UNIT-5 Cloud Resource Management and Scheduling

Cloud Resource Management and Scheduling

Resource management is a core function of any man-made system. It affects the three basic criteria for the evaluation of a system: performance, functionality, and cost. An inefficient resource management has a direct negative effect on performance and cost and an indirect effect on the functionality of a system. Indeed, some functions provided by the system may become too expensive or may be avoided due to poor performance.

A cloud is a complex system with a very large number of shared resources subject to unpredictable requests and affected by external events it cannot control. Cloud resource management requires com-plex policies and decisions for multi-objective optimization. Cloud resource management is extremely challenging because of the complexity of the system, which makes it impossible to have accurate global state information, and because of the unpredictable interactions with the environment.

The strategies for resource management associated with the three cloud delivery models, *IaaS, PaaS*, and *SaaS*, differ from one another. In all cases the cloud service providers are faced with large, fluctuating loads that challenge the claim of cloud elasticity. In some cases, when a spike can be predicted, the resources can be provisioned in advance, e.g., for Web services subject to seasonal spikes. For an unplanned spike, the situation is slightly more complicated. *Auto Scaling* can be used for unplanned spike loads, provided that (a) there is a pool of resources that can be released or allocated on demand and (b) there is a monitoring system that allows a control loop to decide in real time to reallocate resources. Auto Scaling is supported by *PaaS* services such as Google App Engine.

It has been argued for some time that in a cloud, where changes are frequent and unpredictable, centralized control is unlikely to provide continuous service and performance guarantees. Indeed, centralized control cannot provide adequate solutions to the host of cloud management policies that have to be enforced. Autonomic policies are of great interest due to the scale of the system, the large number of service requests, the large user population, and the unpredictability of the load. The ratio of the mean to the peak resource needs can be very large.

6.1 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. Butler Lampson and Per Brinch Hansen offer solid arguments for this separation in the context of operating system design.

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

- Admission control.
- Capacity allocation.
- Load balancing.
- Energy optimization.
- 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 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 *B*; thus, *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.

As we shall see in this section, often resource management strategies jointly target performance and power consumption. *Dynamic voltage and frequency scaling* (*DVFS*) techniques such as Intel's SpeedStep and AMD's PowerNow lower the voltage and the frequency to decrease power consumption.[2] Motivated initially by the need to save power for mobile devices, these techniques have migrated to virtually all processors, including the ones used for high-performance servers.

DVFS is a power management technique to increase or decrease the operating voltage or frequency of a processor in order to increase the instruction execution rate and, respectively, reduce the amount of heat generated and to conserve power.

The **power consumption** P of a CMOS-based circuit is $P = \alpha \cdot C_{eff} \cdot V^2 \cdot f$, with α = the switching factor, C_{eff} = the effective capacitance, V = the operating voltage, and f = the operating frequency.

As a result of lower voltages and frequencies, the performance of processors decreases, but at a substantially slower rate than the energy consumption. Table 6.1 shows the dependence of the normalized performance and the normalized energy consumption of a typical modern processor on clock rate. As we can see, at 1.8 GHz we save 18% of the energy required for maximum performance, whereas the performance is only 5% lower than the peak performance, achieved at 2.2 GHz. This seems a reasonable energy-performance tradeoff!

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

Virtually all optimal – or near-optimal – mechanisms to address the five classes of policies do not scale up and typically target a single aspect of resource management, e.g., admission control, but ignore energy conservation. Many require complex computations that cannot be done effectively in the time available to respond. The performance models are very complex, analytical solutions are intractable, and the monitoring systems used to gather state information for these models can be too intrusive and unable to provide accurate data. Many techniques are concentrated on system performance in terms of throughput and time in system, but they rarely include energy tradeoffs or QoS guarantees. Some techniques are based on unrealistic assumptions; for example, capacity allocation is viewed as an optimization problem, but under the assumption that servers are protected from overload.

Allocation techniques in computer clouds must be based on a disciplined approach rather than ad hoc methods. The four basic mechanisms for the implementation of resource management policies are:

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. Kalman filters have been used for unrealistically simplified models.

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.

Utility-based. Utility-based approaches require a performance model and a mechanism to correlate user-level performance with cost.

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 placement, whereas those for non-interactive workloads are focused on scheduling. A fair amount of work reported in the literature is devoted to resource management of interactive workloads, some to non-interactive, and only a few, e.g., to heterogeneous workloads, a combination of the two.

6.2 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.

A technique to design self-managing systems based on concepts from control theory is discussed in. 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 $u(i), u(i+1), \ldots, u(n-1)$ to minimize the expression

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

where $(n, x(n))$ 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

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

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.

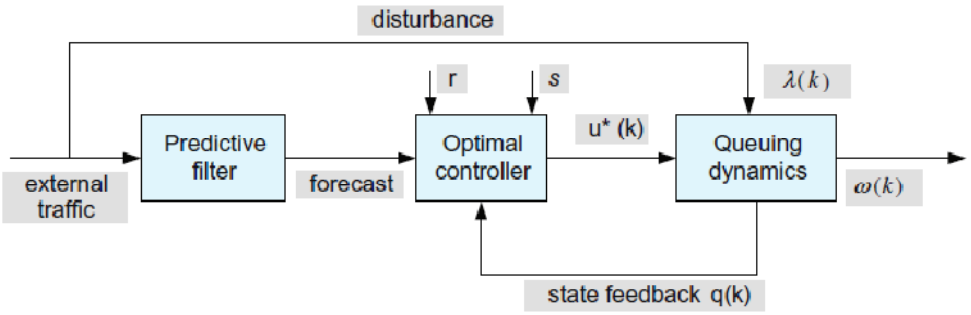


FIGURE 6.1: The structure of an optimal controller described in. The controller uses the feedback regarding the current state as well as the estimation of the future disturbance due to environment to compute the optimal inputs over a finite horizon. The two parameters r and s are the weighting factors of the performance index.

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

$$(x , y, \lambda) = g(x , y) + \lambda \times [h(x , y) - k].$$

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

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

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.

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 $\hat{}(k)$ of the arrival rate (k) at time k.

[3]A boundary value problem has conditions specified at the extremes of the independent variable, whereas an initial value problem has all the conditions specified at the same value of the independent variable in the equation.

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

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

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

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

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

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

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

$$R(u(k)) = 1/2(r \times u(k)^2).$$

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

$$(q(N)) = 1/2(v \times q(n)^2).$$

The performance measure of interest is a cost expressed as

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

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 the above Equation is minimized subject to the following constraints

$$q(k+1) = q(k) + \hat{c}(k) - \frac{u(k)}{\hat{c}(k) \times u_{max}} \times T_s, \quad q(k) \geq 0, \text{ and } u_{min} \leq u(k) \leq u_{max}. \quad (6.11)$$

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

$$1: q(k) > 0, \qquad 2: u(k) \geq u_{min}, \quad 3: u(k) \leq u_{max}, \quad (6.12)$$

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

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

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

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

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

$$\mu 1(k) 0, \mu_1 \quad (k)(-q(k)) = 0, \quad (6.15)$$
$$\mu_2(k) 0, \mu_2(k)(-u(k) + u_{min}) = 0, \quad (6.16)$$
$$\mu_3(k) 0, \mu_3 \quad (k)(u(k) - u_{max}) = 0. \quad (6.17)$$

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.

6.3 Stability of a two-level resource allocation architecture

We can assimilate a server with a closed-loop control system and we can apply control theory principles to resource allocation. In this section we discuss a two-level resource 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.

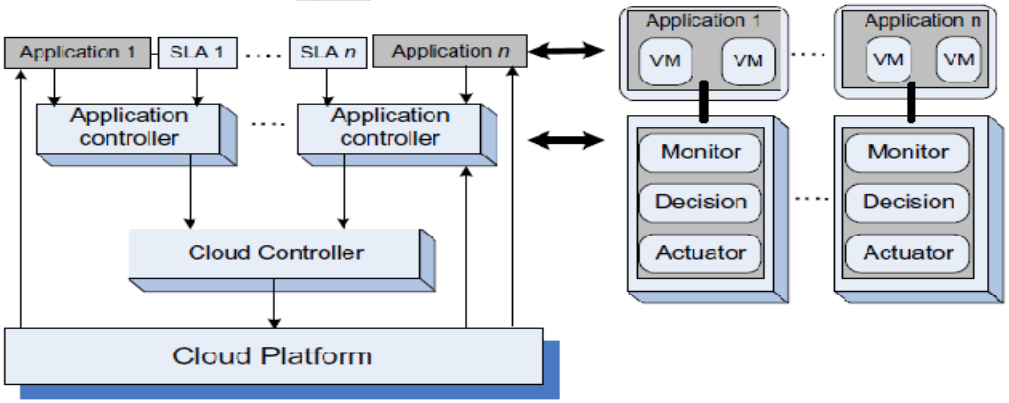


FIGURE 6.2: A two-level control architecture. Application controllers and cloud controllers work in concert.

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. In our context 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.

Two types of policies are used in autonomic systems: (i) threshold-based policies and (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.

Lessons learned from the experiments with two levels of controllers and the two types of policies are discussed in. 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.

6.4 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. 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 outright 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. Once an agreement is in place, user demands must be satisfied; user requests for additional resources within the SLA limits cannot be denied.

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. *Control granularity* refers to the level of detail of the information used to control the system. *Fine control* means that very detailed information about the parameters controlling the system state is used, whereas *coarse control* means that the accuracy of these parameters is traded for the efficiency of implementation.

Proportional Thresholding. Application of these ideas to cloud computing, in particular to the *IaaS* delivery model, and a strategy for resource management called *proportional thresholding* are discussed in . The questions addressed are:

- Is it beneficial to have 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?
- Is it feasible to consider *fine control*? Is *course control* more adequate in a cloud computing environment?
- Are dynamic thresholds based on time averages better than static ones?
- Is it better to have a high and a low threshold, or it is sufficient to define only a high threshold?

The first two questions are related to one another. It seems more appropriate to have two controllers, one with knowledge of the application and one that's aware of the state of the cloud. In this case a coarse control is more adequate for many reasons. As mentioned earlier, the cloud controller can only have a very rough approximation of the cloud state. Moreover, to simplify its resource management policies, the service provider may want to hide some of the information it has. For example, it may not allow a VM to access information available to VMM-level sensors and actuators.

To answer the last two questions, we have to define a measure of "goodness." In the experiments reported in , the parameter measured is the average CPU utilization, and one strategy is better than another if it reduces the number of requests made by the application controllers to add or remove virtual machines to the pool of those available to the application.

Devising a control theoretical approach to address these questions is challenging. The authors of] adopt a pragmatic approach and provide qualitative arguments; they also report simulation results using a synthetic workload for a transaction-oriented application, a Web server.

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.

6.5 Coordination of specialized autonomic performance managers

Can specialized autonomic performance managers cooperate to optimize power consumption and, at the same time, satisfy the requirements of SLAs? This is the question examined by a group from IBM Research in a 2007 paper. The paper reports on actual experiments carried out on a set of blades mounted on a chassis (see Figure 6.3 for the experimental setup). Extending the techniques discussed in this report to a large-scale farm of servers poses significant problems; computational complexity is just one of them.

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 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.

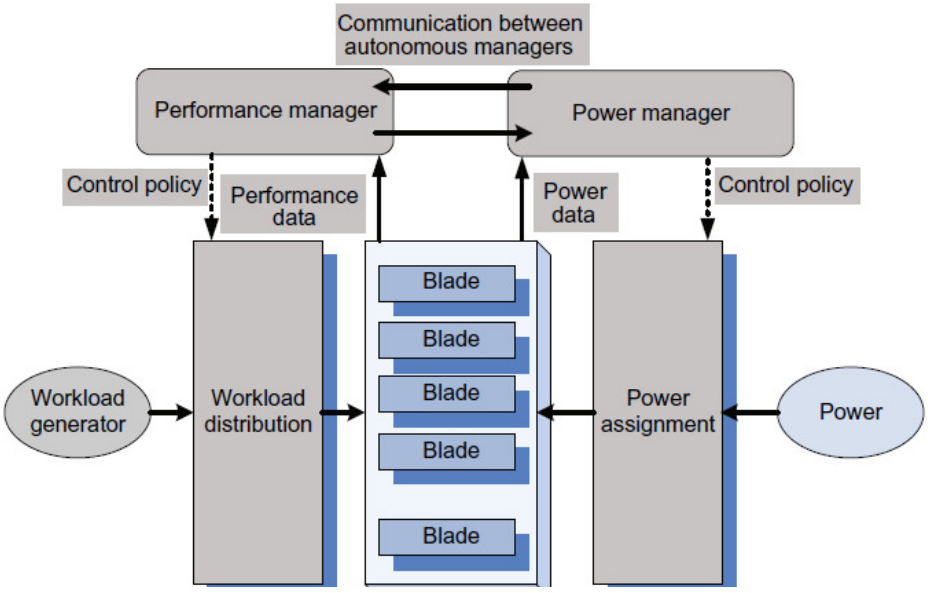


FIGURE 6.3: Autonomous performance and power managers cooperate to ensure SLA prescribed performance and energy optimization. They are fed with performance and power data and implement the performance and power management policies, respectively.

The approach to coordinating power and performance management in [187] 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 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 regard-ing 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.

Intelligent Platform Management Interface (IPMI) is a standardized computer system interface developed by Intel and used by system administrators to manage a computer system and monitor its operation.

- 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_K, the powercap, as in

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

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

$$p_K^{opt}(n_c) = \arg\max_{p_K} U(p_K, n_c). \tag{6.20}$$

The hardware devices used for these experiments were the Goldensbridge blades each with an Intel Xeon processor running at 3 GHz with 1 GB of level 2 cache and 2 GB of DRAM and with hyperthreading enabled. A blade could serve 30 to 40 clients with a response time at or better than a 1,000 msec limit. When p_k is lower than 80 Watts, the processor runs at its lowest frequency, 375 MHz, whereas for p_k at or larger than 110 Watts, the processor runs at its highest frequency, 3 GHz.

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 p_K on n_c 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, p_K , and the number of clients, n_c ; 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 p_k increases from 80 to 110 watts.

- For a given value of the powercap, the consumed power increases rapidly as the load increases.

The machine learning algorithm used for the third type of experiment was based on the Hybrid Reinforcement Learning algorithm described in [349]. In the experiments using the machine learning model, the powercap required to achieve a response time lower than 1,000 msec for a given number of clients was the lowest when = 0.05 and the first utility function given by Eq. (6.18) was used. For example, when n_c = 50, then p_K = 109 Watts when = 0.05, whereas p_K = 120 when = 0.01.

6.6 A utility-based model for cloud-based Web services

A *utility function* relates the "benefits" of an activity or service with the "cost" to provide the service.

For example, the benefit could be revenue and the cost could be the power consumption.

A service-level agreement (SLA) often specifies the rewards as well as the penalties associated with specific performance metrics. Sometimes the quality of services translates into average response time; this is the case of cloud-based Web services when the SLA often explicitly specifies this requirement.

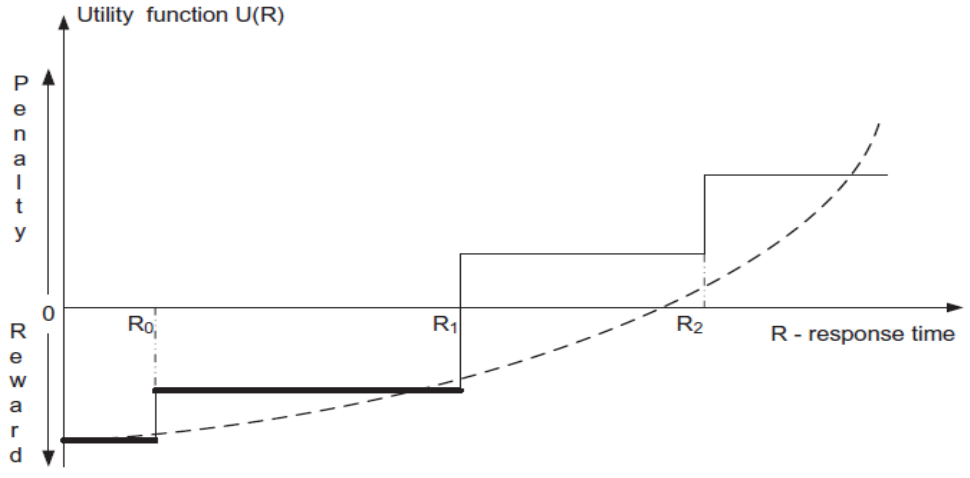


FIGURE 6.4: The utility function *U(R)* is a series of step functions with jumps corresponding to the response time, $R = R_0|R_1|R_2$, when the reward and the penalty levels change according to the SLA. The dotted line shows a quadratic approximation of the utility function.

For example, Figure 6.4 shows the case when the performance metrics is R, the response time. The largest reward can be obtained when R R_0; a slightly lower reward corresponds to $R_0 < R$ R_1. When $R_1 < R$ R_2, instead of gaining a reward, the provider of service pays a small penalty; the penalty increases when $R > R_2$. A utility function, $U(R)$, which captures this behavior, is a sequence of step functions. The utility function is sometimes approximated by a quadratic curve, as we shall see in Section 6.2.

In this section we discuss a utility-based approach for autonomic management. The goal is to max-imize the total profit computed as the difference between the revenue guaranteed by an SLA and the total cost to provide the services. Formulated as an optimization problem, the solution discussed in [9] addresses multiple policies, including QoS. The cloud model for this optimization is quite complex and requires a fair number of parameters.

We assume a cloud providing $|K|$ different classes of service, each class k involving N_k applications. For each class $k \in K$ call v_k the revenue (or the penalty) associated with a response time r_k and assume a linear dependency for this utility function of the form $v_k = v_k^{max}\ 1 - r_k / r_k^{max}$, see Figure 6.5(a). Call $m_k = -v_k^{max} / r_k^{max}$ the slope of the utility function.

The system is modeled as a network of queues with multiqueues for each server and with a delay center that models the think time of the user after the completion of service at one server and the start of processing at the next server [see Figure 6.5(b)]. Upon completion, a class k request either completes with probability $(1 - _{k \in K} \pi_{k,k})$ or returns to the system as a class k request with transition probability $\pi_{k,k}$. Call λ_k the external arrival rate of class k requests and $_k$ the aggregate rate for class k, where $k = \lambda k + _{k \in K}\ k\ \pi k,k$.

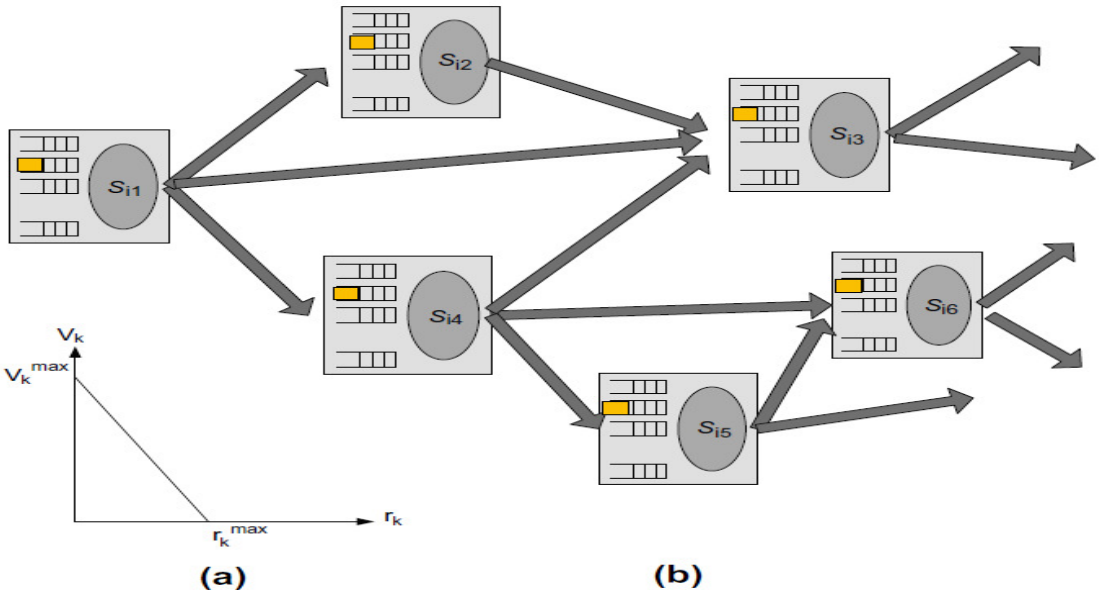


FIGURE 6.5: The utility function, v_k the revenue (or the penalty) associated with a response time r_k for a request of class $k \in K$. The slope of the utility function is $m_k = -v_k^{max} / r_k^{max}$. (b) A network of multiqueues. At each server S_i there are $|K|$ queues for each one of the $k \in K$ classes of requests. A tier consists of all requests of class $k \in K$ at all servers $S_{ij} \in I$, 1<=j<=6.

Typically, CPU and memory are considered representative for resource allocation; for simplicity we assume a single CPU that runs at a discrete set of clock frequencies and a discrete set of supply voltages according to a Dynamic Voltage and Frequency

frequency. The scheduling of a server is work-conserving[6] and is modeled as a Generalized Processor Sharing (GPS) scheduling. Analytical models [4,280] are too complex for large systems.

The optimization problem formulated in involves five terms: A and B reflect revenues; C the cost of servers in a low-power, stand-by mode; D the cost of active servers, given their operating frequency;

, the cost of switching servers from low-power, stand-by mode to active state, and F , the cost of migrating VMs from one server to another. There are nine constraints $_1$, $_2$, . . . , $_9$ for this mixed-integer, nonlinear programming problem. The decision variables for this optimization problem are listed in Table 6.2 and the parameters used are shown in Table 6.3.

Table 6.2 Decision variables for the optimization problem.

Name	Description
x_i	$x_i = 1$ if server $i \in I$ is running, $x_i = 0$ otherwise
$y_{I,h}$	$y_{I,h} = 1$ if server i is running at frequency h, $y_{I,h} = 0$ otherwise
$z_{i,k,j}$	$z_{i,k,j} = 1$ if application tier j of a class k request runs on server i, $z_{i,k,j} = 0$ otherwise
$w_{i,k}$	$w_{i,k} = 1$ if at least one class k request is assigned to server i, $w_{i,k} = 0$ otherwise
$\lambda_{i,k,j}$	Rate of execution of applications tier j of class k requests on server i
$\phi_{i,k,j}$	Fraction of capacity of server i assigned to tier j of class k requests

Table 6.3 The parameters used for the A, B, C, D, E, and F terms and the constraints Γ_j of the optimization problem.

Name	Description
I	The set of servers
K	The set of classes
Λ_k	The aggregate rate for class $k \in K$, $\Lambda_k = \lambda_k + \sum_{k' \in K} \Lambda_{k'} \pi_{k,k'}$
a_i	The availability of server $i \in I$
A_k	Minimum level of availability for request class $k \in K$ specified by the SLA
m_k	The slope of the utility function for a class $k \in K$ application
N_k	Number of applications in class $k \in K$
H_i	The range of frequencies of server $i \in I$
$C_{i,h}$	Capacity of server $i \in I$ running at frequency $h \in H_i$
$c_{i,h}$	Cost for server $i \in I$ running at frequency $h \in H_i$
$\bar{c}_i$	Average cost of running server i
$\mu_{k,j}$	Maximum service rate for a unit capacity server for tier j of a class k request
cm	The cost of moving a virtual machine from one server to another
cs_i	The cost for switching server i from the stand-by mode to an active state
$\overline{RAM}_{k,j}$	The amount of main memory for tier j of class k request
$\overline{RAM}_i$	The amount of memory available on server i

The expression to be maximized is:

$$(A + B) - (C + D + E + F) \tag{6.21}$$

with

$$A = \max \sum_{k \in K} \left(-m_k \sum_{i \in I, j \in N_k} \frac{\lambda_{i,k,j}}{\sum_{h \in H_i} \left(C_{i,h} \times y_{i,h} \right) \mu_{k,j} \times \phi_{i,k,j} - \lambda_{i,k,j}} \right),$$

$$B = \sum_{k \in K} u_k \times \Lambda_k, \tag{6.22}$$

$$C = \sum_{i \in I} \bar{c}_i, \quad D = \sum_{i \in I, h \in H_i} c_{i,h} \times y_{i,h}, \quad E = \sum_{i \in I} cs_i \max(0, x_i - \bar{x}_i), \tag{6.23}$$

and

$$F = \sum_{i \in I, k \in K, j \in N_j} cm \max(0, z_{i,j,k} - \bar{z}_{i,j,k}). \tag{6.24}$$

The nine constraints are:

(Γ_1) $\sum_{i \in I} \lambda_{i,k,j} = \Lambda_k$, $\forall k \in K$, $j \in N_k \rightarrow$ the traffic assigned to all servers for class k requests equals the predicted load for the class.

(Γ_2) $\sum_{k \in K, j \in N_k} \phi_{i,k,j} \leqslant 1 \; \forall i \subset I \Rightarrow$ server i cannot be allocated an workload more than its capacity.

(Γ_3) $\sum_{h \subset H_i} y_{i,h} = x_i$, $\forall i \in I \Rightarrow$ if server $i \in I$ is active it runs at one frequency in the set H_i, and only one $y_{i,h}$ is nonzero.

(Γ_4) $z_{i,k,j} \leqslant x_i$, $\forall i \in I$, $k \in K$, $j \in N_k \Rightarrow$ requests can only be assigned to active servers.

(Γ_5) $\lambda_{i,k,j} \leqslant \Lambda_k \times z_{i,k,j}$, $\forall i \in I$, $k \in K$, $j \in N_k \Rightarrow$ requests may run on server $i \in I$ only if the corresponding application tier has been assigned to server i.

(Γ_6) $\lambda_{i,k,j} \leqslant \left(\sum_{h \in H_i} C_{i,h} \times y_{i,h}\right) \mu_{k,j} \times \phi_{i,k,j}$, $\forall i \in I$, $k \in K$, $j \in N_k \rightarrow$ resources cannot be saturated.

(Γ_7) $RAM_{k,j} \times z_{i,k,j} \leqslant \overline{RAM}_i$, $\forall i \in I$, $k \in K \Rightarrow$ the memory on server i is sufficient to support all applications running on it.

(Γ_8) $\Pi_{j=1}^{N_k} \left(1 - \Pi_{i=1}^{M} (1 - a_i^{w_{i,k}})\right) \geqslant A_k$, $\forall k \in K \Rightarrow$ the availability of all servers assigned to class k request should be at least equal to the minimum required by the SLA.

(Γ_9) $\sum_{j=1}^{N_k} z_{i,k,j} \geqslant N_k \times w_{i,k}$, $\forall i \in I$, $k \in K$
$\lambda_{i,j,k}, \phi_{i,j,k} \geqslant 0$, $\forall i \in I$, $k \in K$, $j \in N_k$
$x_i, y_{i,h}, z_{i,k,j}, w_{i,k} \in \{0, 1\}$, $\forall i \in I$, $k \in K$, $j \in N_k \Rightarrow$ constraints and relations among decision variables.

Clearly, this approach is not scalable to clouds with a very large number of servers. Moreover, the large number of decision variables and parameters of the model make this approach infeasible for a realistic cloud computing resource management strategy.

6.7 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 *pack-ages*, 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* . The algorithm discussed in this chap-ter and introduced in is called the *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.

The final auction prices for individual resources are given by the vector $p = (p^1, p^2, \ldots, p^R)$ and the amounts of resources allocated to user u are $x_u = (x_u^1, x_u^2, \ldots, x_u^R)$. 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 $[\min_{q \in Q_u} (q^T p)]$ 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 π_u 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

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

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.

$x_u \in \{0 \cup \mathcal{Q}_u\}, \forall u$	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.

The ASCA Combinatorial Auction Algorithm. Informally, in the ASCA algorithm the par-ticipants 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*

$$z(t) = \sum_u x_u(t) \qquad (6.26)$$

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. Note that the algorithm satisfies conditions 1 through 6; from Table 6.3 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.

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 Q_u set given the current price

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

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

```
1: set t = 0, p(0) = p̄
2: loop
3:   collect bids x_u(t) = Q_u(p(t)), ∀u
4:   calculate excess demand z(t) = Σ_u x_u(t)
5:   if z(t) <0 then
6:     break
7:     else
```

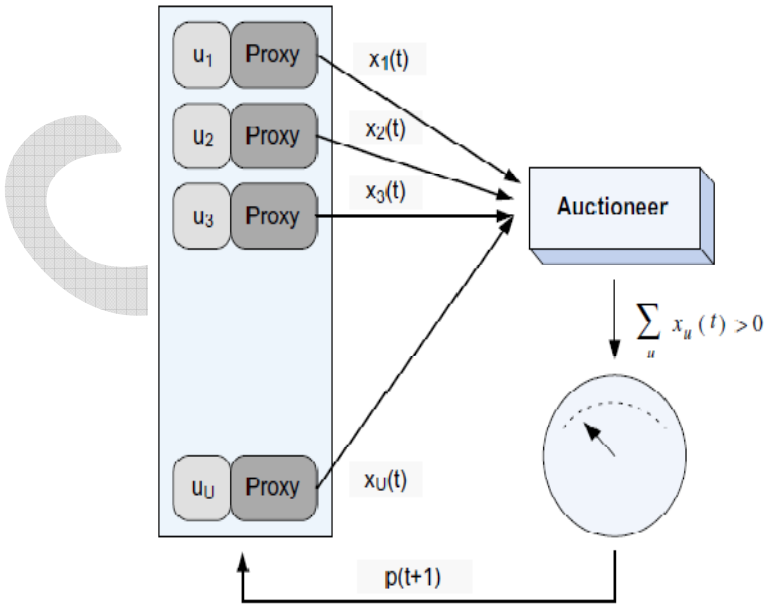


FIGURE 6.6: The schematics of the ASCA algorithm. To allow for a single round, auction users are represented by proxies that place the bids $x_u(t)$. The auctioneer determines whether there is an excess demand and, in that case, raises the price of resources for which the demand exceeds the supply and requests new bids.

```
8:      update prices p(t + 1) = p(t) + g(x(t), p(t))
9:      t ← 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.

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 authors of have implemented the algorithm 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.

6.8 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.

In addition to the requirement to meet its design objectives, a scheduling algorithm should be efficient, fair, and starvation-free. The objectives of a scheduler for a batch system are to maximize the throughput (the number of jobs completed in one unit of time, e.g., in one hour) and to minimize the turnaround time (the time between job submission and its completion). For a real-time system the objectives are to meet the deadlines and to be predictable. 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 *nonpreemptive*.

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 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, $B_i \quad b_i$.

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, $_a\ (t_1, t_2)$ and $_b\ (t_1, t_2)$, respectively, minimize the expression

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

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.

where w_a and w_b 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[7] 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 is discussed in [56], and two new algorithms for integrated scheduling, resource allocation/dispatching (RAD) and rate-based earliest deadline (RBED) are proposed.

Next we discuss several algorithms of special interest for computer clouds. These algorithms illustrate the evolution in thinking regarding the fairness of scheduling and the need to accommodate multi-objective scheduling – in particular, scheduling for multimedia applications.

6.9 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 the outgoing links, as well as limited CPU cycles.

A switch must handle multiple flows and pairs of source-destination endpoints of the traffic. Thus, a scheduling algorithm has to manage several quantities at the same time: the *bandwidth*, the amount of data each flow is allowed to transport; the *timing* when the packets of individual flows are transmitted; and the *buffer space* allocated to each flow. A first strategy to avoid network congestion is to use a FCFS scheduling algorithm. The advantage of the FCFS algorithm is a simple management of the three quantities: bandwidth, timing, and buffer space. Nevertheless, the FCFS algorithm does not guarantee fairness; greedy flow sources can transmit at a higher rate and benefit from a larger share of the bandwidth.

To address this problem, 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).

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

$$R(t) \geq F_i^a \quad \text{when} \quad i = \max\left(j \mid t_j^a \leq t\right). \tag{6.29}$$

The authors of] use for packet-by-packet transmission time the following nonpreemptive schedul-ing 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.

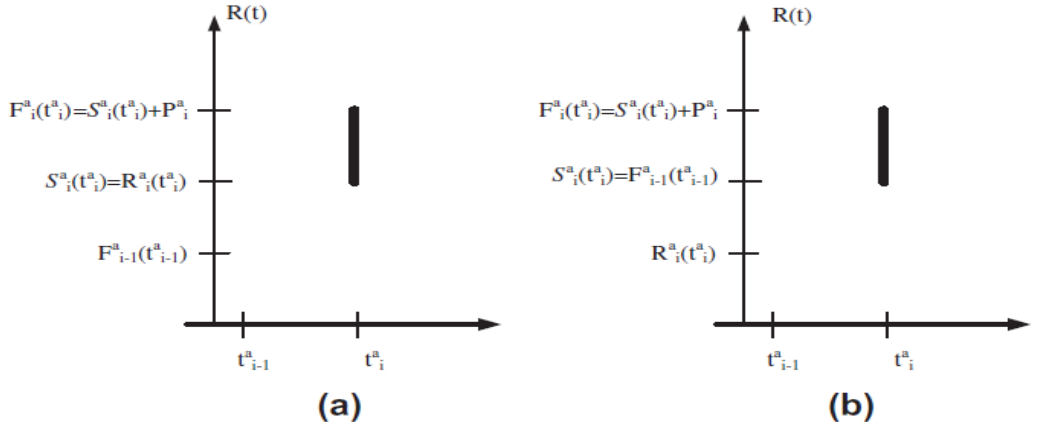


FIGURE 6.8

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 $F_{i-1}^a \geq R(t_i^a)$.

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

$$B_i^a = P_i^a + \max\left(F_i^a{}_{-1},\ R\left(t_i^a\right) - \delta\right), \tag{6.30}$$

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.

6.10 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, $1 \quad j \quad n$, the weight of the n children of node i ; see the example in Figure 6.9.

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.

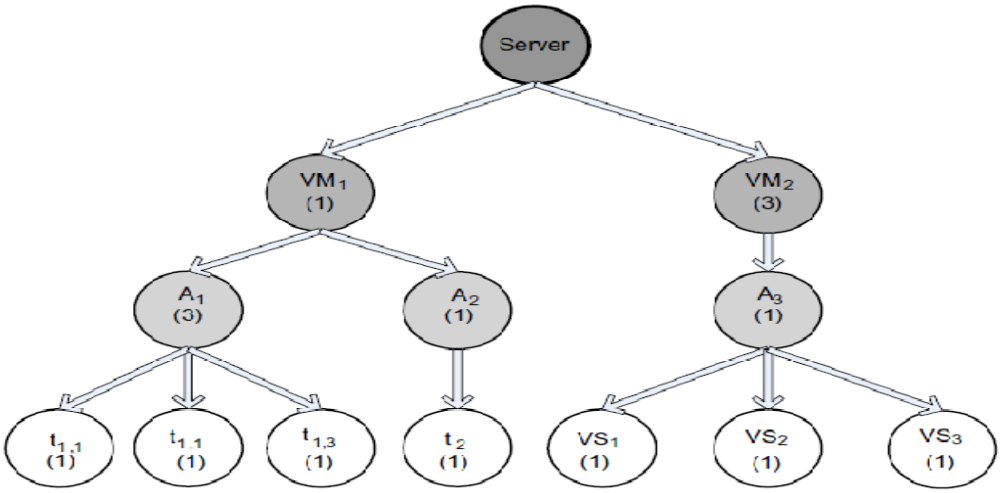


FIGURE 6.9: The SFQ tree for scheduling when two virtual machines, VM_1 and VM_2, run on a powerful server. VM_1 runs two best-effort applications A_1, with three threads $t_{1,1}$, $t_{1,2}$, and $t_{1,3}$, and A_2 with a single thread, t_2. VM_2 runs a video-streaming application, A_3, with three threads vs_1, vs_2, and vs_3. The weights of virtual machines, applications, and individual threads are shown in parenthesis.

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

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

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,

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

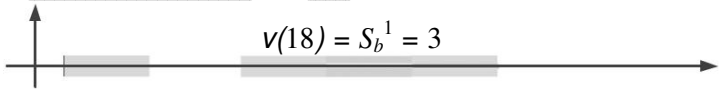
and

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

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,

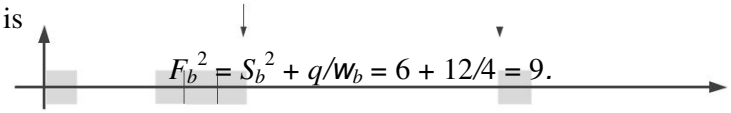


$$v(18) = S_b^1 = 3 \tag{6.41}$$

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,

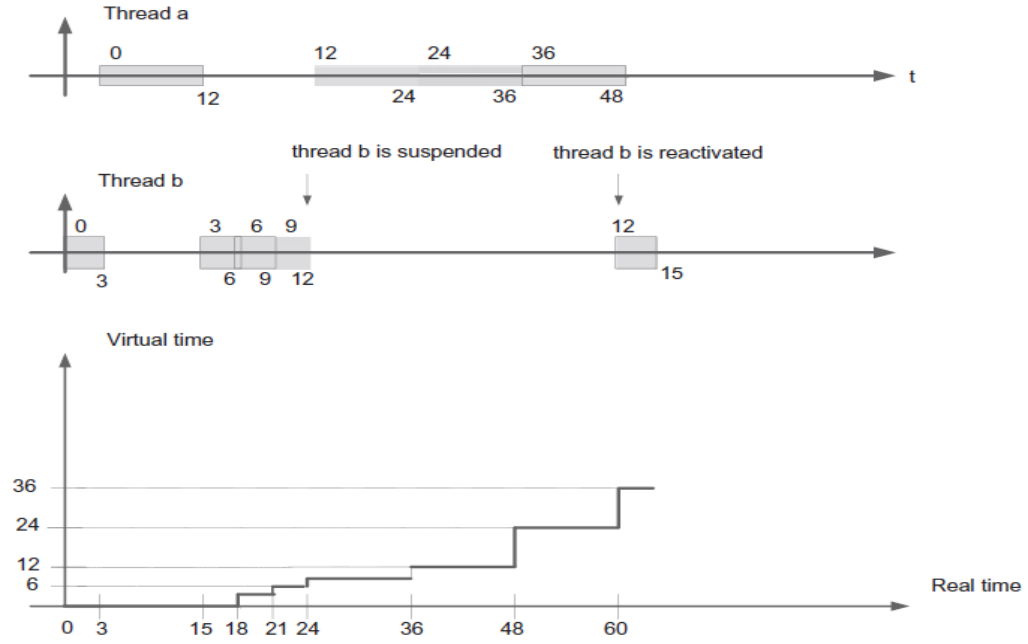


FIGURE 6.10: Top, the virtual start-up time $S_a(t)$ and $S_b(t)$ and the virtual finish time $F_a(t)$ and $F_b(t)$ function of the real time for each activation of threads *a* and *b*, respectively, are marked at the top and bottom of the box representing a running thread. The virtual time of the scheduler $v(t)$ 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

$$F_b^3 = S_b^3 + q/w_b = 9 + 12/4 = 12. \quad (6.46)$$

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

$$v(24) = S_b^3 = 9 \quad (6.47)$$

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

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

$$F_a^1 = S_a^1 + q/w_a = 12 + 12/1 = 24. \quad (6.49)$$

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

$$v(36) = S_a^1 = 12 \quad (6.50)$$

and

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

Then,

$$F_a^2 = S_a^2 + q/w_a = 24 + 12/1 = 36. \quad (6.52)$$

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

$$v(48) = S_a^2 = 24 \quad (6.53)$$

and

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

Then,

$$F_a^3 = S_a^3 + q/w_a = 36 + 12/1 = 48. \quad (6.55)$$

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

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

Several properties of the SFQ algorithm are proved in . The algorithm allocates CPU fairly when the available bandwidth varies in time and provides throughput as well as delay guarantees. The algorithm schedules the threads in the order of their virtual start-up time, the shortest one first; the length of the time quantum is not required when a thread is scheduled but only after the thread has finished its current allocation. The authors of report that the overhead of the SFQ algorithms is comparable to that of the Solaris scheduling algorithm.

6.11 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 [107]. 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*, E_i , an *actual virtual time*, A_i , and a *virtual time warp*, W_i . 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, E_i, 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 *warpBack* is set. In this case a latency-sensitive thread gains dispatching preference as

$$E_i \leftarrow \begin{cases} A_i & \text{if } warpBack = OFF \\ A_i - W_i & \text{if } warpBack = ON. \end{cases} \tag{6.59}$$

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 com-peting with other threads, measured in multiples of *mcu*. Typical values for the two quantities are *mcu* = 100 μ*sec* 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 τ_i becomes runnable after sleeping, its actual virtual time is updated as follows:

$$A_i \leftarrow \max[A_i, SVT]. \tag{6.60}$$

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.

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 used to advance its actual virtual time upon completion of a run:

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

Given two threads a and b,

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

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

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

Example 1. The following example illustrates the application of the BVT algorithm for scheduling two threads a and b of best-effort applications. The first thread has a weight twice that of the second, $w_a = 2w_b$; when $k_a = 180$ and $k_b = 90$, then = 90.

We consider periods of real-time allocation of $C = 9$ *mcu*. The two threads a and b are allowed to run for $2C/3 = 6$ *mcu* and $C/3 = 3$ *mcu*, respectively.

Threads a and b are activated at times

$$a : 0, 5, 5 + 9 = 14, 14 + 9 = 23, 23 + 9 = 32, 32 + 9 = 41, \ldots \tag{6.64}$$

$$b : 2, 2 + 9 = 11, 11 + 9 = 20, 20 + 9 = 29, 29 + 9 = 38, \ldots$$

The context switches occur at real times:

$$2, 5, 11, 14, 20, 23, 29, 32, 38, 41, \ldots \tag{6.65}$$

The time is expressed in units of *mcu*. The initial run is a shorter one, consists of only 3 *mcu*; a context switch occurs when a, which runs first, exceeds b by 2 *mcu*.

Table 6.5 shows the effective virtual time of the two threads at the time of each context switch. At that moment, its actual virtual time is incremented by an amount equal to if the thread was allowed to run for its time allocation. The scheduler compares the effective virtual time of the threads and first runs the one with the minimum effective virtual time.

Table 6.5 The real time of the context switch and the effective virtual time $E_a(t)$ and $E_b(t)$ at the time of a context switch. There is no time warp, so the effective virtual time is the same as the actual virtual time. At time $t = 0$, $E_a(0) = E_b(0) = 0$ and we choose thread a to run.

Context Switch	Real Time	Running Thread	Effective Virtual Time of the Running Thread
1	$t = 2$	a	$E_a(2) = A_a(2) = A_a(0) + \Delta/3 = 30$ b runs next as $E_b(2) = 0 < E_a(2) = 30$
2	$t = 5$	b	$E_b(5) = A_b(5) = A_b(0) + \Delta = 90$ a runs next as $E_a(5) = 30 < E_b(5) = 90$
3	$t = 11$	a	$E_a(11) = A_a(11) = A_a(2) + \Delta = 120$ b runs next as $E_b(11) = 90 < E_a(11) = 120$
4	$t = 14$	b	$E_b(14) = A_b(14) = A_b(5) + \Delta = 180$ a runs next as $E_a(14) = 120 < E_b(14) = 180$
5	$t = 20$	a	$E_a(20) = A_a(20) = A_a(11) + \Delta = 210$ b runs next as $E_b(20) = 180 < E_a(20) = 210$
6	$t = 23$	b	$E_b(23) = A_b(23) = A_b(14) + \Delta = 270$ a runs next as $E_a(23) = 210 < E_b(23) = 270$
7	$t = 29$	a	$E_a(29) = A_a(29) = A_a(20) + \Delta = 300$ b runs next as $E_b(29) = 270 < E_a(29) = 300$
8	$t = 32$	b	$E_b(32) = A_b(32) = A_b(23) + \Delta = 360$ a runs next as $E_a(32) = 300 < E_b(32) = 360$.
9	$t = 38$	a	$E_a(38) = A_a(38) = A_a(29) + \Delta = 390$ b runs next as $E_b(11) = 360 < E_a(11) = 390$
10	$t = 41$	b	$E_b(41) = A_b(41) = A_b(32) + \Delta = 450$ a runs next as $E_a(41) = 390 < E_b(41) = 450$

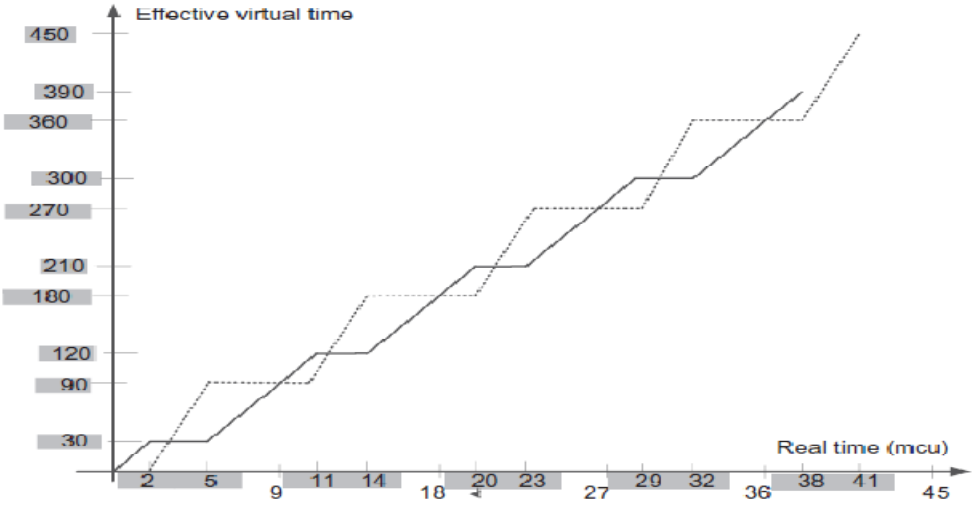


Figure 6.11 displays the effective virtual time and the real time of threads a and b. When a thread is running, its effective virtual time increases as the real time increases; a running thread appears as a diagonal line. When a thread is runnable but not running, its effective virtual time is constant. A runnable period is displayed as a horizontal line. We see that the two threads are allocated equal amounts of virtual time, but thread a, with a larger weight, consumes its real time more slowly.

Example 1, the effective virtual time and the real time of threads a (solid line) and b (dotted line) with weights $w_a = 2w_b$ when the actual virtual time is incremented in steps of = 90 *mcu*. The real time the two threads are allowed to use the CPU is proportional to their weights. The virtual times are equal, but thread a consumes it more slowly. There is no time warp. The threads are dispatched based on their actual virtual time.

Example 2. Next we consider the previous example, but this time there is an additional thread, c, with real-time constraints. Thread c wakes up at time $t = 9$ and then periodically at times $t = 18, 27, 36, \ldots$ for 3 units of time.

Table 6.6 summarizes the evolution of the system when the real-time application thread c competes with the two best-effort threads a and b. Context switches occur now at real times

$t = 2, 5, 9, 12, 14, 18, 21, 23, 27, 30, 32, 36, 39, 41, \ldots$ (6.66)

The context switches at times

$t = 9, 18, 27, 36, \ldots$ (6.67)

are triggered by the waking up of thread c, which preempts the currently running thread. At $t = 9$ the time warp $W_c = -60$ gives priority to thread c. Indeed,

$$E_c(9) = A_c(9) - W_c = 0 - 60 = -60 \tag{6.68}$$

compared with $E_a(9) = 90$ and $E_b(9) = 90$. The same conditions occur every time the real-time thread wakes up. The best-effort application threads have the same effective virtual time when the real-time application thread finishes and the scheduler chooses b to be dispatched first. Note that the ratio of real times used by a and b is the same, as $w_a = 2w_b$.

Table 6.6 A real-time thread c with a time warp $W_c = -60$ is waking up periodically at times $t = 18, 27, 36, \ldots$ for 3 units of time and is competing with the two best effort threads a and b. The real time and the effective virtual time of the three threads of each context switch are shown.

Context Switch	Real Time	Running Thread	Effective Virtual Time of the Running Thread
1	$t-2$	a	$E_a(2) - A_a(2) - A_a(0) + \Delta/3 - 0 + 90/3 - 30$
2	$t-5$	b	$E_b^1 - A_b^1 - A_b^0 + \Delta - 0 + 90 - 90 \rightarrow$ *a runs next*
3	$t-9$	a	c wakes up $E_a^1 = A_a^1 + 2\Delta/3 = 30 + (-60) = 90$ $[E_a(9), E_b(9), E_c(9)] = (90, 90, -60) \Rightarrow$ *c runs next*
4	$t=12$	c	$SVT(12) = \min(90, 90)$ $E_c^s(12) = SVT(12) + W_c = 90 + (-60) = 30$ $E_c(12) - E_c^s(12) + \Delta/3 - 30 + 30 - 60 \rightarrow$ *b runs next*
5	$t=14$	b	$E_b^2 = A_b^2 = A_b^1 + 2\Delta/3 = 90 + 60 = 150 \Rightarrow$ *a runs next*
6	$t=18$	a	c wakes up $E_a^3 = A_a^3 = A_a^2 + 2\Delta/3 = 90 + 60 = 150$ $[E_a(18), E_b(18), E_c(18)] = (150, 150, 60) \Rightarrow$ *c runs next*
7	$t=21$	c	$SVT = \min(150, 150)$ $E_c^s(21) - SVT + W_c - 150 + (-60) - 90$ $E_c(21) = E_c^s(21) + \Delta/3 = 90 + 30 = 120 \Rightarrow$ *b runs next*
8	$t=23$	b	$E_b^3 = A_b^3 = A_b^2 + 2\Delta/3 = 150 + 60 = 210 \Rightarrow$ *a runs next*
9	$t-27$	a	c wakes up $E_a^4 - A_a^4 - A_a^3 + 2\Delta/3 - 150 + 60 - 210$ $[E_a(27), E_b(27), E_c(27)] = (210, 210, 120) \Rightarrow$ *c runs next*
10	$t=30$	c	$SVT = \min(210, 210)$ $E_c^s(30) = SVT + W_c = 210 + (-60) = 150$ $E_c(30) = E_c^s(30) + \Delta/3 = 150 + 30 = 180 \Rightarrow$ *b runs next*
11	$t=32$	b	$E_b^4 = A_b^4 = A_b^3 + 2\Delta/3 = 210 + 60 = 270 \Rightarrow$ *a runs next*
10	$t=36$	a	c wakes up $E_a^5 - A_a^5 - A_a^4 + 2\Delta/3 - 210 + 60 - 270$ $[E_a(36), E_b(36), E_c(36)] - (270, 270, 180) \rightarrow$ *c runs next*
12	$t=39$	c	$SVT = \min(270, 270)$ $E_c^s(39) = SVT + W_c = 270 + (-60) = 210$ $E_c(39) - E_c^s(39) + \Delta/3 - 210 + 30 - 240 \rightarrow$ *b runs next*
13	$t-41$	b	$E_b^5 - A_b^5 - A_b^4 + 2\Delta/3 - 270 + 60 - 330 \rightarrow$ *a runs next*

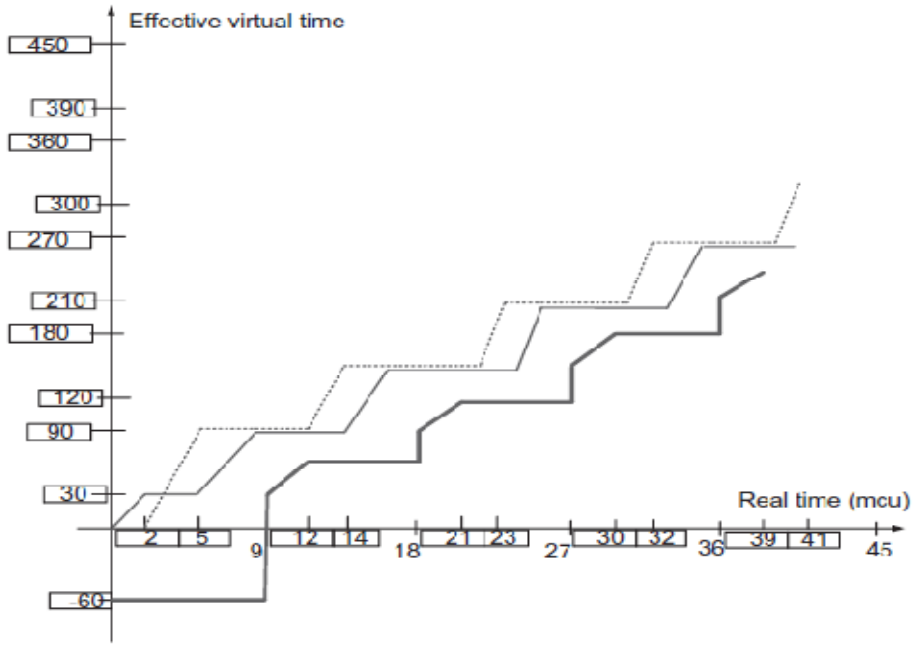


FIGURE 6.12: Example 2, the effective virtual time and the real time of threads *a* (thin solid line), *b* (dotted line), and *c*, with real-time constraints (thick solid line). *c* wakes up periodically at times $t = 9, 18, 27, 36, \ldots$, is active for 3 units of time, and has a time warp of 60 *mcu*.

Figure 6.12 shows the effective virtual times for the three threads *a, b*, and *c*. Every time thread *c* wakes up, it preempts the current running thread and is immediately scheduled to run.

6.12 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 (A_i, σ_i, D_i), where A_i is the arrival time, $\sigma_i > 0$ is the data size of the task, and D_i is the *relative deadline*. Instances of a *periodic task*, τ_i^q, with period q are identical, $\tau_i^q \equiv \tau_i^q$, and arrive at times $A_0, A_1, \ldots A_i, \ldots$, with $A_{i+1} - A_i = q$. The deadlines satisfy the constraint $D_i \leq A_{i+1}$ and generally the data size is the same, $\sigma_i = \sigma$. The individual instances of *aperiodic tasks*, τ_i, are different. Their arrival times A_i are generally uncorrelated, and the amount of data σ_i is different for different instances. The *absolute deadline* for the aperiodic task τ_i is $(A_i + D_i)$. We distinguish *hard deadlines* from *soft deadlines*. In the first case, if the task is not completed by the deadline, other tasks that depend 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 if the deadline is expressed in hours, or hours if the deadlines is expressed in days. The scheduling of tasks on a cloud is generally subject to soft deadlines, though occasionally applications with hard deadlines may be encountered.

System Model. In our discussion we consider only aperiodic tasks with arbitrarily divisible workloads. The application runs on a partition of a cloud, a virtual cloud with a *head node* called S_0 and n *worker nodes* $S_1, S_2, \ldots, S_n$. 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 arrival.
- Earliest deadline first (EDF). The task with the earliest deadline is scheduled first.
- Maximum workload derivative first (MWF).

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

$$DCi\ (_n mi\ n\ ) = W_i\ _{ni} mi\ n + 1\ - W_i\ _{ni} mi\ n\ , \qquad (6.69)$$

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

1. The tasks are scheduled in the order of their derivatives, the one with the highest derivative DC_i first.
2. The number n of nodes assigned to the application is kept to a minimum, n_i^{min} .

We discuss two workload partitioning and task mappings to worker nodes, optimal and the equal partitioning.

Optimal Partitioning Rule (OPR). The optimality in OPR refers to the execution time; in this case, the workload is partitioned to ensure the earliest possible completion time, and all tasks are required to complete at the same time. EPR, as the name suggests, means that the workload is partitioned in equal segments. In our discussion we use the derivations and some of the notations in; these notations are summarized in Table 6.7.

Table 6.7 The parameters used for scheduling with deadlines.

Name	Description
Π_i	The aperiodic tasks with arbitrary divisible load of an application $\mathcal{A}$
A_i	Arrival time of task Π_i
D_i	The relative deadline of task Π_i
σ_i	The workload allocated to task Π_i
S_0	Head node of the virtual cloud allocated to $\mathcal{A}$
S_i	Worker nodes $1 \leqslant i \leqslant n$ of the virtual cloud allocated to $\mathcal{A}$
σ	Total workload for application $\mathcal{A}$
n	Number of nodes of the virtual cloud allocated to application $\mathcal{A}$
n^{min}	Minimum number of nodes of the virtual cloud allocated to application $\mathcal{A}$
$\mathcal{E}(n, \sigma)$	The execution time required by n worker nodes to process the workload σ
τ	Cost of transferring a unit of workload from the head node S_0 to worker S_i
ρ	Cost of processing a unit of workload
α	The load distribution vector $\alpha = (\alpha_1, \alpha_2, \ldots, \alpha_n)$
$\alpha_i \times \sigma$	The fraction of the workload allocated to worker node S_i
Γ_i	Time to transfer the data to worker S_i, $\Gamma_i = \alpha_i \times \sigma \times \tau$, $1 \leqslant i \leqslant n$
Δ_i	Time the worker S_i needs to process a unit of data, $\Delta_i = \alpha_i \times \sigma \times \rho$, $1 \leqslant i \leqslant n$
t_0	Start time of the application $\mathcal{A}$
A	Arrival time of the application $\mathcal{A}$
D	Deadline of application $\mathcal{A}$
$C(n)$	Completion time of application $\mathcal{A}$

The timing diagram in Figure 6.13 allows us to determine the execution time $\mathcal{E}(n, \sigma)$ for the OPR as

$$\begin{aligned}\mathcal{E}(n, \sigma) &= \Gamma_1 + \Delta_1 \\ &= \Gamma_1 + \Gamma_2 + \Delta_2 \\ &= \Gamma_1 + \Gamma_2 + \Gamma_3 + \Delta_3 \\ &\vdots \\ &= \Gamma_1 + \Gamma_2 + \Gamma_3 + \cdots + \Gamma_n + \Delta_n.\end{aligned} \tag{6.70}$$

We substitute the expressions of Γ_i, Δ_i, $1 \leqslant i \leqslant n$, and rewrite these equations as

$$\begin{aligned}\mathcal{E}(n, \sigma) &= \alpha_1 \times \sigma \times \tau + \alpha_1 \times \sigma \times \rho \\ &= \alpha_1 \times \sigma \times \tau + \alpha_2 \times \sigma \times \tau + \alpha_2 \times \sigma \times \rho \\ &= \alpha_1 \times \sigma \times \tau + \alpha_2 \times \sigma \times \tau + \alpha_3 \times \sigma \times \tau + \alpha_3 \times \sigma \times \rho \\ &= \vdots \\ &= \alpha_1 \times \sigma \times \tau + \alpha_2 \times \sigma \times \tau + \alpha_3 \times \sigma \times \tau + \cdots + \alpha_n \times \sigma \times \tau + \alpha_n \times \sigma \times \rho.\end{aligned} \tag{6.71}$$

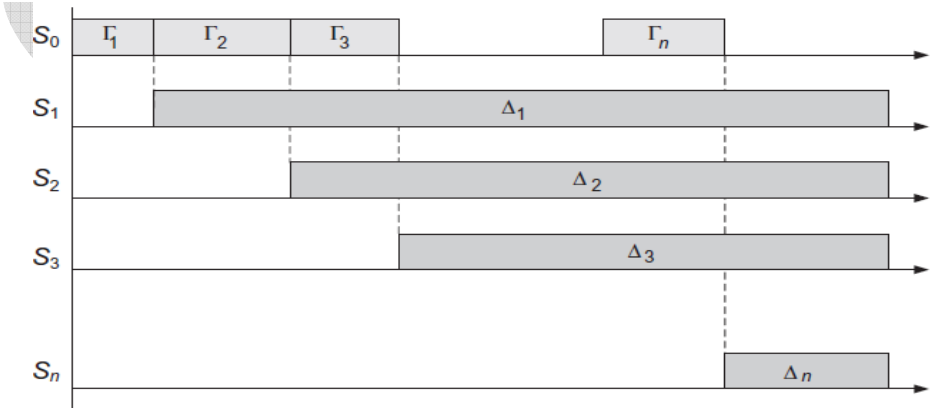


FIGURE 6.13: The timing diagram for the optimal partitioning rule. The algorithm requires worker nodes to complete execution at the same time. The head node, S_0, distributes sequentially the data to individual worker nodes. The communication time is $_i = \alpha_i \times \sigma \times \tau$, 1 i n. Worker node S_i starts processing the data as soon as the transfer is complete. The processing time is $_i = \alpha_i \times \sigma \times \rho$, 1 i n.

From the first two equations we find the relation between α_1 and α_2 as

$$\alpha_1 = \frac{\alpha_2}{\beta} \text{ with } \beta = \frac{\rho}{\tau + \rho}, \ 0 < \beta < 1. \tag{6.72}$$

This implies that $\alpha_2 = \beta \times \alpha_1$. It is easy to see that in the general case

$$\alpha_i = \beta \times \alpha_{i-1} = \beta^{i-1} \times \alpha_1. \tag{6.73}$$

But α_i are the components of the load distribution vector; thus,

$$\sum_{i=1}^{n} \alpha_i = 1. \tag{6.74}$$

Next, we substitute the values of α_i and obtain the expression for α_1:

$$\alpha_1 + \beta \times \alpha_1 + \beta^2 \times \alpha_1 + \beta^3 \times \alpha_1 \ldots \beta^{n-1} \times \alpha_1 = 1 \text{ or } \alpha_1 = \frac{1-\beta}{1-\beta^n}. \tag{6.75}$$

We have now determined the load distribution vector and we can now determine the execution time as

$$E(n, \sigma) = \alpha_1 \times \sigma \times \tau + \alpha_1 \times \sigma \times \rho = \frac{1-\beta}{1-\beta^n} \sigma (\tau + \rho). \tag{6.76}$$

Call $C^A(n)$ the completion time of an application $A = (A, \sigma, D)$, which starts processing at time t_0 and runs on n worker nodes; then

$$C^A(n) = t_0 + E(n, \sigma) = t_0 + \frac{1-\beta}{1-\beta^n} \sigma (\tau + \rho). \tag{6.77}$$

The application meets its deadline if and only if

$$C^A(n) \leq A + D, \tag{6.78}$$

or

$$t_0 + E(n, \sigma) = t_0 + \frac{1-\beta}{1-\beta^n} \sigma (\tau + \rho) \leq A + D. \tag{6.79}$$

But $0 < \beta < 1$ thus, $1 - \beta^n > 0$, and it follows that

$$(1-\beta)\sigma (\tau + \rho)(1-\beta^n) \leq (A + D - t_0). \tag{6.80}$$

The application can meet its deadline only if $(A + D - t_0) > 0$, and under this condition this inequality becomes

$$\beta^n \leq \gamma \text{ with } \gamma = 1 - \frac{\sigma \times \tau}{A + D - t_0}. \tag{6.81}$$

If $\gamma \leq 0$, there is not enough time even for data distribution and the application should be rejected.

When $\gamma > 0$, then $n \leq \frac{\ln \gamma}{\ln \beta}$. Thus, the minimum number of nodes for the OPR strategy is

$$n^{min} = \frac{\ln \gamma}{\ln \beta}. \tag{6.82}$$

Equal Partitioning Rule (EPR). EPR assigns an equal workload to individual worker nodes, $\alpha_i = 1/n$. From the diagram in Figure 6.14 we see that

$$E^{(n,\sigma)} = \sum_{i=1}^{n} \frac{\sigma}{n} \times \tau + \frac{\sigma}{n} \times \rho = \sigma \times \tau + \frac{\sigma}{n} \times \rho. \quad (6.83)$$

The condition for meeting the deadline, $C^A(n) \leq A + D$, leads to

$$\sigma \times \tau + \frac{\sigma}{n} \times \rho \leq A + D \text{ or } n \geq \frac{\sigma \times \rho}{A + D - t_0 - \sigma \times \tau}. \quad (6.84)$$

Thus,

$$n^{min} = \left\lceil \frac{\sigma \times \rho}{A + D - t_0 - \sigma \times \tau} \right\rceil. \quad (6.85)$$

The pseudocode for a general schedulability test for FIFO, EDF, and MWF scheduling policies, for two-node allocation policies, MN (minimum number of nodes) and AN (all nodes), and for OPR and EPR partitioning rules is given in reference . The same paper reports on a simulation study for 10 algorithms. The generic format of the names of the algorithms is **Sp-No-Pa**, with **Sp** = FIFO/EDF/MWF, **No** = MN/AN, and **Pa** = OPR/EPR. For example, MWF-MN-OPR uses MWF scheduling, minimum number of nodes, and OPR partitioning. The relative performance of the algorithms depends on the relations between the unit cost of communication τ and the unit cost of computing ρ.

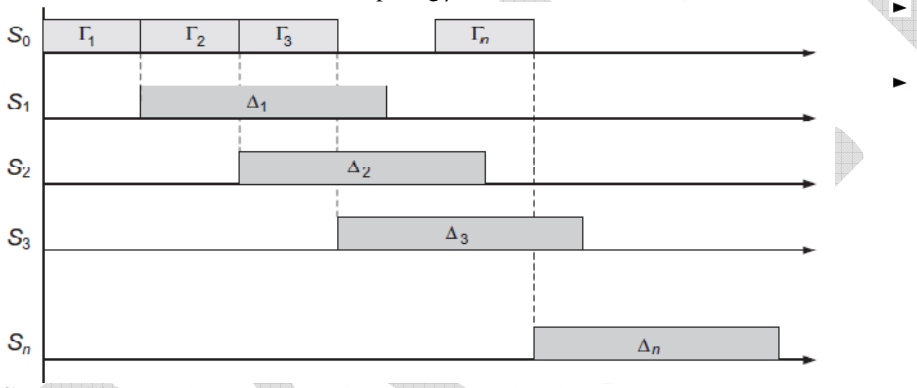


FIGURE 6.14: The timing diagram for the equal partitioning rule. The algorithm assigns an equal workload to individual worker nodes, $\alpha_i = 1/n$. The head node, S_0, distributes sequentially the data to individual worker nodes. The communication time is $_i = (\sigma/n) \times \tau$, $1 \; i \; n$. Worker node S_i starts processing the data as soon as the transfer is complete. The processing time is $_i = (\sigma/n) \times \rho$, $1 \; i \; n$.

6.13 Scheduling *MapReduce* applications subject to deadlines

Now we turn our attention to applications of the analysis in Section 6.12 and discuss 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.

A recent paper applies the deadline scheduling framework analyzed to *Hadoop* tasks. Table 6.8 summarizes 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 ρ_m and ρ_r , 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 σ is

$$E(n_m, n_r, \sigma) = \sigma \left(\frac{n_m + \varphi}{\rho_m} \quad \frac{n_r + \tau}{\rho_r} \right). \tag{6.86}$$

Thus, the condition that query $Q = (A, \sigma, D)$ with arrival time A meets the deadline expressed as

$$\frac{n_m + \varphi}{\rho_m} \quad \frac{n_r + \tau}{\rho_r} \quad A + D. \tag{6.87}$$

can be

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

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

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

$$t_m^0 + \sigma \left(\frac{\rho_m}{n_m} \quad t_r^{max} \right). \tag{6.89}$$

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

$$n_m^{min} \quad \frac{\sigma \rho_m}{t_r^{max} - t_m^0}, \text{thus,} \quad n_m^{min} = \frac{\sigma \rho_m}{t_r^{max} - t_m^0}. \tag{6.90}$$

The assumption of homogeneity of the servers can be relaxed and we assume that individual servers have different costs for processing a unit workload $\rho_m^i = \rho_m^j$ and $\rho_t^i = \rho_t^j$. 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.

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
ρm	Cost of processing a unit data in map task
ρr	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, $\alpha_i = 1/u$
ϕ	Filter ratio, the fraction of the input produced as output by the map process

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

6.14 Resource management and dynamic application scaling

The demand for computing resources, such as CPU cycles, primary and secondary storage, and net-work bandwidth, depends heavily on the volume of data processed by an application. The demand for resources can be a function of the time of day, can monotonically increase or decrease in time, or can experience predictable or unpredictable peaks. For example, a new Web service will experience a low request rate when the service is first introduced and the load will exponentially increase if the service is successful. A service for income tax processing will experience a peak around the tax filling deadline, whereas access to a service provided by Federal Emergency Management Agency (FEMA) will increase dramatically after a natural disaster.

The elasticity of a public cloud, the fact that it can supply to an application precisely the amount of resources it needs and that users pay only for the resources they consume are serious incentives to migrate to a public cloud. The question we address is: How scaling can actually be implemented in a cloud when a very large number of applications exhibit this often unpredictable behavior [62,233,357]. To make matters worse, in addition to an unpredictable external load the cloud resource management has to deal with resource reallocation due to server failures.

We distinguish two scaling modes: vertical and horizontal. *Vertical scaling* keeps the number of VMs of an application constant, but increases the amount of resources allocated to each one of them. This can be done either by migrating the VMs to more powerful servers or by keeping the VMs on the same servers but increasing their share of the CPU time. The first alternative involves additional overhead; the VM is stopped, a snapshot of it is taken, the file is transported to a more powerful server, and, finally, the VM is restated at the new site.

Horizontal scaling is the most common mode of scaling on a cloud; it is done by increasing the number of VMs as the load increases and reducing the number of VMs when the load decreases. Often, this leads to an increase in communication bandwidth consumed by the application. Load balancing among the running VMs is critical to this mode of operation. For a very large application, multiple load balancers may need to cooperate with one another. In some instances the load balancing is done by a front-end server that distributes incoming requests of a transaction-oriented system to back-end servers.

An application should be designed to support scaling. As we saw in Section 4.6 in the case of a *modularly divisible* application, the workload partitioning is static, it is decided a priori, and cannot be changed; thus, the only alternative is vertical scaling. In the case of an *arbitrarily divisible* application the workload can be partitioned dynamically; as the load increases, the system can allocate additional VMs to process the additional workload. Most cloud applications belong to this class, which justifies our statement that horizontal scaling is the most common scaling mode.

Mapping a computation means to assign suitable physical servers to the application. A very important first step in application processing is to identify the type of application and map it accordingly. For example, a communication-intensive application should be mapped to a powerful server to minimize the network traffic. This may increase the cost per unit of CPU usage, but it will decrease the computing time and probably reduce the overall cost for the user. At the same time, it will reduce the network traffic, a highly desirable effect from the perspective of the cloud service provider.

To scale up and down a compute-intensive application, a good strategy is to increase or decrease the number of VMs or instances. Because the load is relatively stable, the overhead of starting up or terminating an instance does not increase significantly the computing time or the cost.

There are several strategies to support scaling. *Automatic VM scaling* uses predefined metrics, e.g., CPU utilization, to make scaling decisions. Automatic scaling requires *sensors* to monitor the state of VMs and servers; *controllers* make decisions based on the information about the state of the cloud, often using a state machine model for decision making. Amazon and Rightscale (`www.rightscale.com`) offer automatic scaling. In the case of *AWS* the *CloudWatch* service supports applications monitoring and allows a user to set up conditions for automatic migrations.

Nonscalable or single-load balancers are also used for horizontal scaling. The *Elastic Load Balanc-ing* service from Amazon automatically distributes incoming application traffic across multiple *EC2* instances. Another service, the *Elastic Beanstalk*, allows dynamic scaling between a low and a high number of instances specified by the user. The cloud user usually has to pay for the more sophisticated scaling services such as *Elastic Beanstalk*.