Unit - 5

CLOUD RESOURCE MANAGEMENT AND SCHEDULING

Policies and mechanisms for resource management

A policy generally refers to the principal guiding decisions, whereas mechanisms represent the means to implement policies. 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 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 In a dynamic system such knowledge, when available, is at best outdated.

Capacity allocation means to allocate resources for individual instances or 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 .The common meaning of load balancing is that of evenly distributing the load to a set of servers.

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.

Quality of service is the most critical to the future of cloud computing. Resource management strategies jointly target performance and power consumption. Dynamic voltage and frequency scaling (DVFS) techniques lower the voltage and the frequency to decrease power consumption. As a result of lower voltages and frequencies, the performance of processors decreases, but at a substantially slower rate than the energy consumption.

The four basic mechanisms for the implementation of resource management policies are:

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.

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. A technique to design self-managing systems based on concepts from control theory is discussed . 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.

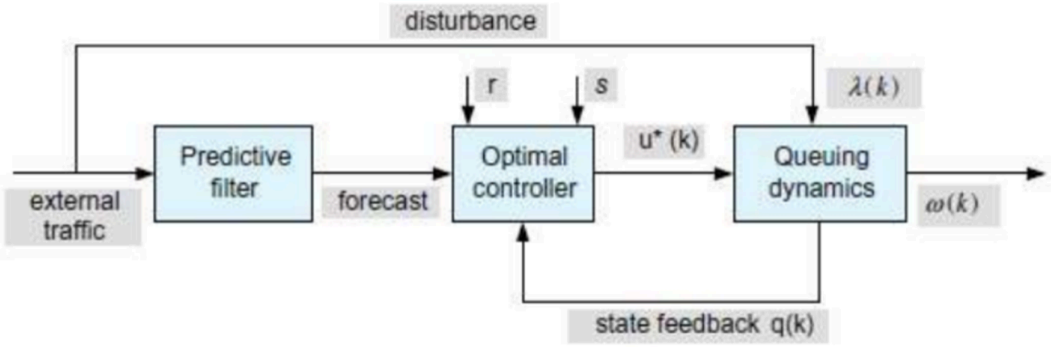
The following discussion considers a single processor serving a stream of input requests. We attempt to minimize a cost function that reflects the response time and the power consumption. Our goal is to illustrate the methodology for optimal resource management based on control theory concepts. The analysis is intricate and cannot be easily extended to a collection of servers.

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), . . . , u(n − 1) to minimize the expression

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

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

$$x(k+1) = f^k(x(k), u(k)), \tag{6.2}$$



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 »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 problem3; the state xk 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 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. We use a simple queuing model to estimate

the response time. Requests for service at processor P are processed on a first-come, first-served (FCFS) basis. We do not assume a priori distributions of the arrival process and of the service process; instead, we use the estimate $\Lambda(k)$ of the arrival rate $\Lambda(k)$ at time k. We also assume that the processor can operate at frequencies $u(k)$ in the range $u(k) \in [u_{min}, u_{max}]$ and call $\hat{c}(k)$ the time to process a request at time k when the processor operates at the highest frequency in the range, u_{max}. Then we define the scaling factor $\alpha(k) = u(k)/u_{max}$ and we express an estimate of the processing rate $N(k)$ as $\alpha(k)/\hat{c}(k)$.

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

$$\mu_1(k) \geqslant 0, \; \mu_1(k)(-q(k)) = 0, \quad (6.15)$$

$$\mu_2(k) \geqslant 0, \; \mu_2(k)(-u(k) + u_{min}) = 0, \quad (6.16)$$

$$\mu_3(k) \geqslant 0, \; \mu_3(k)(u(k) - u_{max}) = 0. \quad (6.17)$$

A detailed analysis of the methods to solve this problem and the analysis of the stability conditions is beyond the scope of our discussion and can be found in [369].

The extension of the techniques for optimal resource management from a single system to a cloud with a very large number of servers is a rather challenging area of research. The problem is even harder when, instead of transaction-based processing, the cloud applications require the implementation of a complex workflow.

Stability of two-level resource allocation architecture

The automatic resource management is based on two levels of controllers, one for the service provider and one for the application, shown in Fig 6.2. 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, the system could experience thrashing, the amount of useful time dedicated to the execution of applications becomes increasingly small and most of the system resources are occupied by management functions. 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.

system could experience thrashing, the amount of useful time dedicated to the execution of applications becomes increasingly small and most of the system resources are occupied by management functions.

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

1. The delay in getting the system reaction after a control action.

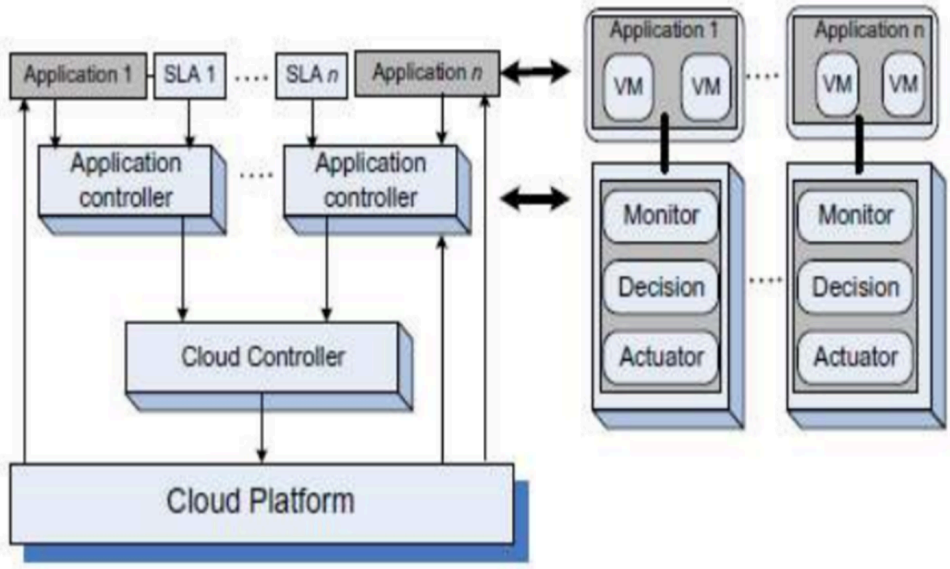


FIGURE 6.2

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

per-application thresholds. Lessons learned from the experiments with two levels of controllers and the two types of policies. A first observation is that the actions of the control system should be carried out in a rhythm that does not lead to instability. Adjustments should be carried out only after the performance of the system has stabilized. The controller should measure the time for an application to stabilize and adapt to the manner in which the controlled system reacts. If upper and lower thresholds are set, instability occurs when they are too close to one another if the

Two types of policies are used in autonomic systems:

(i)threshold-based policies and

(ii)sequential decision policies .

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. Lessons learned from the experiments with two levels of controllers and the two types of policies. A first observation is that the actions of the control system should be carried out in a rhythm that does not lead to instability. Adjustments should be carried out only after the performance of the system has stabilized. The controller should measure the time for an application to stabilize and adapt to the manner in which the controlled system reacts. If upper and lower thresholds are set, instability occurs when they are too close to one another if the variations of the workload are large enough and the time required to adapt does not allow the system to stabilize. The actions consist of allocation/deallocation of one or more virtual machines; sometimes allocation/deallocation of a single VM required by one of the thresholds may cause crossing of the other threshold and this may represent another source of instability.

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; 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. Often the parameter used for admission control policy is the current system load; when a threshold, e.g., 80%, is reached, the cloud stops accepting additional load.

In practice, the implementation of such a policy is challenging or infeasible.

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. 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. Proportional threshold: Application of these ideas to cloud computing, in particular to the IaaS delivery model, and a strategy for resource management is called proportional threshold. mix of the two.

The heart of the proportional threshold 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.

Coordination of specialized autonomic performance managers

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 power 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 non -existent.

- 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 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.

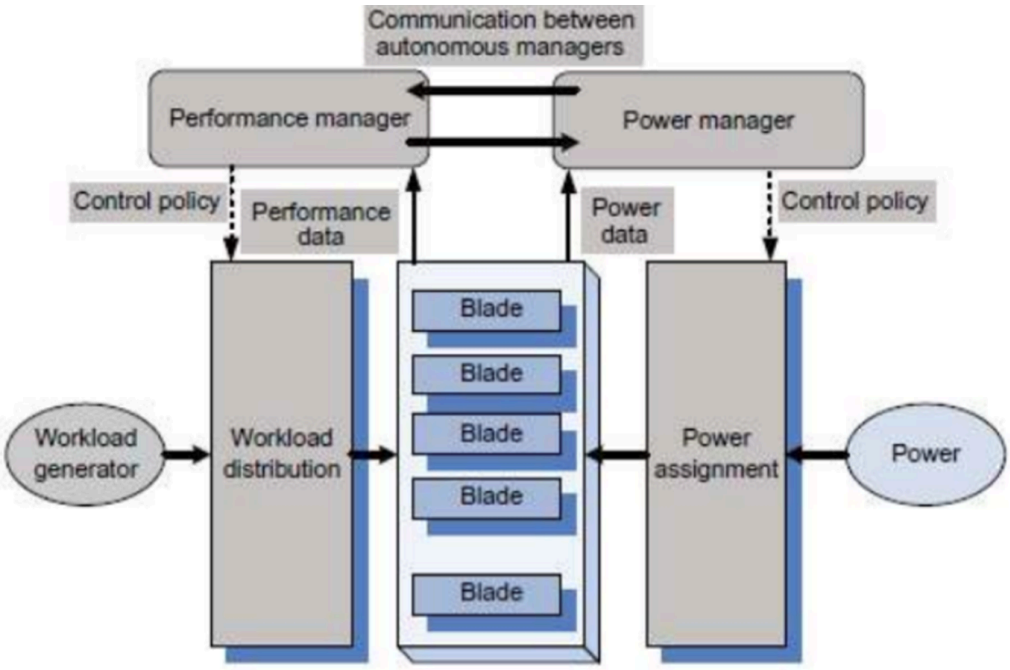
- The approach to coordinating power and performance management is based on several ideas:

- Use a joint utility function for power and performance. The joint performance-power utility function, Upp(R, P), is a function of the response time, R, and the power, P, and it can be of the form 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 Set up a power cap for individual systems based on the utility-optimized power management policy.

$$Upp(R, P) = U(R)/P$$

- 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 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), Wide-Spectrum Stress Tool from the IBM Web Services Toolkit as a workload generator.



Autonomous performance and power managers

Identify a minimal set of parameters to be exchanged between the two managers.

For practical reasons the utility function was expressed in terms of nc, the number of clients, and p_κ , the power cap, as in

$$U'(p_\kappa, nc) = Upp(R(p_\kappa, nc), P(p_\kappa, nc))$$

The optimal power cap $p_\kappa pt$ is a function of the workload intensity expressed by the number of clients,

$$nc \; p_\kappa pt(nc) = argmax U'(p_\kappa, nc).$$

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 when the dependency of the power cap p_κ 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 power cap, p_κ, 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 load the response time decreases rapidly when pk increases from 80 to 110 watts.
- For a given value of the power cap, the consumed power increases rapidly as the load increases.

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. In the context of cloud computing, an auction is the allocation of resources to the highest bidder. Combinatorial Auctions: Auctions, in which participants can bid on combinations of

items, or packages, are called combinatorial auctions. Such auctions provide a relatively simple, scalable, and tractable solution to cloud resource allocation. Two recent combinatorial auction algorithms are the simultaneous clock auction and the clock proxy auction and the algorithm ascending clock auction (ASCA).

In all these algorithms the current price for each resource is represented by a <clock= seen by all participants at the 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, . . . ,U}, and R resources, r = {1, 2, . . . , R}. The bid of user u is Bu = {Qu, πu} with Qi = (q1u , q2u , q3u , . . .) an R-component vector; each element of this vector, qiu, represents a bundle of resources user u would accept and, in return, pay the total price πu. Each vector component qiu 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 I = (q1u XOR q2u XOR q3u XOR . . .).

The final auction prices for individual resources are given by the vector p = (p1, p2, .. . , p R) and the amounts of resources allocated to user u are xu = (x1u , x2u, . . . , x Ru).Thus, the expression [(xu)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∈Qu (qTp)] 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 amounts 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 criterion, 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 for all participants the relationship between the supply and the demand in the system. The function to be maximized is

$$\max_{x,p} f(x, p)$$

The constraints in Table 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.

The ASCA Combinatorial Auction Algorithm: Informally, 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

$$z(t) = \sum xu(t)$$

Table 6.4 The constraints for a combinatorial auction algorithm.

$x_u \in \{0 \cup \mathcal{Q}_u\}, \forall u$	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.
$(x_u)^T p = \min_{q \in \mathcal{Q}_u} (q^T p), \forall u \in \mathcal{W}$	Winners get the cheapest bundle in $\mathcal{I}$.
$\pi_u < \min_{q \in \mathcal{Q}_u} (q^T p), \forall u \in \mathcal{L}$	The bids of the losers are below the final price.
$p \geqslant 0$	Prices must be nonnegative.

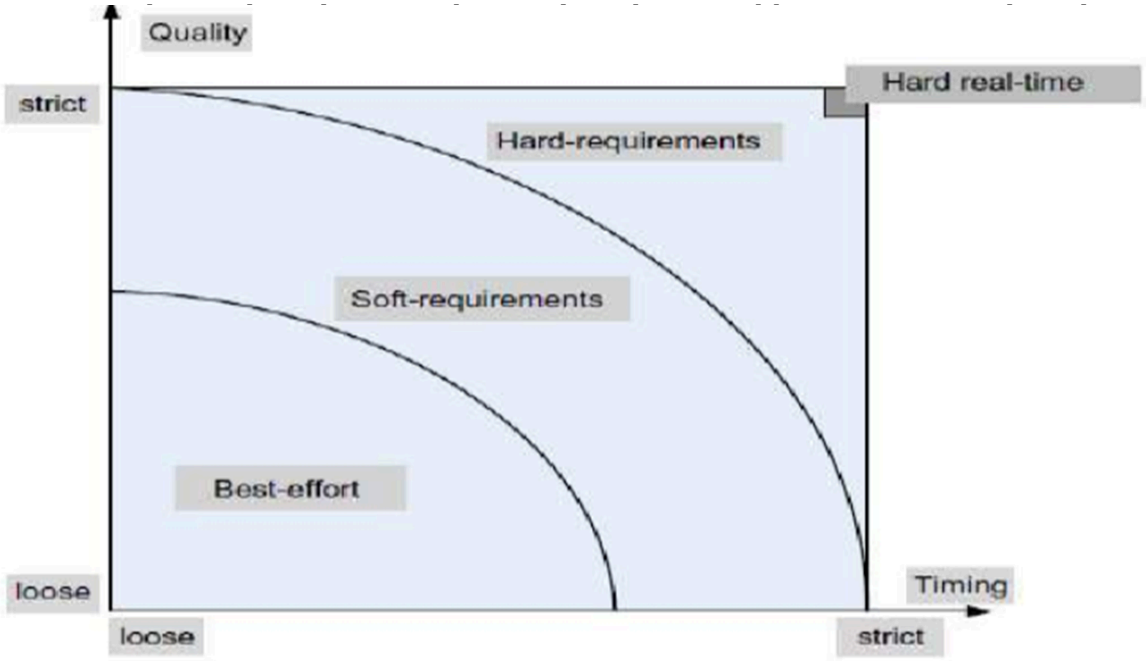
is computed. If all its components are negative, the auction stops; negative components mean that the demand does not exceed the offer. If the demand is larger than the offer, z(t) g 0, the auctioneer increases the price for items with a positive excess demand and solicits bids at the new price. Note that the algorithm satisfies conditions 1 through 6; all users discover the price at the same time and pay or receive a "fair = payment relative to uniform resource prices, the computation is tractable, and the execution time is linear in the number of participants at the auction and the number of resources. The computation is robust and generates plausible results regardless of the initial parameters of the system. The convergence of the optimization problem is guaranteed only if all participants at the auction are either providers of resources or consumers of resources, but not both providers and consumers at the same time. Nevertheless, the clock algorithm only finds a feasible solution; it does not guarantee its optimality. The algorithm implemented and allowed internal use of it within Google. Their preliminary experiments show that the system led to substantial improvements.

One of the most interesting side effects of the new resource allocation policy is that users were encouraged to make their applications more flexible and mobile to take advantage of the flexibility of the system controlled by the ASCA algorithm. An auctioning algorithm is very appealing because it supports resource bundling and does not require a model of the system.

At the same time, a practical implementation of such algorithms is challenging. First, requests for service arrive at random times, whereas in an auction all participants must react to a bid at the same time. Periodic auctions must then be organized, but this adds to the delay of the response. Second, there is an incompatibility between cloud elasticity, which guarantees that the demand for resources of an existing application will be satisfied immediately, and the idea of periodic auctions.

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



(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 amounts of resources. There are multiple definitions of a fair scheduling algorithm. To discuss the max-min fairness criterion, consider a resource with bandwidth B shared among n users who have equal rights.

Each user requests an amount bi and receives Bi. Then, according to the max-min criterion, the following conditions must be satisfied by a fair allocation.

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

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

higher Bmin than the current allocation. C3. When we remove the user receiving the minimum allocation Bmin and then reduce the total amount of the resource available from B to (B − Bmi n), the condition C2 remains recursively truewhere wa and wb are the weights of the threads a and b, respectively. The quality-of-service (QoS) requirements differ for different classes of cloud applications and demand different scheduling policies. Best-effort applications such as batch applications and analytics do not require QoS guarantees. Multimedia applications such as audio and video streaming have soft real-time constraints and require statistically guaranteed maximum delay and throughput. Applications with hard real-time constraints do not use a public cloud at this time but may do so in the future. 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. Integration of scheduling for the three classes of application, and two new algorithms for integrated scheduling, resource allocation/dispatching (RAD) and rate-based earliest deadline (RBED) are proposed.

Fair queuing

Computing and communication on a cloud are intimately related. Interconnection networks allow cloud servers to communicate with one another and with users. These networks consist of communication links of limited bandwidth and

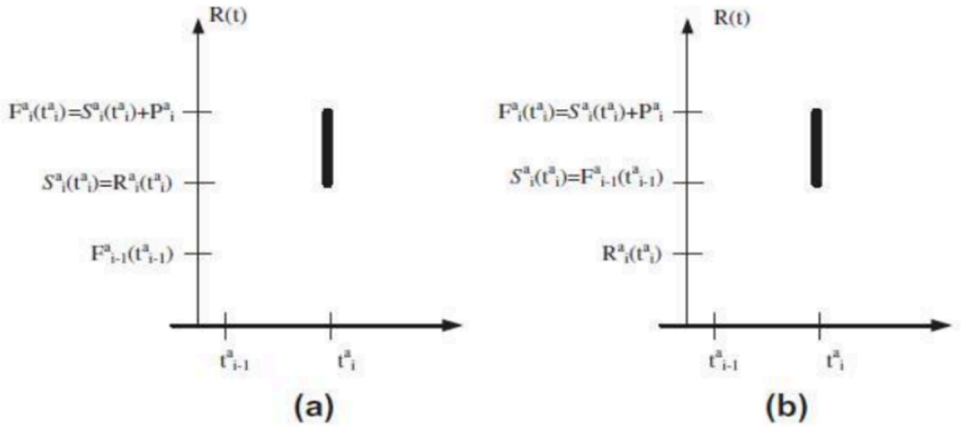


FIGURE 6.8

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

transmit at a higher rate and benefit from a larger share of the bandwidth. To address this problem, a fair queuing algorithm requires that separate queues, one per flow, be maintained by a switch and that the queues are 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.

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 *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

Start-time fair queuing

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 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 va (t) and vb(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, qa and qb, weighted by wa and wb, respectively; thus, in our example, the time quanta of the two threads are q/wa and q/wb, respectively. The i -th activation of thread a will start

at the virtual time Sia and will finish at virtual time Fia. We call Ä 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

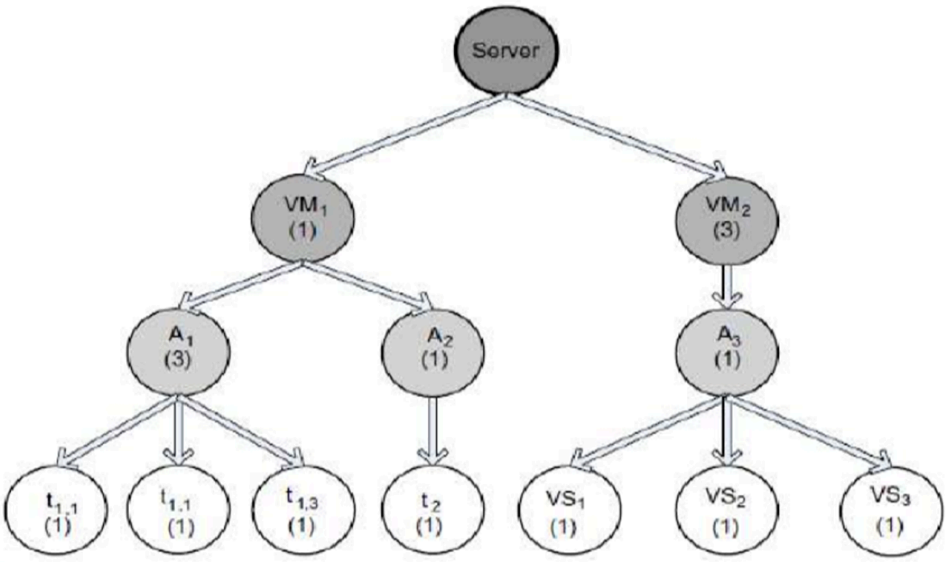


FIGURE 6.9

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.

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

$$Fix (t) = Six (t) + q/wx$$

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, v0x = 0. The virtual time v(t) at real time t is computed as follows:

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

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.

Borrowed virtual time

The objective of the borrowed virtual time (BVT) algorithm is to support low-latency dispatching of real-time applications as well as a weighted sharing of the CPU among several classes of applications. Like SFQ, the BVT algorithm supports scheduling of a mix of applications, some with hard, some with soft real-time constraints, and

applications demanding only a best effort. Thread i has an effective virtual time, Ei , an actual virtual time, Ai ,and a virtual time warp, Wi. The scheduler thread maintains its own scheduler virtual time (SVT), defined as the minimum actual virtual time A j of any thread. The threads are dispatched in the order of their effective virtual time, Ei, a policy called the earliest virtual time (EVT). The virtual time warp allows a thread to acquire an earlier effective virtual time - in other words, to borrow virtual time from its future CPU allocation.

The virtual warp time is enabled when the variable warp Back is set. In this case a laten The algorithm measures the time in minimum charging units (mcu) and uses a time quantum called context switch allowance (C), which measures the real time a thread is allowed to run when competing with other threads, measured in multiples of mcu. Typical values for the two quantities are mcu = 100 ¼sec and C = 100 msec. A thread is charged an integer number of mcu. Context switches are triggered by traditional events, the running thread is blocked waiting for an event to occur, the time quantum expires, and an interrupt occurs. Context switching also occurs when a thread becomes runnable after sleeping. When the thread Äi becomes runnable after sleeping, its actual virtual time is updated as follows: cy-sensitive thread gains dispatching preference as

$$Ai \leftarrow \max[Ai, SV\ T]$$

This policy prevents a thread sleeping for a long time to claim control of the CPU for a longer period of time than it deserves.

$$E_i \leftarrow \begin{cases} A_i & \text{if } warpBack = OFF \\ A_i - W_i & \text{if } warpBack = ON. \end{cases}$$

Cloud scheduling subject to deadlines

Often, an SLA specifies the time when the results of computations done on the cloud should be available. This motivates us to examine cloud scheduling subject to deadlines, a topic drawing on a vast body of literature devoted to real-time applications. Task Characterization and Deadlines: Real-time applications involve periodic or aperiodic tasks with deadlines. A task is characterized by a tuple (Ai, σi , Di), where Ai is the arrival time, σi > 0 is the data size of the task, and Di is the relative deadline. Instances of a periodic task, and arrive at times A0, A1, . . . Ai , . . . , with Ai+1 − Ai = q. The deadlines satisfy the constraint Di Ai+1 and generally the data size is the same, σi = σ. The individual instances of aperiodic tasks, i , are different. Their arrival times Ai are generally uncorrelated, and the amount of data σi is different for different instances. The absolute deadline for the aperiodic task i is (Ai + Di). We distinguish hard deadlines from soft deadlines.

If there are no interrupts, threads are allowed to run for the same amount of virtual time. Individual threads have weights; a thread with a larger weight consumes its virtual time more slowly. In practice, each thread τ_i maintains a constant k_i and uses its weight w_i to compute the amount Δ used to advance its actual virtual time upon completion of a run:

$$A_i \leftarrow A_i + \Delta. \tag{6.61}$$

Given two threads a and b,

$$\Delta = \frac{k_a}{w_a} = \frac{k_b}{w_b}. \tag{6.62}$$

The EVT policy requires that every time the actual virtual time is updated, a context switch from the current running thread τ_i to a thread τ_j occurs if

$$A_j \leqslant A_i - \frac{C}{w_i}. \tag{6.63}$$

on it may be affected and there are penalties, a hard deadline is strict and expressed precisely as milliseconds or possibly seconds. Soft deadlines play more of a guideline role and, in general, there are no Penalties. Soft deadlines can be missed by fractions of the units used to express them, e.g., minutes. The scheduling of tasks on a cloud is generally subject to soft deadlines, though occasionally applications with hard deadlines may be encountered.

The *workload derivative* $DC_i(n^{min})$ of a task Π_i when n^{min} nodes are assigned to the application, is defined as

$$DC_i(n^{min}) = W_i\left(n_i^{min} + 1\right) - W_i\left(n_i^{min}\right), \tag{6.69}$$

with $W_i(n)$ the workload allocated to task Π_i when n nodes of the cloud are available; if $\mathcal{E}(\sigma_i, n)$ is the execution time of the task, then $W_i(n) = n \times \mathcal{E}(\sigma_i, n)$. The MWF policy requires that:

with a head node called S0 and n worker nodes S1, S2,. . . Sn. The system is homogeneous, all workers are identical, and the communication time from the head node to any worker node is the same. The head node distributes the workload to worker nodes, and this distribution is done sequentially. In this context there are two important problems:

1. The order of execution of the tasks i .

2. The workload partitioning and the task mapping to worker nodes. Scheduling Policies: The most common scheduling policies used to determine the order of execution of the tasks are:

- First in, first out (FIFO). The tasks are scheduled for execution in the order of their

Thus, the condition that query Q = (A, σ, D) with arrival time A meets the deadline D can be expressed as

$$t_m^0 + \sigma\left[\frac{\rho_m}{n_m} + \phi\left(\frac{\rho_r}{n_r} + \tau\right)\right] \leqslant A + D. \qquad (6.87)$$

It follows immediately that the maximum value for the start-up time of the reduce task is

$$t_r^{max} = A + D - \sigma\phi\left(\frac{\rho_r}{n_r} + \tau\right). \qquad (6.88)$$

derivative DCI first.

2.The number n of nodes assigned to the application is kept to a minimum, nii n . We discuss two workload partitioning and task mappings to worker nodes, optimal and the equal partitioning.

Scheduling MapReduce applications subject to deadlines

Now we turn our attention to scheduling of MapReduce applications on the cloud subject to deadlines.

Several options for scheduling Apache Hadoop, an open-source implementation of the MapReduce algorithm, are:

- The default FIFO schedule.
- The Fair Scheduler .
- The Capacity Scheduler.
- The Dynamic Proportional Scheduler .

The deadline scheduling framework analyze Hadoop tasks. Table 6.8summarizes the notations used for the analysis of Hadoop; the term slots is equivalent with nodes and means the number of instances. We make two assumptions for our initial derivation:

The system is homogeneous; this means that ρm and ρr, the cost of processing a unit data by the map and the reduce task, respectively, are the same for all servers.

Load equipartition. Under these conditions the duration of the job J with input of size Ã is

$$\mathcal{E}(n_m, n_r, \sigma) = \sigma \left[\frac{\rho_m}{n_m} + \phi \left(\frac{\rho_r}{n_r} + \tau \right) \right].$$

We now plug the expression of the maximum value for the start-up time of the reduce task into the condition to meet the deadline

$$t_m^0 + \sigma \frac{\rho_m}{n_m} \leqslant t_r^{max}. \qquad (6.89)$$

It follows immediately that n^{min}_m , the minimum number of slots for the map task, satisfies the condition

$$n_m^{min} \geqslant \frac{\sigma \rho_m}{t_r^{max} - t_m^0}, \quad \text{thus,} \quad n_m^{min} = \lceil \frac{\sigma \rho_m}{t_r^{max} - t_m^0} \rceil. \qquad (6.90)$$

The assumption of homogeneity of the servers can be relaxed and we assume that individual servers have different costs for processing a unit workload $\rho_m^i \neq \rho_m^j$ and $\rho_t^i \neq \rho_t^j$. In this case we can use the minimum values $\rho_m = \min \rho_m^i$ and $\rho_r = \min \rho_r^i$ in the expression we derived.

A Constraints Scheduler based on this analysis and an evaluation of the effectiveness of this scheduler are presented.

Table 6.8 The parameters used for scheduling with deadlines.

Name	Description
Q	The query $Q = (A, \sigma, D)$
A	Arrival time of query Q
D	Deadline of query Q
Π_m^i	A map task, $1 \leqslant i \leqslant u$
Π_r^j	A reduce task, $1 \leqslant j \leqslant v$
J	The job to perform the query $Q = (A, \sigma, D)$, $J = \left(\Pi_m^1, \Pi_m^2, \ldots, \Pi_m^u, \Pi_r^1, \Pi_r^2, \ldots, \Pi_r^v\right)$
τ	Cost for transferring a data unit
ρ_m	Cost of processing a unit data in map task
ρ_r	Cost of processing a unit data in reduce task
n_m	Number of map slots
n_r	Number of reduce slots
n_m^{min}	Minimum number of slots for the map task
n	Total number of slots, $n = n_m + n_r$
t_m^0	Start time of the map task
t_r^{max}	Maximum value for the start time of the reduce task
α	Map distribution vector; the EPR strategy is used and, $\alpha_j = 1/u$
ϕ	Filter ratio, the fraction of the input produced as output by the map process