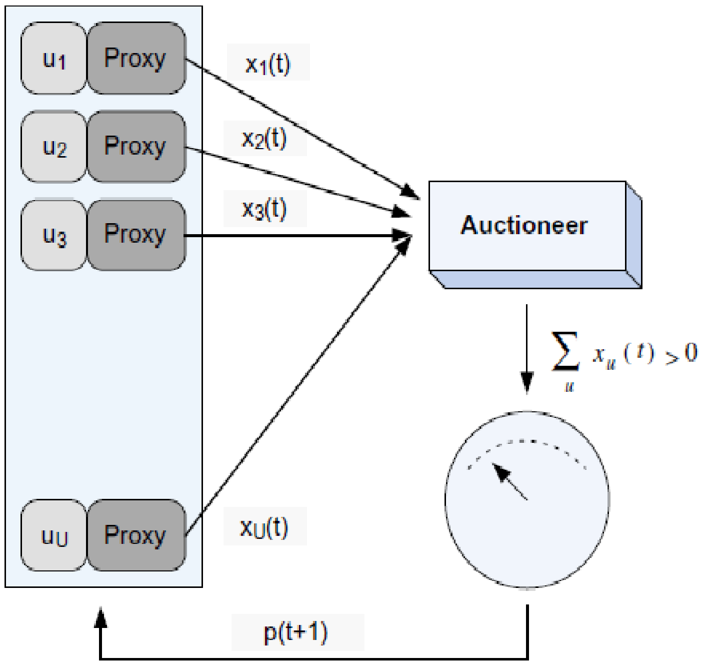
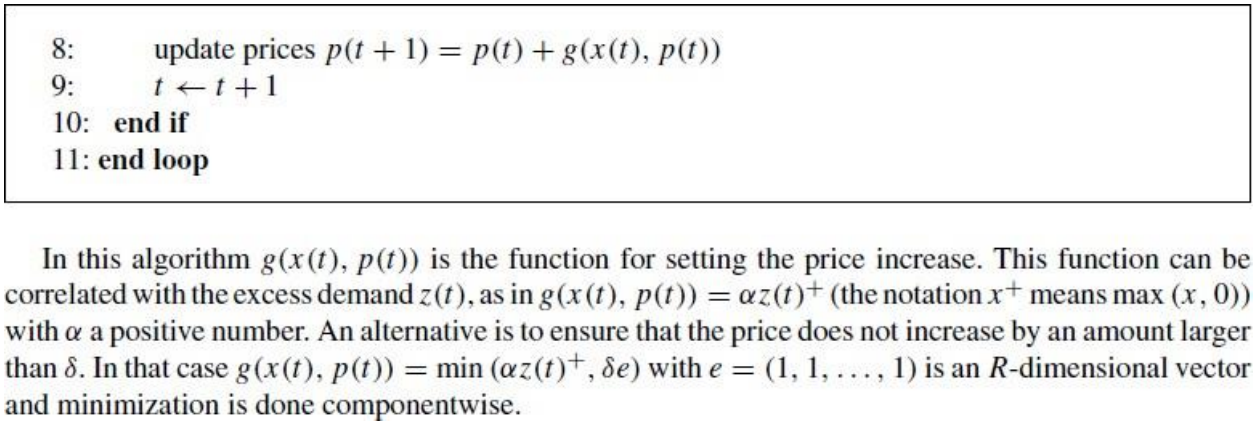


FIGURE 6.6

The schematics of the ASCA algorithm. To allow for a single round, auction users are represented by proxies that place the bids $x_u(t)$. The auctioneer determines whether there is an excess demand and, in that case, raises the price of resources for which the demand exceeds the supply and requests new bids.



```
8:      update prices $p(t+1) = p(t) + g(x(t), p(t))$
9:      $t \leftarrow t+1$
10:   end if
11: end loop
```

In this algorithm $g(x(t), p(t))$ is the function for setting the price increase. This function can be correlated with the excess demand $z(t)$, as in $g(x(t), p(t)) = \alpha z(t)^+$ (the notation x^+ means max $(x, 0)$) with α a positive number. An alternative is to ensure that the price does not increase by an amount larger than δ. In that case $g(x(t), p(t)) = \min(\alpha z(t)^+, \delta e)$ with $e = (1, 1, \ldots, 1)$ is an R-dimensional vector and minimization is done componentwise.

7. Scheduling algorithms for computing clouds:

Scheduling is a critical component of cloud resource management. Scheduling is responsible for resource sharing/multiplexing at several levels.

A server can be shared among several virtual machines, each virtual machine could support several applications, and each application may consist of multiple threads.

CPU scheduling supports the virtualization of a processor, the individual threads acting as virtual processors; a communication link can be multiplexed among a number of virtual channels, one for each flow.

Scheduling algorithm should be efficient, fair, and starvation-free. The objectives of a scheduler for a batch system are to maximize the throughput and to minimize the turnaround time submission and its completion.

Schedulers for systems supporting a mix of tasks – some with hard real-time constraints, others with soft, or no timing constraints – are often subject to contradictory requirements. Some schedulers are preemptive, allowing a high-priority task to interrupt the execution of a lower-priority one; others are non-preemptive

→Two distinct dimensions of resource management must be addressed by a scheduling policy:

a) the amount or quantity of resources allocated and
b) the timing when access to resources is granted.

→**Figure 6.7** identifies several broad classes of resource allocation requirements in the space defined by these two dimensions: best-effort, soft requirements, and hard requirements. Hard- real time systems are the most challenging because they require strict timing and precise amountsof resources.

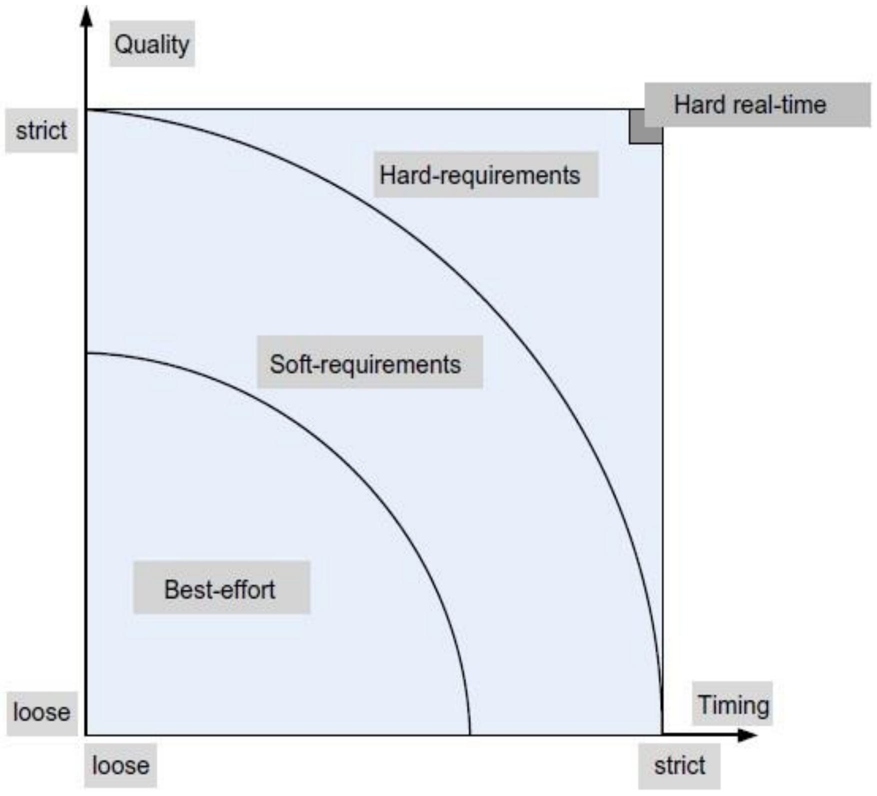


FIGURE 6.7

Best-effort policies do not impose requirements regarding either the amount of resources allocated to an application or the timing when an application is scheduled. Soft-requirements allocation policies require statistically guaranteed amounts and timing constraints; hard-requirements allocation policies demand strict timing and precise amounts of resources.

There are multiple definitions of a fair scheduling algorithm. First, we discuss the *max-min fairness criterion* [128]. Consider a resource with bandwidth B shared among n users who have equal rights. Each user requests an amount b_i and receives B_i. Then, according to the max-min criterion, the following conditions must be satisfied by a fair allocation:

C_1. The amount received by any user is not larger than the amount requested, $B_i \leqslant b_i$.

C_2. If the minimum allocation of any user is B_{min} no allocation satisfying condition C_1 has a higher B_{min} than the current allocation.

C_3. When we remove the user receiving the minimum allocation B_{min} and then reduce the total amount of the resource available from B to $(B - B_{min})$, the condition C_2 remains recursively true.

A fairness criterion for CPU scheduling [142] requires that the amount of work in the time interval from t_1 to t_2 of two runnable threads a and b, $\Omega_a(t_1, t_2)$ and $\Omega_b(t_1, t_2)$, respectively, minimize the expression

$$\left| \frac{\Omega_a(t_1, t_2)}{w_a} - \frac{\Omega_b(t_1, t_2)}{w_b} \right|, \tag{6.27}$$

where w_a and w_b are the weights of the threads a and b, respectively.

Round-robin, FCFS, shortest-job-first (SJF), and priority algorithms are among the most common scheduling algorithms for best-effort applications. Each thread is given control of the CPU for a definite period of time, called a time-slice , in a circular fashion in the case of round-robin scheduling.

The algorithm is fair and starvation-free. The threads are allowed to use the CPU in the order in which they arrive in the case of the FCFS algorithms and in the order of their running time in the case of SJF algorithms. Earliest deadline first (EDF) and rate monotonic algorithms (RMA) are used for real-time applications.

8.Fair queuing

→Computing and communication on a cloud are intimately related. Therefore, it should be no surprise that the first algorithm we discuss can be used for scheduling packet transmission as well as threads.

→Interconnection networks allow cloud servers to communicate with one another and with users. These networks consist of communication links of limited bandwidth and switches/routers/gateways of limited capacity. When the load exceeds its capacity, a switch starts dropping packets because it has limited input buffers for the switching fabric and for theoutgoing links, as well as limited CPU cycles.

→A fair queuing algorithm proposed in requires that separate queues, one per flow, be maintained by a switch and that the queues be serviced in a round-robin manner. This algorithm guarantees the fairness of buffer space management, but does not guarantee fairness of bandwidth allocation. Indeed, a flow transporting large packets will benefit from a larger bandwidth (see **Figure 6.8**).

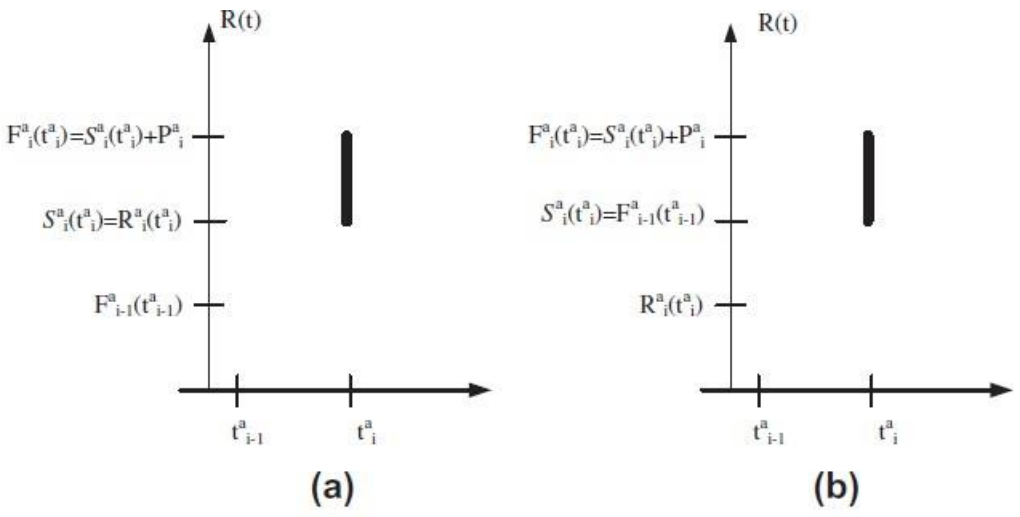

R(t)
$F^a_i(t^a_i)=S^a_i(t^a_i)+P^a_i$
$S^a_i(t^a_i)=R^a_i(t^a_i)$
$F^a_{i-1}(t^a_{i-1})$
t^a_{i-1}
t^a_i
(a)
R(t)
$F^a_i(t^a_i)=S^a_i(t^a_i)+P^a_i$
$S^a_i(t^a_i)=F^a_{i-1}(t^a_{i-1})$
$R^a_i(t^a_i)$
t^a_{i-1}
t^a_i
(b)

FIGURE 6.8

Transmission of a packet i of flow a arriving at time t_i^a of size P_i^a bits. The transmission starts at time $S_i^a = \max[F_{i-1}^a, R(t_i^a)]$ and ends at time $F_i^a = S_i^a + P_i^a$ with $R(t)$ the number of rounds of the algorithm. (a) The case $F_{i-1}^a < R(t_i^a)$. (b) The case $F_{i-1}^a \geqslant R(t_i^a)$.

The *fair queuing (FQ)* algorithm in [102] proposes a solution to this problem. First, it introduces a *bit-by-bit round-robin (BR)* strategy; as the name implies, in this rather impractical scheme a single bit from each queue is transmitted and the queues are visited in a round-robin fashion. Let $R(t)$ be the number of rounds of the BR algorithm up to time t and $N_{active}(t)$ be the number of active flows through the switch. Call t_i^a the time when the packet i of flow a, of size P_i^a bits arrives, and call S_i^a and F_i^a the values of $R(t)$ when the first and the last bit, respectively, of the packet i of flow a are transmitted. Then,

$$F_i^a = S_i^a + P_i^a \quad \text{and} \quad S_i^a = \max\left[F_{i-1}^a, R(t_i^a)\right]. \tag{6.28}$$

The quantities $R(t)$, $N_{active}(t)$, S_i^a, and F_i^a depend only on the arrival time of the packets, t_i^a, and not on their transmission time, provided that a flow a is active as long as

$$R(t) \leqslant F_i^a \quad \text{when} \quad i = \max\left(j | t_i^a \leqslant t\right). \tag{6.29}$$

The authors of [102] use for packet-by-packet transmission time the following nonpreemptive scheduling rule, which emulates the BR strategy: *The next packet to be transmitted is the one with the smallest F_i^a*. A preemptive version of the algorithm requires that the transmission of the current packet be interrupted as soon as one with a shorter finishing time, F_i^a, arrives.

A fair allocation of the bandwidth does not have an effect on the timing of the transmission. A possible strategy is to allow less delay for the flows using less than their fair share of the bandwidth. The same paper [102] proposes the introduction of a quantity called the *bid*, B_i^a, and scheduling the packet transmission based on its value. The bid is defined as

$$B_i^a = P_i^a + \max\left[F_{i-1}^a, \left(R\left(t_i^a\right) - \delta\right)\right], \tag{6.30}$$

with δ a nonnegative parameter. The properties of the FQ algorithm, as well as the implementation of a nonpreemptive version of the algorithms, are analyzed in [102].

9. Start time fair queuing:

→A hierarchical CPU scheduler for multimedia operating systems was proposed in. The basic idea of the start-time fair queuing (SFQ) algorithm is to organize the consumers of the CPU bandwidth in a tree structure; the root node is the processor and the leaves of this tree are the threads of each application.

→A scheduler acts at each level of the hierarchy. The fraction of the processor bandwidth, *B*, allocated to the intermediate node *i* is

$$\frac{B_i}{B} = \frac{w_i}{\sum_{j=1}^{n} w_j} \tag{6.31}$$

with w_j, $1 \leqslant j \leqslant n$, the weight of the n children of node i; see the example in Figure 6.9.

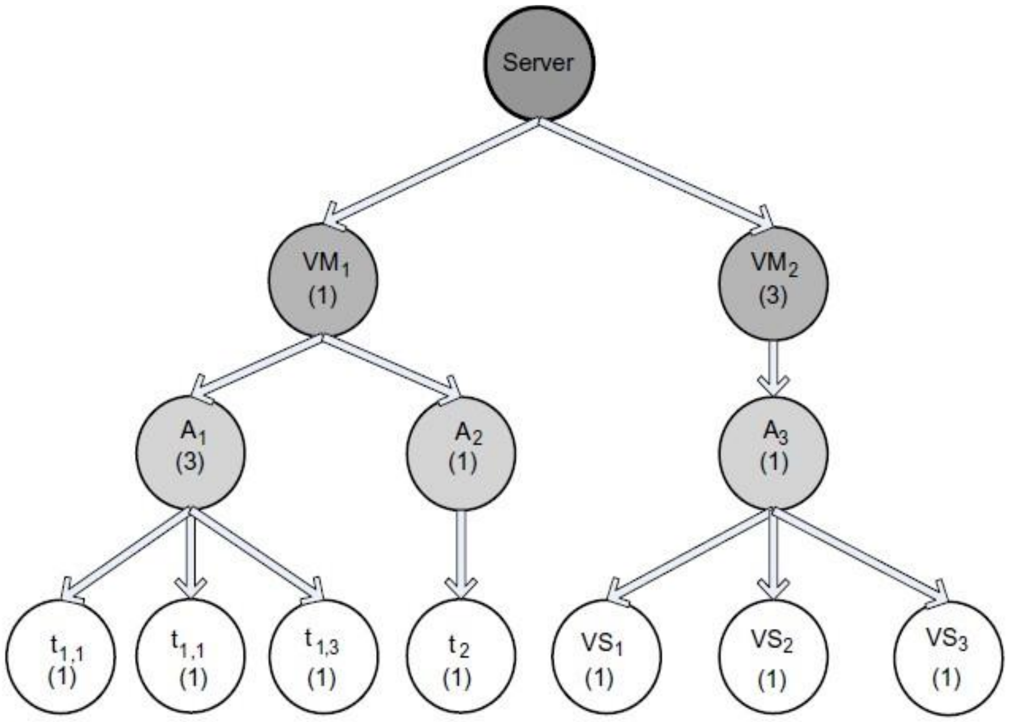


FIGURE 6.9

The SFQ tree for scheduling when two virtual machines, VM_1 and VM_2, run on a powerful server. VM_1 runs two best-effort applications A_1, with three threads $t_{1,1}$, $t_{1,2}$, and $t_{1,3}$, and A_2 with a single thread, t_2. VM_2 runs a video-streaming application, A_3, with three threads vs_1, vs_2, and vs_3. The weights of virtual machines, applications, and individual threads are shown in parenthesis.

When a virtual machine is not active, its bandwidth is reallocated to the other VMs active at the time. When one of the applications of a virtual machine is not active, its allocation is transferred to the other applications running on the same VM. Similarly, if one of the threads of an application is not runnable, its allocation is transferred to the other threads of the applications.

Call $v_a(t)$ and $v_b(t)$ the virtual time of threads a and b, respectively, at real time t. The virtual time of the scheduler at time t is denoted by $v(t)$. Call q the time quantum of the scheduler in milliseconds. The threads a and b have their time quanta, q_a and q_b, weighted by w_a and w_b, respectively; thus, in our example, the time quanta of the two threads are q/w_a and q/w_b, respectively. The i-th activation of thread a will start at the virtual time S_a^i and will finish at virtual time F_a^i. We call τ^j the real time of the j-th invocation of the scheduler.

An SFQ scheduler follows several rules:

R1. The threads are serviced in the order of their virtual start-up time; ties are broken arbitrarily.
R2. The virtual startup time of the i-th activation of thread x is

$$S_x^i(t) = \max\left[v\left(\tau^j\right), F_x^{(i-1)}(t)\right] \quad \text{and} \quad S_x^0 = 0. \tag{6.32}$$

The condition for thread i to be started is that thread $(i-1)$ has finished and that the scheduler is active.

R3. The virtual finish time of the i-th activation of thread x is

$$F_x^i(t) = S_x^i(t) + \frac{q}{w_x}. \tag{6.33}$$

A thread is stopped when its time quantum has expired; its time quantum is the time quantum of the scheduler divided by the weight of the thread.

R4. The virtual time of all threads is initially zero, $v_x^0 = 0$. The virtual time $v(t)$ at real time t is computed as follows:

$$v(t) = \begin{cases} \text{Virtual start time of the thread in service at time } t, & \text{if CPU is busy} \\ \text{Maximum finish virtual time of any thread,} & \text{if CPU is idle.} \end{cases} \tag{6.34}$$

In this description of the algorithm we have included the real time t to stress the dependence of all events in virtual time on the real time. To simplify the notation we use in our examples the real time as the index of the event. In other words, S_a^6 means the virtual start-up time of thread a at real time $t = 6$.

Example. The following example illustrates the application of the SFQ algorithm when there are two threads with the weights $w_a = 1$ and $w_b = 4$ and the time quantum is $q = 12$ (see Figure 6.10.)

Initially $S_a^0 = 0$, $S_b^0 = 0$, $v_a(0) = 0$, and $v_b(0) = 0$. Thread b blocks at time $t = 24$ and wakes up at time $t = 60$.

The scheduling decisions are made as follows:

1. $t = 0$: We have a tie, $S_a^0 = S_b^0$, and arbitrarily thread b is chosen to run first. The virtual finish time of thread b is

$$F_b^0 = S_b^0 + q/w_b = 0 + 12/4 = 3. \tag{6.35}$$

2. $t = 3$: Both threads are runnable and thread b was in service; thus, $v(3) = S_b^0 = 0$; then

$$S_b^1 = \max[v(3), F_b^0] = \max(0, 3) = 3. \tag{6.36}$$

But $S_a^0 < S_b^1$, thus thread a is selected to run. Its virtual finish time is

$$F_a^0 = S_a^0 + q/w_a = 0 + 12/1 = 12. \tag{6.37}$$

3. $t = 15$: Both threads are runnable, and thread a was in service at this time; thus,

$$v(15) = S_a^0 = 0 \tag{6.38}$$

and

$$S_a^1 = \max[v(15), F_a^0] = \max[0, 12] = 12. \tag{6.39}$$

As $S_b^1 = 3 < 12$, thread b is selected to run; the virtual finish time of thread b is now

$$F_b^1 = S_b^1 + q/w_b = 3 + 12/4 = 6. \tag{6.40}$$

4. $t = 18$: Both threads are runnable, and thread b was in service at this time; thus,

$$v(18) = S_b^1 = 3 \tag{6.41}$$

and

$$S_b^2 = \max[v(18), F_b^1] = \max[3, 6] = 6. \tag{6.42}$$

As $S_b^2 < S_a^1 = 12$, thread b is selected to run again; its virtual finish time is

$$F_b^2 = S_b^2 + q/w_b = 6 + 12/4 = 9. \tag{6.43}$$

5. $t = 21$: Both threads are runnable, and thread b was in service at this time; thus,

$$v(21) = S_b^2 = 6 \tag{6.44}$$

and

$$S_b^3 = \max[v(21), F_b^2] = \max[6, 9] = 9. \tag{6.45}$$

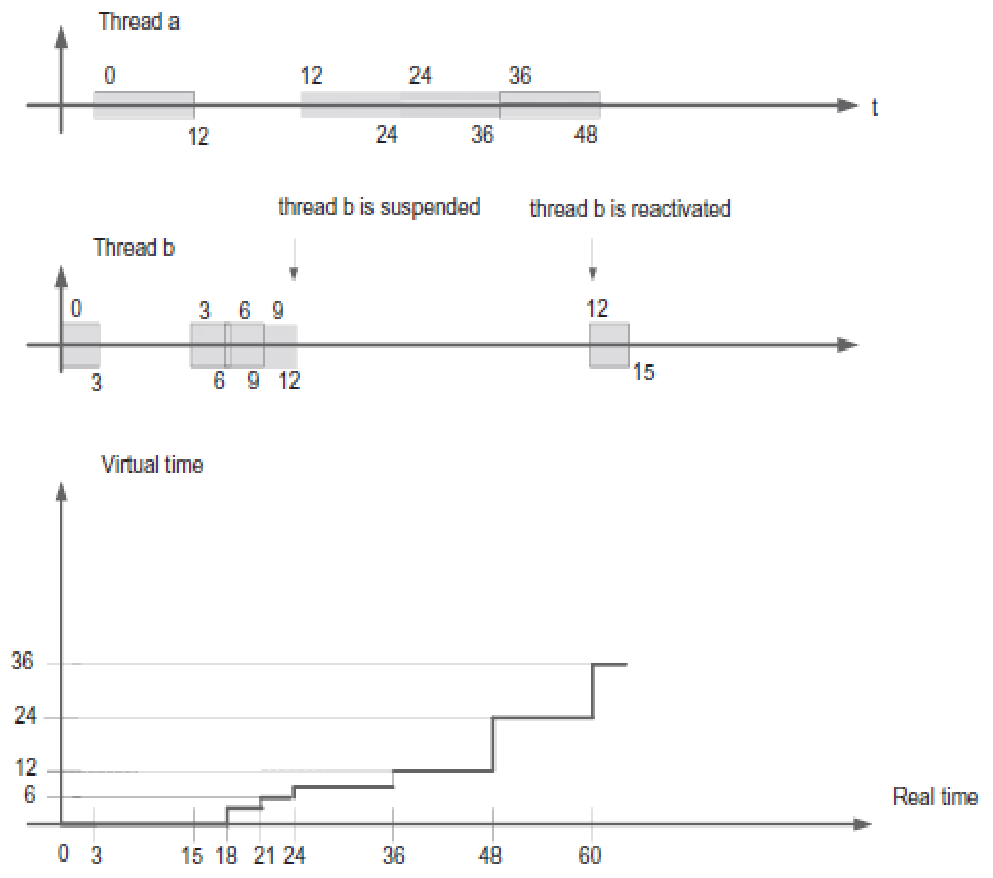


FIGURE 6.10

Top, the virtual start-up time $S_a(t)$ and $S_b(t)$ and the virtual finish time $F_a(t)$ and $F_b(t)$ function of the real time t for each activation of threads a and b, respectively, are marked at the top and bottom of the box representing a running thread. The virtual time of the scheduler $v(t)$ function of the real time is shown on the bottom graph.

As $S_b^2 < S_a^1 = 12$, thread b is selected to run again; its virtual finish time is

$$F_b^3 = S_b^3 + q/w_b = 9 + 12/4 = 12. \tag{6.46}$$

6. $t = 24$: Thread b was in service at this time; thus,

$$v(24) = S_b^3 = 9 \tag{6.47}$$

$$S_b^4 = \max[v(24), F_b^3] = \max[9, 12] = 12. \tag{6.48}$$

Thread b is suspended till $t = 60$; thus, thread a is activated. Its virtual finish time is

$$F_a^1 = S_a^1 + q/w_a = 12 + 12/1 = 24. \tag{6.49}$$

7. $t = 36$: Thread a was in service and the only runnable thread at this time; thus,

$$v(36) = S_a^1 = 12 \tag{6.50}$$

and

$$S_a^2 = \max[v(36), F_a^2] = \max[12, 24] = 24. \tag{6.51}$$

Then,

$$F_a^2 = S_a^2 + q/w_a = 24 + 12/1 = 36. \tag{6.52}$$

8. $t = 48$: Thread a was in service and is the only runnable thread at this time; thus,

$$v(48) = S_a^2 = 24 \tag{6.53}$$

and

$$S_a^3 = \max[v(48), F_a^2] = \max[24, 36] = 36. \tag{6.54}$$

Then,

$$F_a^3 = S_a^3 + q/w_a = 36 + 12/1 = 48. \tag{6.55}$$

9. $t = 60$: Thread a was in service at this time; thus,

$$v(60) = S_a^3 = 36 \tag{6.56}$$

and

$$S_a^4 = \max[v(60), F_a^3] = \max[36, 48] = 48. \tag{6.57}$$

But now thread b is runnable and $S_b^4 = 12$.
Thus, thread b is activated and

$$F_b^4 = S_b^4 + q/w_b = 12 + 12/4 = 15. \tag{6.58}$$

IMPORTANT QUESTIONS

1. Illustrate various Policies and Mechanisms for Resource Management?
2. Explain various Applications of control theory to task scheduling on a Cloud?
3. With a neat sketch explain Stability of a two-level resource allocation architecture.
4. Explain a two-level control architecture where application controllers and cloud controllers work in concert.
5. Explain feedback control based on dynamic thresholds?
6. Explain coordination of specialized automatic performance managers?
7. What is the role of power managers in cloud resource scheduling and management? Explain briefly.
8. State resource bundling? Explain combinational auctions?
9. Describe in brief about the scheduling algorithms for computing clouds.
10. Explain fair queuing?
11. With an example explain start time fair queuing algorithm?