CHAPTER

9 Cloud resource management and scheduling

Resource management is a core function of any manmade system; it affects the three dimensions of the system evaluation space: performance, functionality, and cost. Efficient resource management has a direct effect on performance and cost and an indirect effect on the functionality of the system because some of the functions may be avoided due to the poor performance and/or high cost.

A cloud is a complex system with a large number of shared resources subject to unpredictable requests and affected by external events it cannot control. Cloud resource management requires complex policies and decisions for multiobjective optimization. Effective resource management is extremely challenging due to the scale of the cloud infrastructure and to the unpredictable interactions of the system with a large population of users. The scale makes it impossible to have accurate global state information, and the large user population makes it nearly impossible to predict the type and intensity of system workload.

Resource management becomes even more complex when resources are oversubscribed and users are uncooperative, driven by self-interest. In addition to external factors, resource management is affected by internal factors, such as the heterogeneity of the hardware and software systems, the ability to approximate the global state of the system and to redistribute the load, the failure rates of various components, and many other factors.

Strategies for resource management associated with IaaS, PaaS, and SaaS delivery models differ. In all cases, cloud service providers are faced with large fluctuating workloads challenging the claim of cloud elasticity. When a spike can be predicted, resources can be provisioned in advance, e.g., for web services subject to seasonal spikes. The situation is slightly more complicated for unplanned spikes.

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. Auto-scaling for IaaS is complicated due to the lack of standards.

Centralized control cannot provide adequate solutions for management policies when changes in the environment are frequent and unpredictable. Distributed control poses its own challenges since it requires some form of coordination between the entities in control. Autonomic policies are of great interest due to the scale of the system and the unpredictability of the load when the ratio of peak to mean resource demands can be very large.

Throughout this text we use the term *bandwidth* in a broad sense to mean the number of operations or the amount of data transferred per time unit. For example, Mips (Million Instructions Per Second) or Mflops (Million Floating Point Instructions Per Second) measure the CPU speed, and Mbps (Megabits per second) measure the speed of a communication channel. The *latency* is defined as the time elapsed from the instance an operation is initiated until the instance its effect is sensed. Latency is context dependent. For example, the latency of a communication channel is the time it takes a bit to traverse the communication channel from its source to its destination; memory latency is the time elapsed from the instance a memory *read* instruction is issued until the time the data becomes available in a memory register. The demand for computing resources, such as CPU cycles, primary and secondary storage, and network bandwidth, depend heavily on the volume of data processed by an application.

This chapter covers a broad range of topics related to cloud resource management and scheduling. The sections marked (R) introduce the reader to cloud resource management research topics. An overview of policies and mechanisms for cloud resource management in Section 9.1 is followed by a discussion of scheduling algorithms for computer clouds in Section 9.2 and by an analysis of delay scheduling and of data-aware scheduling in Sections 9.3 and 9.4, respectively.

The Apache capacity scheduler is presented in Section 9.5, and the start-time fair queuing and the borrowed virtual time scheduling algorithms are analyzed in Sections 9.6 and 9.7, respectively. Cloud scheduling subject to deadlines and MapReduce scheduling with deadlines are discussed in Sections 9.8 and 9.9, respectively. Resource bundling and combinatorial auctions are covered in Section 9.10.

Cloud energy efficiency and cloud resource utilization and the impact of application scaling on resource management are analyzed in Sections 9.11 and 9.12, respectively. A control theoretic approach to resource allocations is discussed in Sections 9.13, 9.14, and 9.15, followed by a machine learning algorithm for coordination of specialized autonomic performance managers in Section 9.16.

A utility model for resource allocation for a web service is then presented in Section 9.17. Cloud infrastructure will most likely continue to scale up to accommodate the demands of an increasingly larger cloud user community. A fair question is whether current cloud management systems can sustain this expansion. Self-organization and self-management alternatives come to mind, but the very slow progress made by the autonomic computing initiative is likely to dampen the enthusiasm of those believing in self-management. We discuss emergence and self-organization in Section 9.18 and conclude this chapter with an in-depth analysis of cloud interoperability in Section 9.19.

9.1 Policies and mechanisms for resource management

A *policy* refers to *the principles guiding the decisions*, while *mechanisms* represent *the means to implement policies*. Separation of policies from mechanisms is a guiding principle in computer science. Butler Lampson [295] and Per Brinch Hansen [225] 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, and QoS guarantees. The explicit goal of an *admission control policy* is to prevent the system from accepting workload in violation of high-level system policies; for example, a system may not accept 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* is concerned with resource allocation to individual instances; an instance is an activation of a service. Locating resources for an instance is subject to multiple global optimization constraints and requires the search of a very large search space when the state of individual systems are changing rapidly.

Load balancing and *energy optimization* can be done locally, but global load balancing and energy optimization policies encounter the same difficulties as the ones we have already discussed. Load balancing and energy optimization are correlated and affect the cost for providing services [145].

The common meaning of the term “load balancing” is an even distribution of the workload 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 a perfect load balancing all servers would end up with the same load, 50% of each server’s capacity.

An important goal of cloud resource management is minimization of the cost for providing cloud service and, in particular, minimization of cloud energy consumption. This leads to a different meaning of “load balancing;” instead of evenly distributing the load amongst all servers, we wish to *concentrate the workload and use the smallest number of servers* while switching the others to a standby mode, a state where a server uses very little 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 to full capacity, while C and D will be switched to standby mode. *Quality of Service* is the resource management facet probably the most difficult to address and, at the same time possibly the most critical for the future of cloud computing.

Often, as we shall see in this section, resource management strategies jointly target performance and power consumption. The Dynamic Voltage and Frequency Scaling (DVFS)[1] techniques, such as Intel’s SpeedStep and AMD’s PowerNow, lower the voltage and the frequency to decrease the power consumption.[2] Motivated initially by the need to save power for mobile devices, these techniques have migrated virtually to all processors, including the ones used for high-performance servers.

Processor performance decreases, but at a substantially lower rate than energy consumption, as a result of lower voltages and clock frequencies [301]. Table 9.1 shows the dependence of normalized performance and normalized energy consumption of a typical modern processor on the clock rate. As we can see, at 1.8 GHz, we save 18% of the energy required for maximum performance, while the performance is only 5% lower than the peak performance achieved at 2.2 GHz. This seems a reasonable energy–performance tradeoff!

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. 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 rarely include energy trade-offs 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 systematic approach, rather than ad hoc methods. The four basic mechanisms for the implementation of resource management policies are:

[1] Dynamic voltage and frequency scaling is a power management technique to increase or decrease the operating voltage or frequency of a processor to increase the instruction execution rate and, respectively, to reduce the amount of heat generated and to conserve power.

[2] 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.

Table 9.1 Normalized performance and energy consumption, function of processor speed; performance decreases at a lower rate than energy consumption 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

- *Control theory.* Control theory uses the feedback to guarantee system stability and predict transient behavior [264,287], 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 [480]; this technique could be applied for coordination of several autonomic system managers, as discussed in [268].
- *Utility-based.* Utility-based approaches require a performance model and a mechanism to correlate user-level performance with cost as discussed in [11].
- *Market-oriented/economic mechanisms.* Such mechanisms do not require a model of the system, e.g., combinatorial auctions for bundles of resources discussed in [456].

A distinction should be made between interactive and noninteractive workloads. Management techniques for interactive workloads, e.g., web services, involve flow control and dynamic application placement, while those for noninteractive workloads are focused on scheduling. A fair amount of work reported in the literature is devoted to resource management of interactive workloads, some to noninteractive ones, and only a few to heterogeneous workloads, a combination of the two, e.g. [467].

9.2 Scheduling algorithms for computer clouds

Scheduling is a critical component of the cloud resource management responsible for resource sharing/multiplexing at several levels. A server can be shared among several VMs, each VM can 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 need 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). The objectives of a real-time system scheduler are to meet the deadlines and to be predictable.

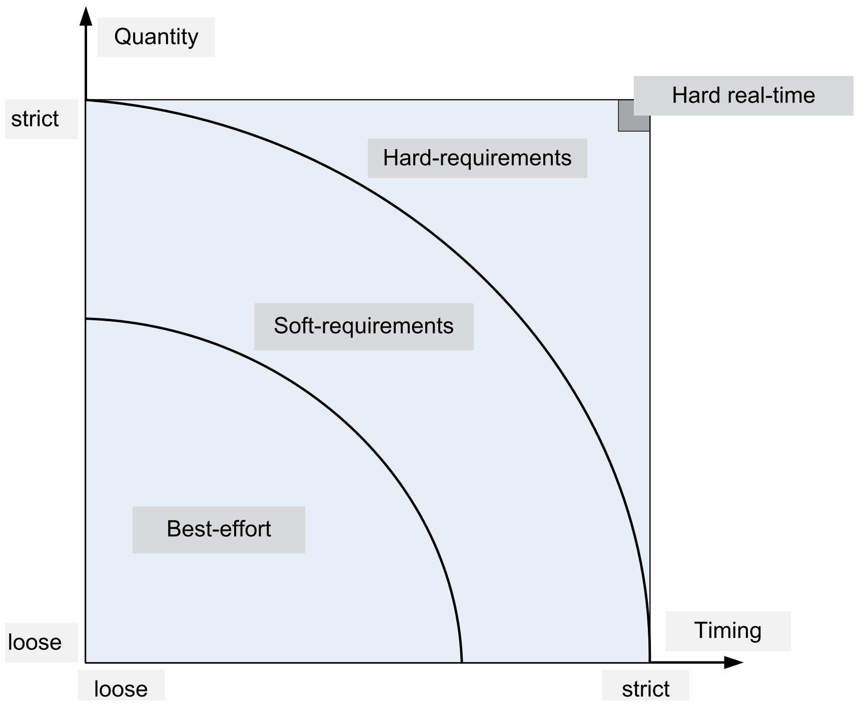

FIGURE 9.1

Resource requirement policies. *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 policies require statistically guaranteed amounts of resources and timing constraints. *Hard*-requirements policies demand strict timing and precise amounts of resources.

Schedulers for systems supporting a mixture 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/quantity of resources allocated; and (b) the timing when access to resources is granted. Fig. 9.1 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 requirements are the most challenging, as 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* [185]. 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, $B_i \leq b_i$.

- 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 [205] requires that the amount of work $\Omega_a(t_1, t_2)$ and $\Omega_b(t_1, t_2)$ in the time interval from t_1 to t_2 of two runnable threads a and b minimize the expression

$$\left| \frac{\Omega_a(t_1, t_2)}{w_a} - \frac{\Omega_b(t_1, t_2)}{w_b} \right|, \tag{9.1}$$

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

The QoS requirements differ for different classes of cloud applications and demand different scheduling policies. Best-effort applications, such as batch applications and analytics,[3] 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 constrains do not use a public cloud at this time, but may do so in the future.

Round-robin, first-come-first-serve (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 case of round-robin scheduling; the algorithm is fair and starvation-free. The threads are allowed to use the CPU in the order 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 applications is discussed in [71] and two new algorithms for integrated scheduling, the *Resource Allocation/Dispatching* (RAD) and the *Rate-Based Earliest Deadline* (RBED) are proposed.

Several algorithms of special interest for computer clouds are discussed below. These algorithms illustrate the evolution in thinking regarding the fairness of scheduling and the need to accommodate multi-objective scheduling, in particular scheduling for Big Data and for multimedia applications.

9.3 Delay scheduling (R)

How does one simultaneously ensure fairness and maximize resource utilization without compromising locality and throughput for Big Data applications running on large computer clusters? This is a question faced early on by large IT service providers and we discuss it next.

Hadoop scheduler. A Hadoop job consists of multiple Map and Reduce tasks and the question is how to allocate resources to the tasks of newly submitted jobs. Recall from Section 11.7 that the *job tracker* of the *Hadoop master* manages a number of slave servers running under the control of *task trackers*

[3] The term *analytics* is overloaded; sometimes it means discovery of patterns in the data; it could also mean statistical processing of the results of a commercial activity.

with slots for Map and Reduce tasks. A FIFO scheduler with five priority levels assigns slots to tasks based on their priority. The lower the number of tasks of a job already running on slots of all servers, the higher is the priority of the remaining tasks.

An obvious problem with this policy is that priority-based allocation does not consider data locality, namely the need to place tasks close to their input data. The network bandwidth in a large cluster is considerably lower than the disk bandwidth; also the latency for local data access is much lower than the latency of a remote disk access. Locality affects the throughput: *server locality*, i.e., getting data from the local server is significantly better in terms of time and overhead than *rack locality*, i.e. getting input data from a different server in the same rack.

In steady-state, priority scheduling leads to the tendency to assign the same slot repeatedly to the next task(s) of the same job. As one of the job's tasks completes execution its priority decreases and the available slot is allocated to the next task of the same job. Input data for every job are striped over the entire cluster so spreading the tasks over the cluster can potentially improve data locality, but the priority scheduling favors the occurrence of *sticky slots*.

According to [532] sticky slots did not occur in Hadoop at the time of the report "due to a bug in how Hadoop counts running tasks. Hadoop tasks enter a *commit pending* state after finishing their work, where they request permission to rename their output to its final filename. The job object in the master counts a task in this state as running, whereas the slave object doesn't. Therefore, another job can be given the task's slot." Data gathered at Facebook shows that only 5% of the jobs with a low number, 1–25, of Map tasks achieve server locality and only 59% show rack locality.

Task locality and average job locality. A task assignment satisfies the locality requirement if the input task data are stored on the server hosting the slot allocated to the task. We wish to compute the expected locality of job $\mathcal{J}$ with the fractional cluster share $f_{\mathcal{J}}$, assuming that a server has L slots and that each block of the files system has R replicas. By definition, $f_{\mathcal{J}} = n/N$ with n the number of slots allocated to job $\mathcal{J}$ and N the number of servers in the cluster.

The probability that a slot does not belong to $\mathcal{J}$ is $(1 - f_{\mathcal{J}})$, there are R replicas of block $\mathcal{B}_{\mathcal{J}}$ of job $\mathcal{J}$ and each replica is on a node with L slots. Thus, the probability that none of the slots of job $\mathcal{J}$ has a copy of block $\mathcal{B}_{\mathcal{J}}$ is $(1 - f_{\mathcal{J}})^{RL}$. It follows that $\mathcal{L}_{\mathcal{J}}$, the locality of job $\mathcal{J}$, can be at most

$$\mathcal{L}_{\mathcal{J}} = 1 - (1 - f_{\mathcal{J}})^{RL}. \tag{9.2}$$

How should a fair scheduler operate on a shared cluster? What is the number n of slots of a shared cluster the scheduler should allocate to jobs assuming that tasks of all jobs take an average of T seconds to complete? A sensible answer is that the scheduler should provide enough slots such that the response time on the shared cluster should be the same as $R_{n,\mathcal{J}}$, the completion time of job $\mathcal{J}$ would experience on a fictitious private cluster with n available slots for the n tasks of $\mathcal{J}$ as soon as job $\mathcal{J}$ arrives.

The job completion time can be approximated by the sum of the job processing time for all but the last task plus the waiting time until a slot becomes available for the last task of the job. There is no waiting time for running on the private cluster with n slots, while on a shared cluster the last task will have to wait before a slot is available.

A slot allocated to a task on the shared cluster will be free on average every T/N seconds; thus, the time the job will have to wait until all its n tasks have found a slot on the shared cluster will be $n \times T/N$. This implies that a fair scheduler should guarantee that the waiting time of job $\mathcal{J}$ on the shared cluster is much smaller than the completion time, $R_{n,\mathcal{J}}$ on the fictitious private cluster

$$R_{n,\mathcal{J}} \gg f_{\mathcal{J}} \times T. \tag{9.3}$$

Eq. (9.3) is satisfied if one of the following three conditions is satisfied:

1. $f_{\mathcal{J}}$ is small—there are many jobs sharing the cluster and the fraction of slots allocated to each job is small;
2. T is small—the individual tasks are short;
3. $R_{n,\mathcal{J}} \gg T$—the completion time of a job is much longer than the average task completion time; the large jobs dominate the workload.

The cumulative distribution function of running time of MapReduce jobs at Facebook resembles a sigmoid function[4] with the middle stage of a job duration starting at about 10 seconds and ending at about 1 000 seconds. The median completion time of a Map task is much shorter that the median completion time of a job, 19 versus 84 seconds. There are fewer Reduce tasks, but of a longer average duration, 231 seconds. Eighty-three percent of the jobs are launched within 10 seconds. Results reported in [532] show that delay scheduling performs well when most tasks are short relative to job duration, and when a running task can read a given data block from multiple locations.

Delay scheduling. A somewhat counterintuitive scheduling policy, *delay scheduling*, is proposed in [532]. As the name implies, the new policy delays scheduling the tasks of a new job for a relatively short time to address the conflict between fairness and locality.

This policy skips the task at the head of the priority queue if its input data are not available on the server where the slot is located. It repeats this process up to D times as specified by the delay scheduling algorithm pseudocode in the box on the next page. The almost doubling of the throughput under the new policy, while ensuring fairness for workloads at Yahoo and Facebook, is a good indication of delay scheduling policy merits.

An analysis of the new policy assumes a cluster with N servers and L slots per server, thus with a total number of slots $S = NL$. A job $\mathcal{J}$ prefers slots on servers where its data are stored, call this set of slots $\mathcal{P}_{\mathcal{J}}$. Call $p_{\mathcal{J}}$ the probability that a task of job $\mathcal{J}$ has data on the server with the slot allocated to it

$$p_{\mathcal{J}} = \frac{\mid \mathcal{P}_{\mathcal{J}} \mid}{N}. \tag{9.4}$$

The probability that a task skipped D times does not have the input data on the server the slot allocated to run the task resides decreases exponentially with D; after being skipped D times, this probability is $(1 - p_q\mathcal{J})^D$. For example, if $p_{\mathcal{J}} = 0.1$ and $D = 40$ then the probability of having data on the slot allocated to the task is $1 - (1 - p_{\mathcal{J}})^D = 0.99$, a 99% chance.

[4] A sigmoid function $S(t)$ is "S-shaped." It is defined as $S(t) = \frac{1}{1-e^{-t}}$, and its derivative can be expressed as function of itself, $S'(t) = S(t)(1 - S(t))$. S(t) can describe biological evolution with an initial segment describing early/childhood stage, a median stage representing maturity, and a third stage describing the late life stage.

Delay scheduling algorithm

```
1  Initialize $j.skipcount$ to 0 for all jobs $j$
2      when a heartbeat is received from node $n$
3          if $n$ has a free slot then
4              sort jobs in increasing order of number of running tasks
5              for $j$ in $jobs$ do
6                  if $j$ has unlaunched task $t$ with data on $n$ then
7                      launch $t$ on $n$
8                      set j.skipcount = 0
9                  else if $j$ has unlaunched task $t$ then
10                     if $j.skipcount > D + 1$ then
11                         launch $t$ on $n$
12                     else
13                         set $j.skipcount = j.skipcount + 1$
14                     end if
15                 end if
16             end for
17         end if
```

How does one achieve the desired level of locality for job $\mathcal{J}$ with n tasks? An approximate analysis reported in [532] assumes that all tasks are of the same length and that the preferred location sets $\mathcal{P}_{\mathcal{J}}$ are uncorrelated. If $\mathcal{J}$ has k task left to launch and the replication factor is as before equal to $\mathcal{R}$, then

$$p_{\mathcal{J}} = 1 - \left(1 - \frac{k}{N}\right)^R \tag{9.5}$$

and the probability of launching a task of $\mathcal{J}$'s after $\mathcal{D}$ skips is

$$p_{\mathcal{J},D} = 1 - (1 - p_{\mathcal{J}})^D = 1 - \left(1 - \frac{k}{N}\right)^{RD} \geq 1 - e^{-RDk/N}. \tag{9.6}$$

The expected value of $p_{\mathcal{J},D}$ is then

$$\mathcal{L}_{\mathcal{J},D} = \frac{1}{N}\sum_{k=1}^{N}\left(1 - e^{-RDk/N}\right) = 1 - \frac{1}{N}\sum_{k=1}^{N} e^{-RDk/N}. \tag{9.7}$$

Then

$$\mathcal{L}_{\mathcal{J},D} \geq 1 - \frac{1}{N}\sum_{k=1}^{\infty} e^{-RDk/N} = 1 - \frac{e^{-RD/N}}{N(1 - e^{-RD/N})} \tag{9.8}$$

It follows that a locality $\mathcal{L}_{\mathcal{J},D} \geq \lambda$ requires job $\mathcal{J}$ to forgo D times its turn for a new slot while the head of the priority queue, with D satisfying the following condition

$$D \geq -\frac{N}{R}\ln\frac{n(1-\lambda)}{1+n(1-\lambda)} \quad \text{or} \quad D \leq \frac{N}{R}\ln\left[1 + \frac{1}{n(1-\lambda)}\right]. \tag{9.9}$$

Hadoop fair scheduler (HFS). The next objective of [532] is the development of a more complex Hadoop scheduler with several new capabilities:

1. Fair sharing at the level of users rather than jobs. This requires a two-level scheduling: the first level allocates task slots to pools of jobs using a fair sharing policy; at the second level each pool allocates its slots to jobs in the pool.
2. User controlled scheduling; the second level policy can be either FIFO or fair sharing of the slots in the pool.
3. Predictable turnaround time. Each pool has a guaranteed minimum share of slots. To accomplish this goal HFS defines a *minimum share timeout* and a *fair share timeout* and when the corresponding timeout occurs it kills buggy jobs or tasks taking a very long time. Instead of using a minimum skip count D use a *wait time* to determine how long a job waits to allocate a slot to its next ready-to-run task.

HFS creates a sorted list of jobs ordered according to its scheduling policy. Then, it scans down this list to identify the job allowed to schedule a task next, and within each pool, it applies the pool's internal scheduling policy. Pools missing their minimum share are placed at the head of the sorted list and the other pools are sorted to achieve a weighted fair sharing. The pseudocode of the HFS scheduling algorithm maintains three variables for each job j initialized as $j.level = 0$, $j.wait = 0$, and $j.skipped = false$ when a heartbeat is received from node n:

HFS scheduling algorithm

```
for each job $j$ with $j.skipped = true$
 increase $j.wait$ by the time since the last heartbeat and set $j.skipped = false$
  if $n$ has a free slot then
    sort jobs using hierarchical scheduling policy
    for $j$ in jobs do
      if $j$ has a node-local task $t$ on $n$ then
        set $j.wait = 0$ and $j.level = 0$
        return $t$ to $n$
      else
          if $j$ has a rack-local task $t$ on $n$ and $j.level > 2$ or $j.wait > W1 + 1$ then
            set $j.wait = 0$ and $j.level = 1$ return $t$ to $n$
         else
            if $j.level = 2$ and $j.level = 1$ and $j.wait > W2 + 1)$
                  or $j.level = 0$ and $j.wait > W1 + W2 + 1$ then
                      set $j.wait = 0 and j.level = 2$ return any unlaunched task $t$ in $j$ to $n$
            else
                      set $j.skipped = true$
           end if
     end for
   end if
```

A job starts at locality level 0 and can only launch node-local tasks. After at least W1 seconds, the job advances at level 1 and may launch rack-local tasks, then after a further W2 seconds, it goes to level

2 and may launch off-rack tasks. If a job launches a local task with a locality higher than the level it is on, it goes back down to a previous level.

In summary, delay scheduling can be generalized to preferences other than locality and the only requirement is to have a sorted list of jobs according to some criteria. It can also be applied to resource types other than slots. Measurements reported in [532] show that near 100% locality can be achieved by relaxing fairness.

9.4 Data-aware scheduling (R)

The analysis of delay scheduling emphasizes the important role of data locality for I/O-intensive application performance. This is the topic discussed in this section and addressed by a paper covering task scheduling of I/O-intensive applications [492].

There are many I/O-intensive applications translated into jobs described as a DAG (Direct Acyclic Graph) of tasks. Examples of such applications include MapReduce, approximate query processing applied to exploratory data analysis and interactive debugging, and machine learning applied to spam classification and machine translation.

Jobs running such applications consist of multiple sets of tasks running in every stage; later-stage tasks consume data generated by early-stage tasks. For some of these applications different subsets of tasks can be scheduled independently to optimize data locality, without affecting the correctness of the results. This is the case of an application using the gradient descent algorithm.

The gradient descent is a first-order iterative optimization algorithm. To find local minima the algorithm takes steps proportional with the negative of the gradient of the function at each iteration. Fig. 9.2 shows that such an application has multiple stages and that groups of tasks at each stage can be scheduled independently. Data-aware scheduling improves locality hence, response time and performance.

In this context, *late binding* means correlating tasks with data dynamically, depending on the state of the cluster. *Data-aware scheduling* improves early-stage tasks locality and, whenever possible, the locality of the later-stage tasks of the job. Locality of all tasks is important because the job completion time is determined by the slowest task; thus, a special attention should be paid to *straggler*[5] tasks.

Recall from Chapter 4 that communication latency increases, while the communication bandwidth decreases with the "distance" between the server running a task and the one where the data is stored. An I/O-intensive task operates efficiently when its input data is already loaded in local memory and less efficiently when the data is stored on a local disk. The task efficiency decreases even further when the data resides on a different server of the same rack, and it is significantly lower when the data is on a server in a different rack.

A number of racks are interconnected by a cell switch; thus, one of the scheduler's objective is to balance cross-rack traffic. Intermediate stages of a job often involve group communication among tasks running on servers in different racks, e.g., as one-to-many, many-to-one, and many-to-many data exchanges. A second important goal of data-aware scheduling is to reduce cross-rack communication through the placement of the producer and consumer tasks.

[5] According to the Merriam Webster dictionary, "straggling" means to "walk or move in a slow and disorderly way," or "spread out from others of the same kind."

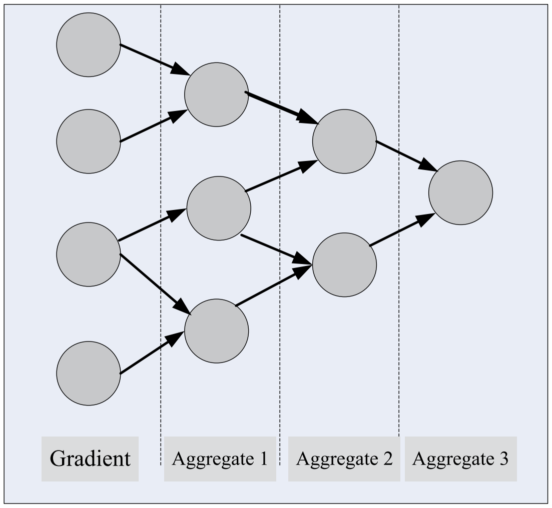

FIGURE 9.2

Stages of an application using the gradient descent algorithm. Tasks in the later stages of the computation use data produced by the early stage tasks. Of the four tasks in the *Gradient* stage, the top group of two tasks can be scheduled independently of the group of the lower two tasks.

KMN, the scheduler discussed in [492], launches a number of additional tasks in the early stages thus, allows choices for the later stage tasks. The name of the system comes from its basic ideas, it chooses K out of N blocks from the input data and schedules $M > K$ first-stage tasks on servers where these blocks reside. The "best" K out of $\binom{M}{K}$ choices increases the likelihood that upstream tasks outputs are distributed across racks and the next-stage tasks using these data as input are scheduled such that the cross-rack traffic is balanced. This heuristic is justified because selecting the best location of next stage tasks based on the output produced by earlier tasks is an NP-complete problem.

KMN is implemented in Scala,[6] and it is built on top of Spark, a system for in-memory cluster computing discussed in Section 4.12. One of the novel ideas of the KMN scheduler is to choose K out of N input data blocks to improve locality of a cluster with S slots per server. If u denotes the utilization of a slot, then server utilization when all its slots are busy is u^S. The probability that one of the S tasks running on the server has a block of input data on the local disk is $p_t = 1 - u^S$.

When we choose K out of N the scheduler can choose $\binom{N}{K}$ input block combinations. The probability that K out of N tasks enjoy locality, $p_{K|N}$, is given by the binomial distribution assuming that

[6] Scala is a general-purpose programming language with a strong static type system and support for functional programming. Scala code is compiled as Java byte code and runs on JVM (Java Virtual Machine). KMN consists of some 1 400 lines of Scala code.

the probability of success is p_t.

$$p_{K|N} = 1 - \sum_{i=0}^{K-1} \binom{N}{i} p_t^i (1 - p_t)^{N-i}, \tag{9.10}$$

or

$$p_{K|N} = 1 - \sum_{i=0}^{K-1} \binom{N}{i} \left(1 - u^S\right)^i u^{S(N-i)}. \tag{9.11}$$

It is easy to see that the probability of achieving locality is high even for very large utilization of the server slots, e.g., when $u = 90\%$.

The probability that all K blocks in one of the $f = \binom{N}{K}$ samples achieve locality is p_t^K, and, as the samples are independent, the probability that at lest one of the samples achieves locality is

$$p_{K|N}^{(1)} = (1 - p_t^K)^f. \tag{9.12}$$

This probability increases as f increases.

Selection of the best K outputs from the M upstream tasks using a round-robin strategy is described by the following pseudocode from [492]:

```
//Given: upstreamTasks - list with rack, index within rack for each task
//Given: K - number of tasks to pick
// Number of upstream tasks in each rack
upstreamRacksCount = map()
// Initialize
for task in upstreamTasks do
          upstreamRacksCount[task:rack] += 1
end for
// Sort the tasks in round-robin fashion
roundRobin = upstreamTasks.sort(CompareTasks)
chosenK = roundRobin[0 : K]
return chosenK
procedure COMPARETASKS(task1; task2)
        if task1:idx != task2:idx then
               // Sort first by index
        return task1:idx < task2:idx
        else
              // Then by number of outputs
              numRack1 = upstreamRacksCount[task1:rack]
              numRack2 = upstreamRacksCount[task2:rack]
              return numRack1 > numRack2
         end if
end procedure
```

A hash map includes the list of the upstream tasks and how many tasks should run on each rack. Then the tasks are sorted first by their index in the rack and then by the number of tasks in the rack.

Experiments conducted on an EC2 cluster with 100 servers show that when the KMN scheduler is used instead of the native *Spark* scheduler the average job completion time is reduced by 81%. This reduction is due to the 98% locality of input tasks and a 48% improvement in data transfer. The overhead of the KMN scheduler is small, it uses 5% additional resources.

9.5 Apache capacity scheduler

Apache capacity scheduler [28] is a pluggable MapReduce scheduler for Hadoop. It supports multiple queues, job priorities, and guarantees to each queue a fraction of the capacity of the cluster. Other features of the scheduler are:

1. Free resources can be allocated to any queue beyond its guaranteed capacity. Excess allocated resources can be reclaimed and made available to another queue to meet its guaranteed capacity.
2. Excess resources taken from a queue will be restored to the queue within N minutes from the instance they are need.
3. Higher priority jobs in a queue have access to resources allocated to the queue before jobs with lower priority have access.
4. Does not support preemption; once running, a job will not be preempted for a higher priority job.
5. Each queue enforces a limit on the percentage of resources allocated to a user at any given time, if there is competition for them.
6. Supports memory-intensive jobs. A job can specify higher memory requirements than the default, and the tasks of the job will only be run on *TaskTrackers*[7] that have enough memory to spare.

When a *TaskTracker* is free, the scheduler chooses the queue that needs to reclaim any resources the earliest and if no such queue exits it then picks a queue whose ratio of the number of running slots to the guaranteed capacity is the lowest. Once a queue is selected, the scheduler chooses a job in the queue. Jobs are sorted based on submission time and priority (if supported). Once a job is selected, the scheduler chooses a task to run. Periodically, the scheduler takes actions allowing queues to reclaim capacity as follows:

(a) A queue reclaims capacity when it has at least one task pending and a fraction of its guaranteed capacity is used by another queue; the scheduler determines the amount of resources to reclaim within the reclaim time for the queue.
(b) The scheduler kills the tasks that started the last when a queue which has already received Fair Queue resources, it is allowed to reclaim, but its reclaim time is about to expire.

The scheduler can be configured with several properties for each queue using the file *conf/capacity-scheduler.xml*. Queue properties can be defined by concatenating the string *mapred.capacity-scheduler.queue.<queue-name>* with the property name:

[7] In *Hadoop*, the *JobTracker* and *TaskTracker* daemons handle the processing of *MapReduce* jobs.

.*guaranteed capacity*—percentage of the number of slots guaranteed to be available for jobs in the queue i.
.*reclaim-time-limit*—the amount of time, in seconds, before resources distributed to other queues will be reclaimed.
.*supports-priority*; if true, priorities of jobs will be taken into account in scheduling decisions.
.*user-limit-percent*; if there is competition for resources each queue enforces a limit on the percentage of resources allocated to a user at any given time. If two users have submitted jobs to a queue, no single user can use more than 50% of the queue resources. If a third user submits a job, no single user can use more than 33% of the queue resources. With 4 or more users, no user can use more than 25% of the queue's resources. A value of 100% implies no user limits are imposed.

9.6 Start-time fair queuing (R)

A hierarchical CPU scheduler for multimedia operating systems was proposed in [205]. 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

$$\frac{B_i}{B} = \frac{w_i}{\sum_{j=1}^{n} w_j} \tag{9.13}$$

with w_j, $1 \le j \le n$, the weight of the n children of node i; see Fig. 9.3.

When a VM is not active, its bandwidth is reallocated to the other VMs active at the time; when one of the applications of a VM 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, then its allocation is transferred to the other threads of the application.

Call $v_a(t)$, $v_b(t)$ the virtual time of threads a, b and $v(t)$ the virtual time of the scheduler at time t. Call q the time quantum of the scheduler, in milliseconds. The threads a and b have their time quantum, q_a and q_b, weighted by w_a and w_b, respectively; thus, in our example, the time quantum of the two threads are q/w_a and q/w_b, respectively. The i-th activation of thread a will start at 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:

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

$$S_x^i(t) = \max\left\{v(\tau^j),\, F_x^{(i-1)}(t)\right\} \quad \text{and} \quad S_x^0 = 0. \tag{9.14}$$

   The condition for thread i to be started is that thread $(i-1)$ has finished and that the scheduler is active.
3. The virtual finish time of the i-th activation of thread x is

$$F_x^i(t) = S_x^i(t) + \frac{q}{w_x}. \tag{9.15}$$

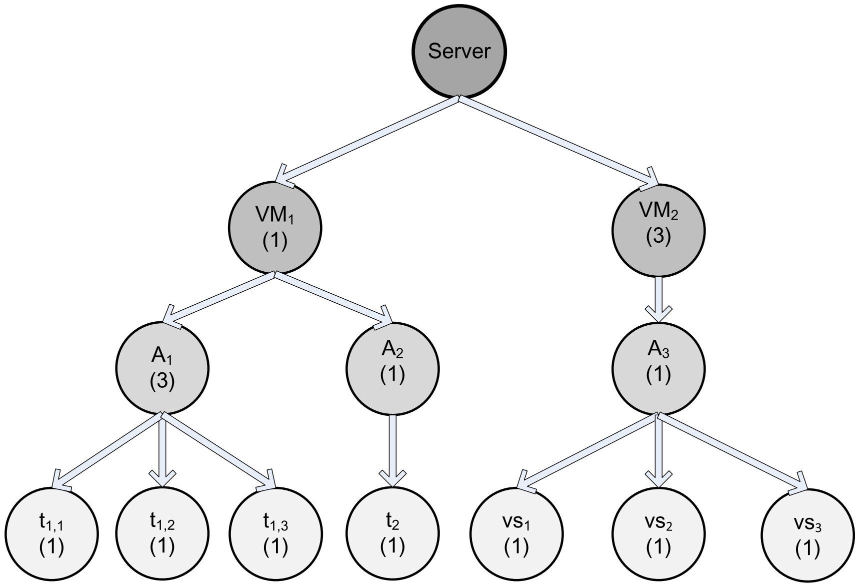

FIGURE 9.3

SFQ tree for scheduling when 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 VMs, applications, and individual threads are shown in parenthesis.

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.

4. 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:

$$v(t) = \begin{cases} \text{Virtual start time of the thread in service at time } t, & \text{if CPU busy} \\ \text{Maximum finish virtual time of any thread,} & \text{if CPU idle} \end{cases} \tag{9.16}$$

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'll use in our examples the real time as the index of the event, in other words at S_a^6 means the 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 Fig. 9.4.

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 at successive intervals of time are:

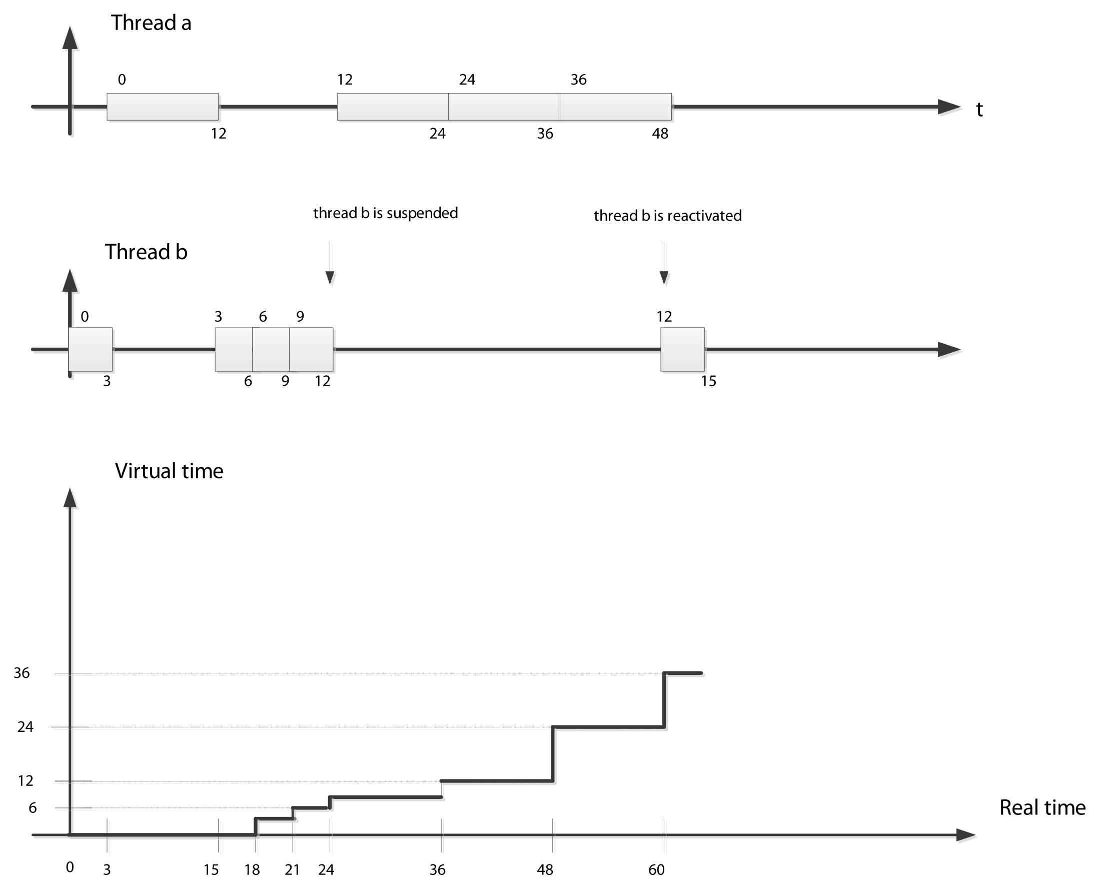

FIGURE 9.4

(Top) Virtual startup time $S_a(t)$ and $S_b(t)$ and virtual finish time $F_a(t)$ and $F_b(t)$ function of real time t for each activation of threads a and b, respectively, are marked at the top and, respectively, at the bottom of the box representing a running thread. (Bottom) Scheduler virtual time $v(t)$ function of the real time.

$\underline{t=0}$: 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

$$F_b^0 = S_b^0 + q/w_b = 0 + 12/4 = 3. \tag{9.17}$$

$\underline{t=3}$: both threads are runnable and thread b was in service, thus, $v(3) = S_b^0 = 0$; then

$$S_b^1 = \max\left\{v(3), F_b^0\right\} = \max\{0, 3\} = 3. \tag{9.18}$$

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

$$F_a^0 = S_a^0 + q/w_a = 0 + 12/1 = 12. \tag{9.19}$$

$\underline{t = 15}$: both threads are runnable and thread a was in service at this time thus,

$$v(15) = S_a^0 = 0 \quad \text{and} \quad S_a^1 = \max\left\{v(15), F_a^0\right\} = \max\{0, 12\} = 12. \tag{9.20}$$

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

$$F_b^1 = S_b^1 + q/w_b = 3 + 12/4 = 6. \tag{9.21}$$

$\underline{t = 18}$: both threads are runnable, and thread b was in service at this time, thus,

$$v(18) = S_b^1 = 3 \quad \text{and} \quad S_b^2 = \max\left\{v(18), F_b^1\right\} = \max\{3, 6\} = 6. \tag{9.22}$$

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

$$F_b^2 = S_b^2 + q/w_b = 6 + 12/4 = 9. \tag{9.23}$$

$\underline{t = 21}$: both threads are runnable, and thread b was in service at this time, thus,

$$v(21) = S_b^2 = 6 \quad \text{and} \quad S_b^3 = \max\left\{v(21), F_b^2\right\} = \max\{6, 9\} = 9. \tag{9.24}$$

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

$$F_b^3 = S_b^3 + q/w_b = 9 + 12/4 = 12. \tag{9.25}$$

$\underline{t = 24}$: Thread b was in service at this time, thus,

$$v(24) = S_b^3 = 9 \quad \text{and} \quad S_b^4 = \max\left\{v(24), F_b^3\right\} = \max\{9, 12\} = 12. \tag{9.26}$$

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

$$F_a^1 = S_a^1 + q/w_a = 12 + 12/1 = 24. \tag{9.27}$$

$\underline{t = 36}$: thread a was in service, and it is the only runnable thread at this time, thus,

$$v(36) = S_a^1 = 12 \quad \text{and} \quad S_a^2 = \max\left\{v(36), F_a^2\right\} = \max\{12, 24\} = 24 \quad \text{then} \tag{9.28}$$

$$F_a^2 = S_a^2 + q/w_a = 24 + 12/1 = 36. \tag{9.29}$$

$\underline{t = 48}$: thread a was in service, and it is the only runnable thread at this time thus,

$$v(48) = S_a^2 = 24 \quad \text{and} \quad S_a^3 = \max\left\{v(48), F_a^2\right\} = \max\{24, 36\} = 36, \quad \text{then} \tag{9.30}$$

$$F_a^3 = S_a^3 + q/w_a = 36 + 12/1 = 48. \tag{9.31}$$

$\underline{t = 60}$: thread a was in service at this time, thus,

$$v(60) = S_a^3 = 36 \quad \text{and} \quad S_a^4 = \max\left\{v(60), F_a^3\right\} = \max\{36, 48\} = 48. \tag{9.32}$$

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

$$F_b^4 = S_b^4 + q/w_b = 12 + 12/4 = 15. \tag{9.33}$$

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 startup 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 overhead of SFQ algorithm is comparable to that of the Solaris scheduling algorithm [205].

9.7 Borrowed virtual time (R)

The objective of the *borrowed virtual time* (BVT) scheduling 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 [154]. Like SFQ, the BVT algorithm supports scheduling a mixture 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, as well as 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).

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

$$E_i \leftarrow \begin{cases} A_i & \text{if } warpBack = OFF \\ A_i - W_i & \text{if } warpBack = ON \end{cases} \tag{9.34}$$

The algorithm measures the time in *minimum charging units*, mcu, and uses a time quantum called *context switch allowance (C)*, which measures in multiples of mcu, the real time a thread is allowed to run when competing with other threads; typical values for the two quantities are $mcu = 100$ µsec 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, an interrupt occurs; context switching also occurs when a thread becomes runnable after sleeping.

This policy prevents a thread that has been sleeping for a long time to claim control of the CPU for a longer period of time than it deserves. When thread τ_i becomes runnable after sleeping, its actual virtual time is updated $A_i \leftarrow \max\{A_i, SVT\}$.

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

Table 9.2 The real time and the effective virtual time $E_a(t)$ and $E_b(t)$ at time of a context switch. There is no time warp, thus, 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. CS—context switch; rt—real time; RT—running thread.

CS	rt	RT	Effective virtual time of running thread	Thread running next
1	$t = 2$	a	$E_a(2) = A_a(2) = A_a(0) + \Delta/3 = 30$	b, $E_b(2) = 0 < E_a(2) = 30$
2	$t = 5$	b	$E_b(5) = A_b(5) = A_b(0) + \Delta = 90$	a, $E_a(5) = 30 < E_b(5) = 90$
3	$t = 11$	a	$E_a(11) = A_a(11) = A_a(2) + \Delta = 120$	b, $E_b(11) = 90 < E_a(11) = 120$
4	$t = 14$	b	$E_b(14) = A_b(14) = A_b(5) + \Delta = 180$	a, $E_a(14) = 120 < E_b(14) = 180$
5	$t = 20$	a	$E_a(20) = A_a(20) = A_a(11) + \Delta = 210$	b, $E_b(20) = 180 < E_a(20) = 210$
6	$t = 23$	b	$E_b(23) = A_b(23) = A_b(14) + \Delta = 270$	a, $E_a(23) = 210 < E_b(23) = 270$
7	$t = 29$	a	$E_a(29) = A_a(29) = A_a(20) + \Delta = 300$	b, $E_b(29) = 270 < E_a(29) = 300$
8	$t = 32$	b	$E_b(32) = A_b(32) = A_b(23) + \Delta = 360$	a, $E_a(32) = 300 < E_b(32) = 360$
9	$t = 38$	a	$E_a(38) = A_a(38) = A_a(29) + \Delta = 390$	b, $E_b(11) = 360 < E_a(11) = 390$
10	$t = 41$	b	$E_b(41) = A_b(41) = A_b(32) + \Delta = 450$	a, $E_a(41) = 390 < E_b(41) = 450$

its actual virtual time upon completion of a run $A_i \leftarrow A_i + \Delta$. Given two threads a and b, then $\Delta = k_a/w_a = k_b/w_b$.

The BVT 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

$$A_j \leq A_i - \frac{C}{w_i}. \tag{9.35}$$

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

$a : 0, 5, 5 + 9 = 14, 14 + 9 = 23, 23 + 9 = 32, 32 + 9 = 41, \ldots$ and
$b : 2, 2 + 9 = 11, 11 + 9 = 20, 20 + 9 = 29, 29 + 9 = 38, \ldots.$

Context switches occur at real times 2, 5, 11, 14, 20, 23, 29, 32, 38, 41, ... The time is expressed in units of *mcu*. The initial run is a shorter one, consisting of only 3 *mcu*; a context switch occurs when a, which runs first, exceeds b by 2 *mcu*.

Table 9.2 shows the effective virtual time of the two threads at the time of each context switch. At that moment, the actual virtual time is incremented by an amount equal to Δ if the thread was allowed to run for its time allocation. The scheduler compares the effective virtual time of the threads and runs first the one with the minimum effective virtual time.

Fig. 9.5 displays the effective virtual time and the real time of the threads a and b. When a thread is running, its effective virtual time increases as the real time increase; 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 the thread a, with a larger weight, consumes its real time more slowly.

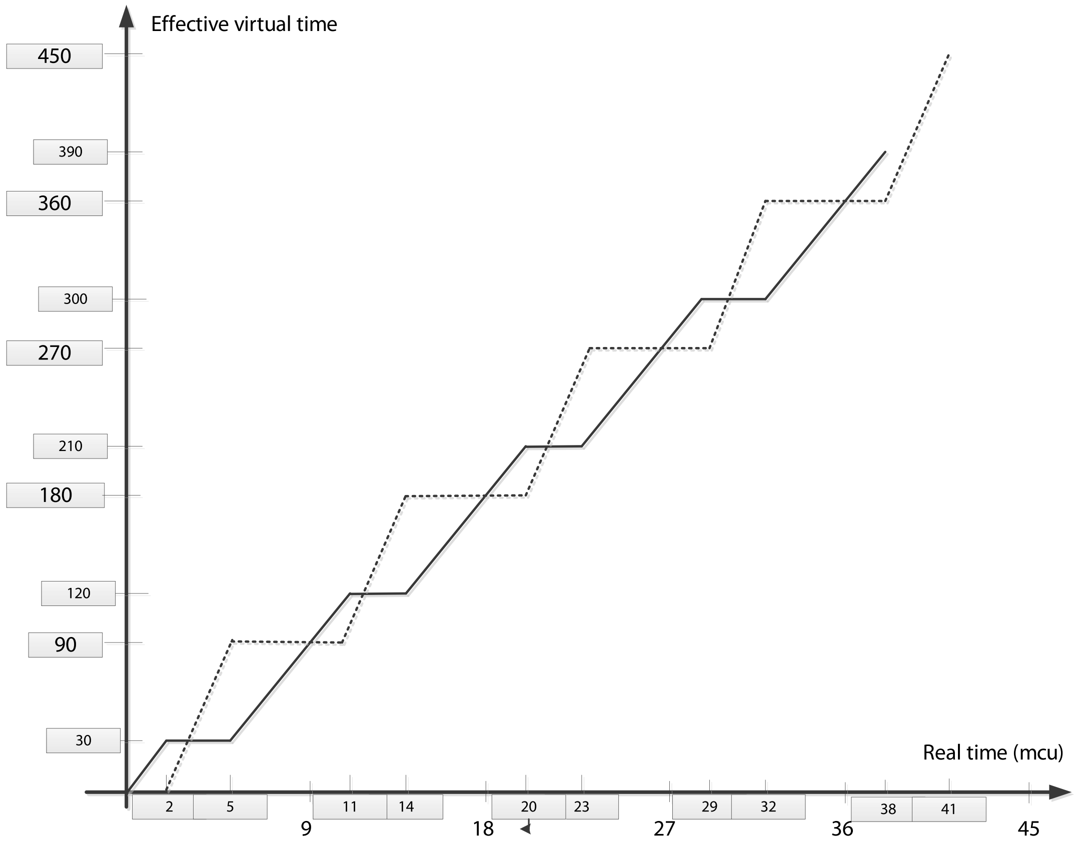

FIGURE 9.5

Example 1. The effective virtual time and the real time of the threads a (solid line) and b (dotted line) with weights $w_a = 2w_b$ when the actual virtual time is incremented in steps of $\Delta = 90$ mcu. The real time the two threads are allowed to use the CPU is proportional with 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, which wakes up at time $t = 9$ and then periodically at times $t = 18, 27, 36, \ldots$ for 3 units of time.

Table 9.3 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 $t = 2, 5, 9, 12, 14, 18, 21, 23, 27, 30, 32, 36, 39, 41, \ldots$. Context switches at times $t = 9, 18, 27, 36, \ldots$ are triggered by the waking up of the thread c which preempts the currently running thread. At $t = 9$ the time warp $W_c = -60$ gives priority to thread c. Indeed, $E_c(9) = A_c(9) - W_c = 0 - 60 = -60$ 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

Table 9.3 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*

application thread finishes and the scheduler chooses b to be dispatched first. We should also notice that the ratio of real times used by a and b is the same, as $w_a = 2w_b$. Fig. 9.6 shows the effective virtual times for the three threads a, b, and c. Every time when thread c wakes up, it preempts the current running thread, and it is immediately scheduled to run.

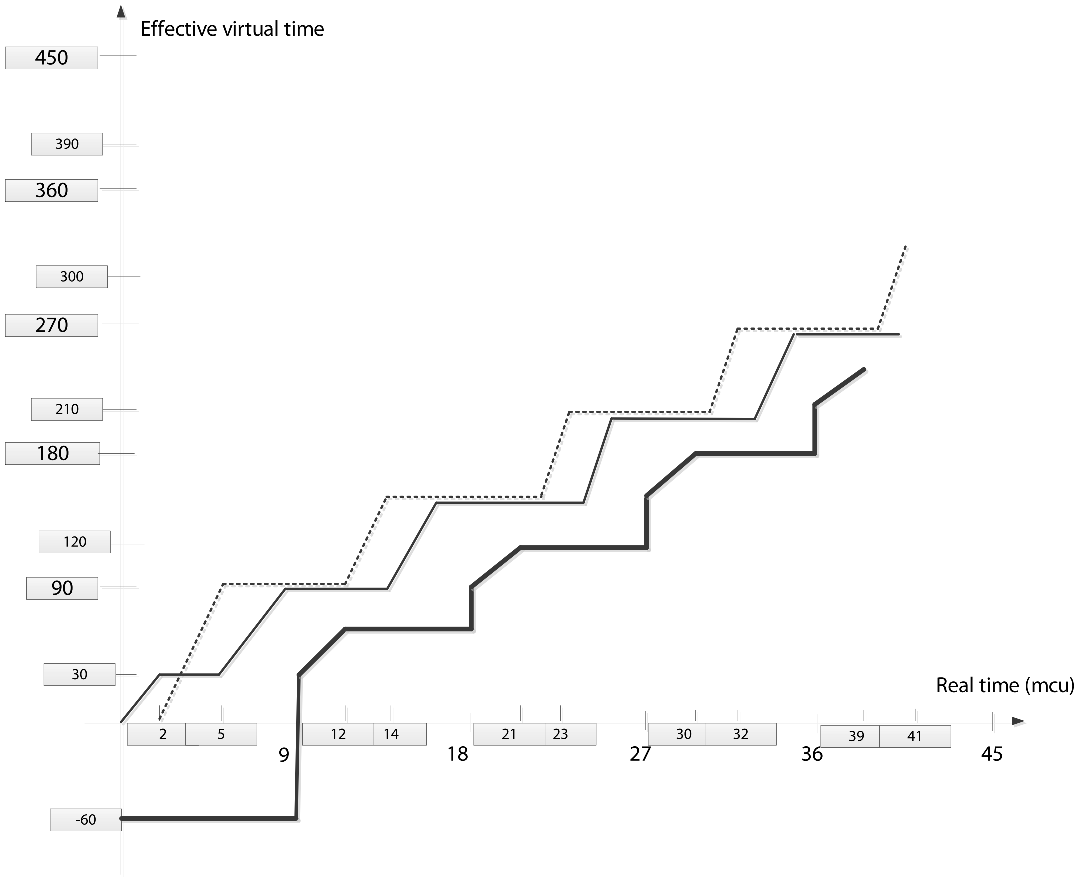

FIGURE 9.6

Example 2. The effective virtual time and the real time of the threads *a* (solid line), *b* (dotted line), and the *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*.

9.8 Cloud scheduling subject to deadlines (R)

Often, a service level agreement 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 from 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*, Π_i^q, with period q are identical, $\Pi_i^q \equiv \Pi^q$, 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*, Π_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 Π_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 which 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 are more of a guideline, 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. 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 every 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 Π_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:

- FIFO—First-In-First-Out, the tasks are scheduled for execution in order of their arrival.
- EDF—Earliest Deadline First, the task with the earliest deadline is scheduled first.
- MWF—Maximum Workload Derivative First.

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

$$DC_i(n^{min}) = W_i(n_i^{min} + 1) - W_i(n_i^{min}), \tag{9.36}$$

with $W_i(n)$ the workload allocated to task Π_i when n nodes of the cloud are available. The MWF policy 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}.

Two workload partitioning and task mappings to worker nodes are discussed next, the optimal partitioning and the equal partitioning. In our discussion, we use the derivations and some of the notations in [308]. These notations are summarized in Table 9.4.

Optimal Partitioning Rule (OPR). The workload is partitioned to ensure the earliest possible completion time. The optimality of OPR scheduling requires all tasks to finish execution at the same time.

Head node, S_0, distributes data sequentially to individual worker nodes. The workload assigned to worker node S_i is $\alpha_i \sigma$. The time to deliver input data to worker node S_i is $\Gamma_i = (\alpha_i \times \sigma) \times \tau, 1 \leq i \leq n$.

Table 9.4 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 $1 \leq i \leq n$ 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	time for transferring a unit of workload from the head node S_0 to worker S_i
ρ	time for 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, $\Gamma_i = \alpha_i \times \sigma \times \tau$, $1 \leq i \leq n$
Δ_i	time the worker S_i needs to process a unit of data, $\Delta_i = \alpha_i \times \sigma \times \rho$, $1 \leq i \leq n$
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 $\mathcal{A}$

Worker node S_i starts processing the data as soon as the transfer is complete. The processing time of worker node S_i is $\Delta_i = (\alpha_i \times \sigma) \times \rho$, $1 \leq i \leq n$.

The timing diagram in Fig. 9.7 allows us to determine the execution time $\mathcal{E}(n,\sigma)$ for the OPR as

$$\begin{aligned}
\mathcal{E}(1,\sigma) &= \Gamma_1 + \Delta_1 \\
\mathcal{E}(2,\sigma) &= \Gamma_1 + \Gamma_2 + \Delta_2 \\
\mathcal{E}(3,\sigma) &= \Gamma_1 + \Gamma_2 + \Gamma_3 + \Delta_3 \\
&\vdots \\
\mathcal{E}(n,\sigma) &= \Gamma_1 + \Gamma_2 + \Gamma_3 + \ldots + \Gamma_n + \Delta_n.
\end{aligned} \tag{9.37}$$

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

$$\begin{aligned}
\mathcal{E}(1,\sigma) &= \alpha_1 \times \sigma \times \tau + \alpha_1 \times \sigma \times \rho \\
\mathcal{E}(2,\sigma) &= \alpha_1 \times \sigma \times \tau + \alpha_2 \times \sigma \times \tau + \alpha_2 \times \sigma \times \rho \\
\mathcal{E}(3,\sigma) &= \alpha_1 \times \sigma \times \tau + \alpha_2 \times \sigma \times \tau + \alpha_3 \times \sigma \times \tau + \alpha_3 \times \sigma \times \rho \\
&\vdots \\
\mathcal{E}(n,\sigma) &= \alpha_1 \times \sigma \times \tau + \alpha_2 \times \sigma \times \tau + \alpha_3 \times \sigma \times \tau + \ldots + \alpha_n \times \sigma \times \tau + \alpha_n \times \sigma \times \rho.
\end{aligned} \tag{9.38}$$

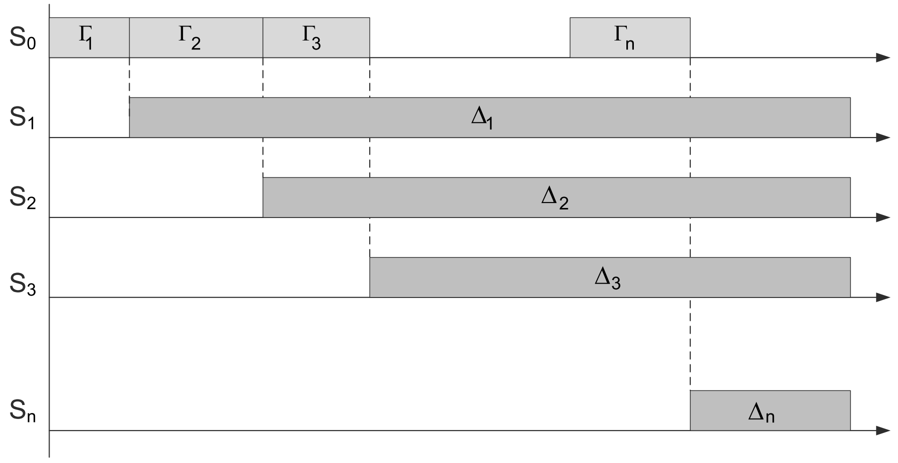

FIGURE 9.7

OPR timing diagram. The algorithm requires worker nodes to complete execution at the same time.

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

$$\alpha_1 = \frac{\alpha_2}{\beta} \quad \text{with} \quad \beta = \frac{\rho}{\tau + \rho}, \quad 0 \leq \beta \leq 1. \tag{9.39}$$

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

$$\alpha_i = \beta \times \alpha_{i-1} = \beta^{i-1} \times \alpha_1. \tag{9.40}$$

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

$$\sum_{i=1}^{n} \alpha_i = 1. \tag{9.41}$$

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

$$\alpha_1 + \beta \times \alpha_1 + \beta^2 \times \alpha_1 + \beta^3 \times \alpha_1 \ldots \beta^{n-1} \times \alpha_1 = 1 \quad \text{or} \quad \alpha_1 = \frac{1-\beta}{1-\beta^n}. \tag{9.42}$$

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

$$\mathcal{E}(n,\sigma) = \alpha_1 \times \sigma \times \tau + \alpha_1 \times \sigma \times \rho = \frac{1-\beta}{1-\beta^n}\sigma(\tau+\rho). \tag{9.43}$$

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

$$C^{\mathcal{A}}(n) = t_0 + \mathcal{E}(n,\sigma) = t_0 + \frac{1-\beta}{1-\beta^n}\sigma(\tau+\rho). \tag{9.44}$$

The application meets its deadline if and only if

$$C^{\mathcal{A}}(n) \leq A + D, \tag{9.45}$$

or

$$t_0 + \mathcal{E}(n,\sigma) = t_0 + \frac{1-\beta}{1-\beta^n}\sigma(\tau+\rho) \leq A + D. \tag{9.46}$$

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

$$(1-\beta)\sigma(\tau+\rho) \leq (1-\beta^n)(A+D-t_0). \tag{9.47}$$

The application can meet its deadline only if $(A + D - t_0) > 0$, and under this condition, the inequality (9.47) becomes

$$\beta^n \leq \gamma \quad \text{with} \quad \gamma = 1 - \frac{\sigma \times \tau}{A + D - t_0}. \tag{9.48}$$

If $\gamma \leq 0$, there is not enough time even for data distribution and the application should be rejected. When $\gamma > 0$, then $n \geq \frac{\ln\gamma}{\ln\beta}$. Thus, the minimum number of nodes for the OPR strategy is

$$n^{min} = \lceil \frac{\ln\gamma}{\ln\beta} \rceil. \tag{9.49}$$

Equal Partitioning Rule (EPR). EPR assigns an equal workload to individual worker nodes, $\alpha_i = 1/n$. The workload allocated to worker node S_i is σ/n. The head node, S_0, distributes sequentially the data to individual worker nodes. The time to deliver the input data to S_i is $\Gamma_i = (\sigma/n) \times \tau$, $1 \leq i \leq n$. Worker node S_i starts processing the data as soon as the transfer is complete. The processing time for node S_i is $\Delta_i = (\sigma/n) \times \rho$, $1 \leq i \leq n$.

From the diagram in Fig. 9.8, we see that

$$\mathcal{E}(n,\sigma) = \sum_{i=1}^{n} \Gamma_i + \Delta_n = n \times \frac{\sigma}{n} \times \tau + \frac{\sigma}{n} \times \rho = \sigma \times \tau + \frac{\sigma}{n} \times \rho. \tag{9.50}$$

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

$$t_0 + \sigma \times \tau + \frac{\sigma}{n} \times \rho \leq A + D \quad \text{or} \quad n \geq \frac{\sigma \times \rho}{A + D - t_0 - \sigma \times \tau}. \tag{9.51}$$

Thus,

$$n^{min} = \lceil \frac{\sigma \times \rho}{A + D - t_0 - \sigma \times \tau} \rceil. \tag{9.52}$$

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 [308]. The same reference reports on a simulation study for ten algorithms.

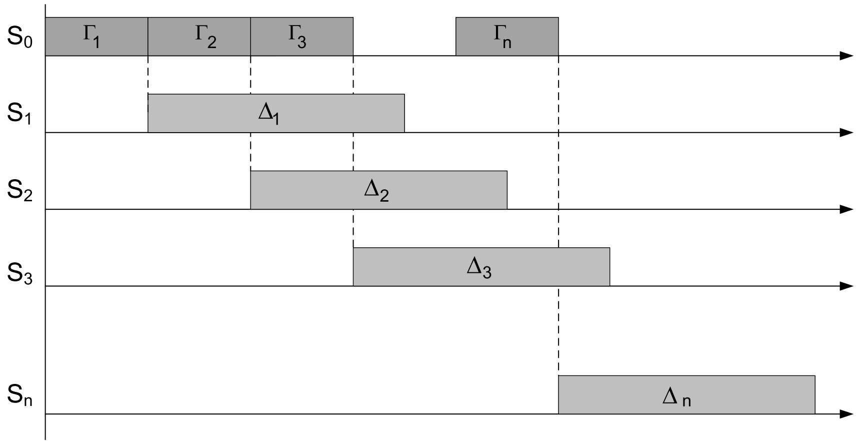

FIGURE 9.8

EPR timing diagram.

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.

9.9 MapReduce application scheduling subject to deadlines (R)

We now discuss cloud scheduling of MapReduce applications subject to deadlines. Several options for scheduling Apache Hadoop, an open source implementation of the MapReduce algorithm are: FIFO, the Fair Scheduler [532], the Capacity Scheduler, and the Dynamic Proportional Scheduler [434].

A recent paper [267] applies the deadline scheduling framework discussed in Section 9.8 to Hadoop tasks. Table 9.5 summarizes the notations used for this analysis. 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, 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

$$\mathcal{E}(n_m, n_r, \sigma) = \sigma \left[\frac{\rho_m}{n_m} + \phi \left(\frac{\rho_r}{n_r} + \tau \right) \right]. \tag{9.53}$$

Table 9.5 Parameters used for Hadoop 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, $1 \le i \le u$
Π_r^j	a reduce task, $1 \le j \le v$
J	the job to perform the query $Q = (A, \sigma, D)$, $J = (\Pi_m^1, \Pi_m^2, \ldots, \Pi_m^u, \Pi_r^1, \Pi_r^2, \ldots, \Pi_r^v)$
τ	cost for transferring a data unit
ρ_m	map task time for processing a unit data
ρ_r	reduce task time for processing a unit data
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

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

$$t_m^0 + \sigma \left[\frac{\rho_m}{n_m} + \phi \left(\frac{\rho_r}{n_r} + \tau \right) \right] \le A + D. \tag{9.54}$$

It follows immediately that the maximum value for the startup time of the *reduce* task is

$$t_r^{max} = A + D - \sigma \phi \left(\frac{\rho_r}{n_r} + \tau \right). \tag{9.55}$$

We now plug the expression of the maximum value for the startup time of the *reduce* task in the condition to meet the deadline

$$t_m^0 + \sigma \frac{\rho_m}{n_m} \le t_r^{max}. \tag{9.56}$$

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

$$n_m^{min} \ge \frac{\sigma \rho_m}{t_r^{max} - t_m^0}, \quad \text{thus,} \quad n_m^{min} = \lceil \frac{\sigma \rho_m}{t_r^{max} - t_m^0} \rceil. \tag{9.57}$$

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

A Constraints Scheduler based on this analysis and an evaluation of the effectiveness of this scheduler are presented in [267].

9.10 Resource bundling; combinatorial auctions for cloud resources

Resources in a cloud are allocated in *bundles*; 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 an interest in economic models and, in particular, in auction algorithms. In the context of cloud computing, an auction is the allocation of resources to the highest bidder.

Combinatorial auctions. Auctions in which participants can bid on combinations of items or *packages* are called *combinatorial auctions* [118]; such auctions provide a relatively simple, scalable, and tractable solution to cloud resource allocation. Two recent combinatorial auction algorithms are the *Simultaneous Clock Auction* [36] and the *Clock Proxy Auction* [37]; the algorithm discussed in this section and introduced in [456] is called *Ascending Clock Auction (ASCA)*. In all these algorithms, the current price for each resource is represented by a "clock" seen by all participants at the auction.

We consider a strategy when prices and allocation are set as a result of an auction; in this auction, users provide bids for desirable bundles and the price they are willing to pay. We assume a population of U users, $u = \{1, 2, \dots, U\}$, and R resources, $r = \{1, 2, \dots, R\}$. The bid of user u is $\mathcal{B}_u = \{\mathcal{Q}_u, \pi_u\}$ with $\mathcal{Q}_u = (q_u^1, q_u^2, q_u^3, \dots)$ an R-component vector.

Each element of this vector, q_u^i, represents a bundle of resources a user u would accept and, in return, pay the total price π_u. Each vector component q_u^i is either a positive quantity and encodes the quantity of a resource desired, or if negative, it encodes the quantity of the resource offered. A user expresses her desires as an *indifference set* $\mathcal{I} = (q_u^1 \text{ XOR } q_u^2 \text{ XOR } q_u^3 \text{ XOR } \dots)$.

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

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

Pricing and allocation algorithms. A pricing and allocation algorithm partitions the set of users in two disjoint sets, winners and losers, denoted as $\mathcal{W}$ and $\mathcal{L}$, respectively; the algorithm should:

1. Be computationally tractable. Traditional combinatorial auction algorithms such as Vickey-Clarke-Groves (VLG) fail this criteria; 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 then the user is a winner; otherwise, the user is a loser.

Table 9.6 The constraints for a combinatorial auction algorithm.

$x_u \in \{0 \cup \mathcal{Q}_u\},\ \forall u$	a user gets all resources or nothing
$\sum_u x_u \leq 0$	final allocation leads to a net surplus of resources
$\pi_u \geq (x_u)^T p, \forall u \in \mathcal{W}$	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
$p \geq 0$	prices must be non-negative

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 $\max_{x,p} f(x,p)$. The constraints in Table 9.6 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 one expresses the fact that the system awards only available resources, only offered resources can be allocated; (c) the third one is that the bid of the winners exceeds the final price; (d) the fourth one states that the winners get the least expensive bundles in their indifference set; (e) the fifth one states that losers bid below the final price; (f) finally, the last one states that all prices are positive numbers.

Ascending Clock Auction combinatorial auction algorithm. Informally, the participants at the auction based on the ASCA algorithm [456] specify the resource and the quantities of that resource offered or desired at the price listed for that time slot. Then the *excess vector* is computed as follows:

$$z(t) = \sum_u x_u(t) \tag{9.58}$$

If all its components are negative, then auction stops; negative components mean that the demand does not exceed the offer. If the demand is larger than the offer, $z(t) \geq 0$, then the auctioneer increases the price for items with a positive excess demand and solicits bids at the new price. The algorithm satisfies conditions 1 through 6. All users discover the price at the same time and pay or receive a "fair" payment relative to uniform resource prices, the computation is tractable, and the execution time is linear in the number of participants at the auction and the number of resources. The computation is robust, 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 Fig. 9.9. These proxies can be modeled as functions which compute the "best bundle" from each $\mathcal{Q}_u$ set given the current price

$$\mathcal{Q}_u = \begin{cases} \hat{q}_u & \text{if } \hat{q}_u^T p \leq \pi_u \quad \text{with } \hat{q}_u \in \arg\min(q_u^T p) \\ 0 & \text{otherwise} \end{cases} \tag{9.59}$$

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 $\max(x, 0)$)

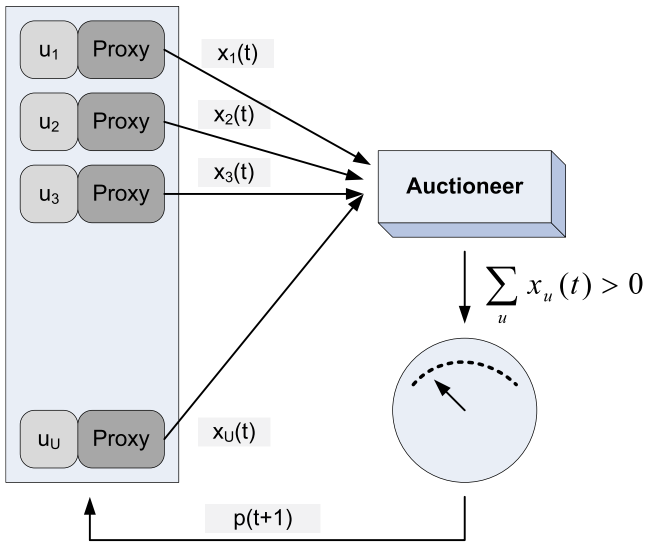

FIGURE 9.9

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

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

Pseudo code for the ASCA algorithm

```
set $t = 0$, $p(0) = \bar{p}$
  **loop**
    collect bids $x_u(t) = \mathcal{G}_u(p(t))$  $\forall u$
    calculate excess demand $z(t) = \sum_u x_u(t)$
    **if** $z(t) < 0$ **then**
      break
    **else**
      update prices $p(t+1) = p(t) + g(x(t), p(t))$
      $t \leftarrow t + 1$
    **end if**
  **end loop**
```

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 [456] 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 the 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.

Auctioning algorithms are very appealing because they support resource bundling and do 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, while 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 time. 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.

9.11 Cloud resource utilization and energy efficiency

According to Moore's Law, the number of transistors on a chip, thus, the computing power of microprocessors doubles approximately every 1.5 years, and electrical efficiency of computing devices doubles also about every 1.5 years [280]. Thus, performance growth rate and improvements in electrical efficiency almost cancel out. It follows that the energy use for computing scales linearly with the number of computing devices. The number of computing devices continues to grow and many are now housed in large cloud data centers.

The energy consumption of cloud data centers is growing and has a significant ecological impact. It also affects the cost of cloud services. The energy costs are passed on to the users of cloud services and differ from one country to another and from one region to another. For example, the costs for two AWS regions, US East and South America are: upfront for a year \$2 604 versus \$5 632 and hourly \$0.412 versus \$0.724, respectively. Higher energy and communication costs are responsible for the significant difference in this example; the energy costs for the two regions differ by about 40%.

All these facts justify the need to take a closer look at cloud energy consumption, a complex subject extensively discussed in the literature [9,30,45,51,52,331,494,499]. The topics to be covered are: how to define the energy efficiency, how energy-efficient are the processors, the storage devices, the networks, and the other physical elements of the cloud infrastructure, and what are the constraints and how well are these resources managed?

Cloud elasticity and over provisioning. One of the main appeals of utility computing is elasticity. *Elasticity* means that additional resources are guaranteed to be allocated when an application needs them and these resources will be released when they are no longer needed. The user ends up paying only for the resources it has actually used.

Over-provisioning means that a cloud service provider has to invest in a larger infrastructure than the *typical* cloud workload warrants. It follows that the average cloud server utilization is low [9,64,331]; see Fig. 9.10. The low server utilization affects negatively the common measure of energy efficiency,

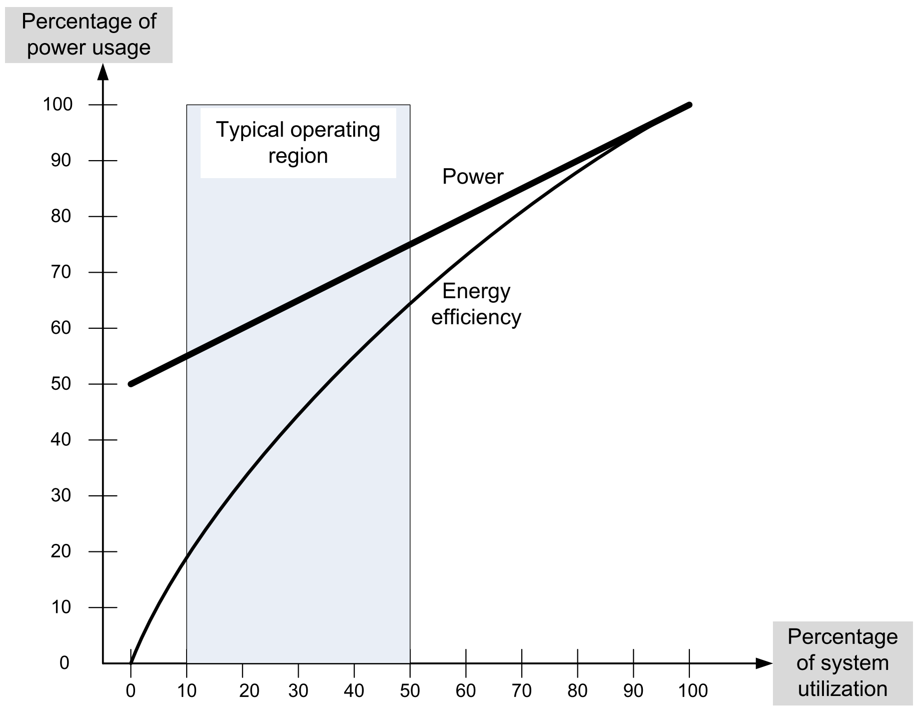

FIGURE 9.10

Even when power requirements scale linearly with the load, the energy efficiency of a computing system is not a linear function of the load; even when idle, a system may use 50% of the power corresponding to the full load. Data collected over a long period of time shows that the typical operating region for the servers at a data center is from about 10% to 50% of the load [51].

the performance per watt of power and the ecological impact of cloud computing. Overprovisioning is not economically sustainable [95].

Elasticity is based on overprovisioning and on the assumption that there is an effective admission control mechanism. Another assumption is that the likelihood of all running applications dramatically increasing their resource consumption at the same time is extremely low. This assumption is realistic, though we have seen cases when a system is overloaded due to concurrent access by large crowds, e.g., the phone system in case of a catastrophic event such as an earthquake. A possible solution is to ask cloud users to specify in their service request the type of workloads and to pay for access accordingly, e.g., a low rate for slow varying and a high rate for workloads with sudden peaks.

Energy efficiency and energy-proportional systems. An energy-proportional system consumes no power when idle, very little power under a light load and, gradually, more power as the load increases. By definition, an ideal energy-proportional system is always operating at 100% efficiency. Humans are a good approximation of an ideal energy proportional system; the human energy consumption is about 70 W at rest, 120 W on average on a daily basis, and can go as high as 1 000–2 000 W during a strenuous, short-time effort [51].

In real life, even systems whose power requirements scale linearly, when idle, use more than half the power consumed at full load [9]. Indeed, a 2.5 GHz Intel E5200 dual-core desktop processor with 2 GB of RAM consumes 70 W when idle and 110 W when fully loaded; a 2.4 GHz Intel Q6600 processor with 4 GB of RAM consumes 110 W when idle and 175 W when fully loaded [45].

Different subsystems of a computing system behave differently in terms of energy efficiency; while many processors have relatively good energy-proportional profiles, significant improvements in memory and disk subsystems are necessary. The processors used in servers consume less than one-third of their peak power at very low load and have a dynamic range of more than 70% of peak power; the processors used in mobile and/or embedded applications are better in this respect.

The dynamic power range[8] of other system components is narrower, less than 50% for DRAM, 25% for disk drives, and 15% for networking switches [51]. Power consumption of such devices is: 4.9 KW for a 604.8 TB, HP 8100 EVA storage server, 3.8 KW for the 320 Gbps Cisco 6509 switch, 5.1 KW for the 660 Gbps Juniper MX-960 gateway router [45].

The alternative to the wasteful resource management policy when the servers are *always on*, regardless of their load, is to develop *energy-aware load balancing and scaling* policies. Such policies combine *dynamic power management* with load balancing and attempt to identify servers operating outside their optimal energy regime and decide if and when they should be switched to a sleep state or what other actions should be taken to optimize the energy consumption.

Energy saving. The effort to reduce the energy use is focused on the computing, networking, and storage activities of a data center. A 2010 report shows that a typical Google cluster spends most of its time within the 10–50% CPU utilization range; there is a mismatch between server workload profile and server energy efficiency [9]. A similar behavior is also seen in the data center networks; these networks operate in a very narrow dynamic range; the power consumed when the network is idle is significant compared to the power consumed when the network is fully utilized.

A strategy to reduce energy consumption is to concentrate the workload on a small number of disks and allow the others to operate in a low-power mode. One of the techniques to accomplish this is based on replication. A replication strategy based on a sliding window is reported in [499]; measurement results indicate that it performs better than LRU, MRU, and LFU[9] policies for a range of file sizes, file availability, and number of client nodes and the power requirements are reduced by as much as 31%.

Another technique is based on data migration. The system in [229] uses data storage in virtual nodes managed with a distributed hash table; the migration is controlled by two algorithms: a short-term optimization algorithm used for gathering or spreading virtual nodes according to the daily variation of the workload so that the number of active physical nodes is reduced to a minimum, and a long-term optimization algorithm, used for coping with changes in the popularity of data over a longer period, e.g., a week.

A number of proposals have emerged for *energy proportional* networks [10]; the energy consumed by such networks is proportional with the communication load. For example, a data center interconnec-

[8] The dynamic range in this context is determined by the lower and the upper limit of the device power consumption. A large dynamic range means that the device is better, it is able to operate at a lower fraction of its peak power when its load is low.

[9] LRU (Least Recently Used), MRU (Most Recently Used), and LFU (Least Frequently Used) are replacement policies used by memory hierarchies for caching and paging.

tion network based on a flattened butterfly topology is more energy and cost efficient according to [9]. High-speed channels typically consist of multiple serial lanes with the same data rate; a physical unit is stripped across all the active lanes. Channels commonly operate plesiochronously[10] and are always on, regardless of the load, because they must still send idle packets to maintain byte and lane alignment across the multiple lanes. An example of an energy proportional network is *InfiniBand*, discussed in Section 6.7.

Many proposals argue that dynamic resource provisioning is necessary to minimize power consumption. Two main issues are critical for energy saving: the amount of resources allocated to each application and the placement of individual workloads. For example, a resource management framework combining a utility-based dynamic Virtual Machine provisioning manager with a dynamic VM placement manager to minimize power consumption and reduce Service Level Agreement violations is presented in [490].

Energy optimization is an important policy for cloud resource management but it cannot be considered in isolation, it has to be coupled with admission control, capacity allocation, load balancing, and quality of service. Existing mechanisms cannot support concurrent optimization of all the policies. Mechanisms based on a solid foundation such as control theory are too complex and do not scale well, those based on machine learning are not fully developed, and the others require a model of a system with a dynamic configuration operating in a fast-changing environment.

9.12 Resource management and dynamic application scaling

The demand for resources can be a function of the time of the 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 at the very beginning, 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, while access to a service provided by FEMA (Federal Emergency Management Agency) 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 one pays only for the resources it consumes are serious incentives to migrate to a public cloud. The question we address is how scaling can be actually implemented in a cloud when a very large number of applications exhibit this often unpredictable behavior [83,335,489]. 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 strategies, 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.

[10] Different parts of the system are almost, but not quite perfectly, synchronized; in this case, the core logic in the router operates at a frequency different from that of the I/O channels.

Horizontal scaling is the most common scaling strategy on a cloud; it is done by increasing the number of VMs as the load increases and reducing this number when the load decreases. Often, this leads to an increase of communication bandwidth consumed by the application. Load balancing among the running VMs is critical for 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, which distributes incoming requests of a transaction-oriented system to back-end servers.

An application should be designed to support scaling. Workload partitioning of a *modularly divisible* application is static as we have seen in Section 11.5. Static workload partitioning is decided a priori and cannot be changed thus, the only alternative is vertical scaling. The workload of an *arbitrarily divisible* application 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, and this justifies our statement that horizontal scaling is the most common scaling strategy.

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. Such a mapping 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 network traffic, a highly desirable effect from the perspective of an *arbitrarily divisible* application.

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

Several strategies to support scaling exist. *Automatic VM scaling* uses predefined metrics, e.g., CPU utilization to make scaling decisions. Automatic scaling requires *sensors* to monitor the state of the VMs and servers and *controllers* to make decisions based on the information about the state of the cloud. Controllers often use a state machine model for decision making. Amazon and Rightscale offer automatic scaling. AWS *CloudWatch* service supports applications monitoring and allows a user to set up conditions for automatic migrations. AWS *Elastic Load Balancing* service automatically distributes incoming application traffic across multiple *EC2*; *Elastic Beanstalk* allows dynamic scaling between a low and a high number of instances specified by the user; see Section 2.2. A cloud user usually has to pay for the more sophisticated scaling services such as *Elastic Beanstalk*.

Nonscalable or single load balancers are also used for horizontal scaling. Google uses machine learning to optimize resource utilization and minimize the risk of killing a task or performance degradation due to CPU throttling. The system called *Autoscaling* [426] configures automatically resources allocated to a job using horizontal and vertical scaling in an attempt to reduce the *slack*, i.e., the difference between user specified limits for CPU cores and RAM and the actual resource usage.

9.13 Control theory and optimal resource management (R)

Control theory has been used to design adaptive resource management for many classes of applications including power management [268], task scheduling [315], QoS adaptation in web servers [4], and load balancing [355,395]. 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 for 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, which satisfies stability and sensitivity constraints.

A technique to design self-managing systems based on concepts from control theory is discussed in [504]. This technique allows multiple QoS objectives and operating constraints to be expressed as a cost function. The technique can be applied to stand-alone or distributed web servers, database servers, high performance application servers, and embedded systems.

The following discussion considers a single processor serving a stream of input requests with the goal of minimizing a cost function reflecting the response time and the power consumption. The 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. An overview of control theory principles used for optimal resource allocation is presented next. Optimal control generates a sequence of control inputs over a look-ahead horizon, while estimating changes in operating conditions. A convex cost function has as 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 $u(i), u(i+1), \ldots, u(n-1)$ minimizing the expression

$$J(i) = \Phi(n, x(n)) + \sum_{k=i}^{n-1} L^k(x(k), u(k)) \tag{9.60}$$

where $\Phi(n, x(n)))$ is the cost function of the final step, n, and $L^k(x(k), u(k))$ is a time-varying cost function at the intermediate step k over the horizon $[i, n]$. The minimization is subject to the constraints $x(k+1) = f^k(x(k), u(k))$, where $x(k+1)$, the system state at time $k+1$, is a function of $x(k)$, the state at time k, and of $u(k)$, the input at time k; in general, the function f^k is time-varying, thus its superscript.

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

$$\Lambda(x, y, \lambda) = g(x, y) + \lambda \times [h(x, y) - k]. \tag{9.61}$$

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

$$\nabla_{x,y,\lambda}\Lambda(x, y, \lambda) = 0 \quad \text{or} \quad \left(\frac{\partial\Lambda(x, y, \lambda)}{\partial x}, \frac{\partial\Lambda(x, y, \lambda)}{\partial y}, \frac{\partial\Lambda(x, y, \lambda)}{\partial \lambda}\right) = 0. \tag{9.62}$$

The Lagrange multiplier at time step k is $\lambda(k)$ and we solve Eq. (9.62) as an unconstrained optimization problem. We define an adjoint cost function which includes the original state constrains as the Hamiltonian function H, then we construct the adjoint system consisting of the original state equation

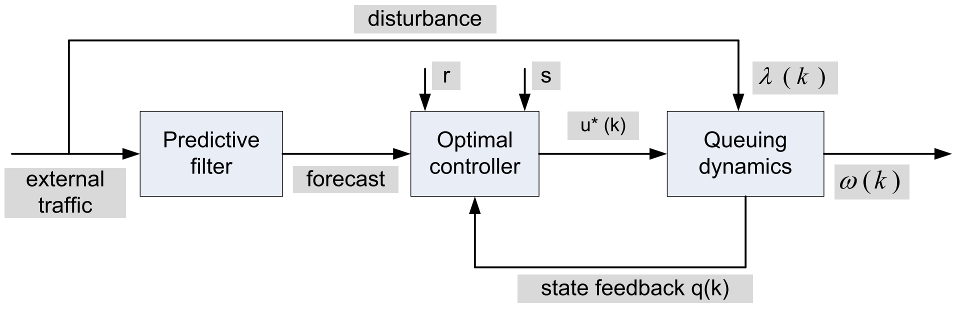

FIGURE 9.11

The structure of an optimal controller described in [504]; 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.

and the *costate equation*[11] governing the Lagrange multiplier. Thus, we define a two-point boundary problem[12]; the state x_k develops forward in time, while the costate occurs backward in time.

A model capturing both QoS and energy consumption for a single server system. We now 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 Fig. 9.11 uses the feedback regarding the current state and the 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 an FCFS basis. We do not assume a priori distributions of the arrival process and of the service process; instead, we use the estimate $\hat{\Lambda}(k)$ of the arrival rate $\Lambda(k)$ at time k. We also assume that the processor can operate at frequencies $u(k)$ in the range $u(k) \in [u_{min}, u_{max}]$ and call $\hat{c}(k)$ the time to process a request at time k when the processor operates at the highest frequency in the range, u_{max}. Then, we define the scaling factor $\alpha(k) = u(k)/u_{max}$ and we express an estimate of the processing rate $N(k)$ as $\alpha(k)/\hat{c}(k)$.

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

$$q(k+1) = \max\left\{\left[q(k) + \left(\hat{\Lambda}(k) - \frac{u(k)}{\hat{c}(k) \times u_{max}}\right) \times T_s\right], 0\right\} \tag{9.63}$$

[11] The costate equation is related to the state equation in optimal control.

[12] A boundary value problem has conditions specified at the extremes of the independent variable while an initial value problem has all of the conditions specified at the same value of the independent variable in the equation. The common case is when boundary conditions are supposed to be satisfied at two points—usually the starting and ending values of the integration.

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 wish to capture both the QoS and the energy consumption, as both affect the cost of providing the service. A utility function, such as the one depicted in Fig. 9.14, captures the rewards, as well as the penalties specified by the SLA for the response time. In the queuing model, the utility is a function of the size of the queue and can be expressed as a quadratic function of the response time

$$S(q(k)) = 1/2\left(s \times (\omega(k) - \omega_0)^2\right) \tag{9.64}$$

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

$$R(u(k)) = 1/2\left(r \times u(k)^2\right). \tag{9.65}$$

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

$$\Phi(q(N)) = 1/2\left(v \times q(n)^2\right). \tag{9.66}$$

The performance measure of interest is a cost expressed as

$$J = \Phi(q(N)) + \sum_{k=1}^{N-1} [S(q(k)) + R(q(k))]. \tag{9.67}$$

The problem is to find the optimal control u^* and the finite time horizon $[0, N]$ such that the trajectory of the system subject to optimal control is q^*, and the cost J in Eq. (9.67) is minimized subject to the following constraints

$$q(k+1) = \left[q(k) + \left(\hat{\Lambda}(k) - \frac{u(k)}{\hat{c}(k) \times u_{max}}\right) \times T_s\right], \quad q(k) \geq 0, \quad \text{and} \quad u_{min} \leq u(k) \leq u_{max}. \tag{9.68}$$

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

$$\Gamma 1: \ q(k) > 0, \quad \Gamma 2: \ u(k) \geq u_{min}, \quad \Gamma 3: \ u(k) \leq u_{max}, \tag{9.69}$$

then the pair $[q(\cdot), u(\cdot)]$ is called a *feasible state*. If the pair minimizes the Eq. (9.67) then the pair is *optimal*.

The Hamiltonian H in our example is

$$\begin{aligned} H = {} & S(q(k)) + R(u(k)) + \lambda(k+1) \times \left[q(k) + \left(\Lambda(k) - \frac{u(k)}{c \times u_{max}}\right) T_s\right] \\ & + \mu_1(k) \times (-q(k)) + \mu_2(k) \times (-u(k) + u_{min}) + \mu_3(k) \times (u(k) - u_{max}). \end{aligned} \tag{9.70}$$

According to Pontryagin's minimum principle,[13] the necessary condition for a sequence of feasible pairs to be optimal pairs is the existence of a sequence of costates λ and a Lagrange multiplier $\mu = [\mu_1(k), \mu_2(k), \mu_3(k)]$ such that

$$H(k, q^*, u^*, \lambda^*, \mu^*) \leq H(k, q, u^*, \lambda^*, \mu^*), \ \forall q \geq 0 \tag{9.71}$$

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

$$\mu_1(k) \geq 0, \ \mu_1(k)(-q(k)) = 0, \tag{9.72}$$

$$\mu_2(k) \geq 0, \ \mu_2(k)(-u(k) + u_{min}) = 0, \tag{9.73}$$

$$\mu_3(k) \geq 0, \ \mu_3(k)(u(k) - u_{max}) = 0. \tag{9.74}$$

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 [504].

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.

9.14 Stability of two-level resource allocation strategy (R)

The discussion in Section 9.13 shows that a server can be assimilated with a closed-loop control system and that we can apply theoretical control principles to resource allocation. We now discuss a two-level resource allocation architecture based on control theory concepts for the entire cloud; see Fig. 9.12. The automatic resource management is based on two levels of controllers, one for the service provider and one for the application.

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* which 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 in the output. If the change is too large, then 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 smaller 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.

[13] Pontryagin's principle is used in the optimal control theory to find the best possible control which leads a dynamic system transition from one state to another, subject to a set of constraints.

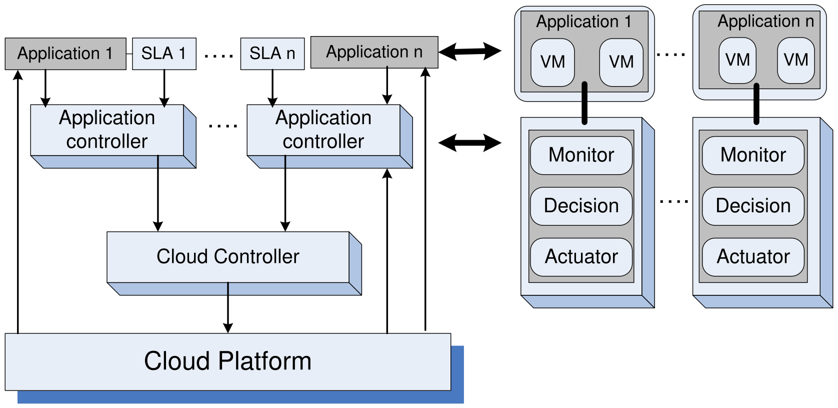

FIGURE 9.12

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

2. The granularity of the control, the fact that a small change enacted by the controllers leads to very large changes in the output.
3. Oscillations, when the changes in the input are too large and the control is too weak, such that the changes in 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 [156]. 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 only be carried out 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 an upper and a lower threshold are set, then 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 VMs; sometimes, allocation/deallocation of a single VM required by one of the thresholds may cause crossing the other threshold, another source of instability.

9.15 Feedback control based on dynamic thresholds (R)

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 if the system behavior must be changed, and, if so, it requests the *actuators* to 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 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 mixture 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, while *coarse control* means that the accuracy of these parameters is traded off 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 [306]. The questions addressed are:

- Is it beneficial to have two types of controllers: (1) *application controllers*, which determine if additional resources are needed, and (2) *cloud controllers*, which arbitrate requests for resources and allocate the physical resources?
- Is it feasible to consider *fine control*? Is *coarse 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 is it sufficient to define only a high threshold?

The first two questions are interrelated. It seems more appropriate to have two controllers, one with knowledge of the application and one 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, the service provider may wish to hide some of the information they have to simplify its resource management policies. For example, CSPs may not allow a VM to access information available to hypervisor-level sensors and actuators.

To answer the last two questions one has to define a measure of "goodness." In the experiments reported in [306], the parameter measured is the average CPU utilization, and a strategy is better than another if it reduces the number of requests made by the application controllers to add or remove VMs to the pool of those available to the application.

A control theoretical approach to address these questions is challenging. A pragmatic approach, qualitative arguments, and simulation results using a synthetic workload for a transaction-oriented application on a web server are reported in [306].

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

1. Compute the integral value of the high and the low threshold 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 3 VMs are: (a) dynamic thresholds perform better than the static ones and (b) two thresholds are better than one. While confirming 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.

9.16 Coordination of autonomic performance managers (R)

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 in a paper reporting on experiments carried out on a set of blades mounted on a chassis [268]. The experimental setup is shown in Fig. 9.13. 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 as a mechanism for energy saving; indeed, the energy dissipation scales quadratically with the supply voltage. The power management controls the CPU frequency, thus, the rate of instruction execution. For some compute-intensive workloads the performance decreases linearly with the CPU clock frequency, while for others the effect of lower clock frequency is less noticeable or non existent. The clock frequency of individual blades/servers is controlled by a power manager typically implemented in the firmware; it adjusts the clock frequency several times a second.

The approach to coordinating power and performance management in [268] 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
$$U_{pp}(R, P) = U(R) - \epsilon \times P \quad \text{or} \quad U_{pp}(R, P) = \frac{U(R)}{P}, \tag{9.75}$$
with $U(R)$ being the utility function based on response time only and ϵ being a parameter to weigh the influence of the two factors, response time and power.
- Identify a minimal set of parameters to be exchanged between the two managers.
- Set up a power cap for individual systems based on the utility-optimized power management policy.
- Use a standard performance manager modified only to accept input from the power manager regarding the frequency determined according to the power management policy. The power manager consists of TCL and C programs to compute the per-server (per-blade) power caps and send them

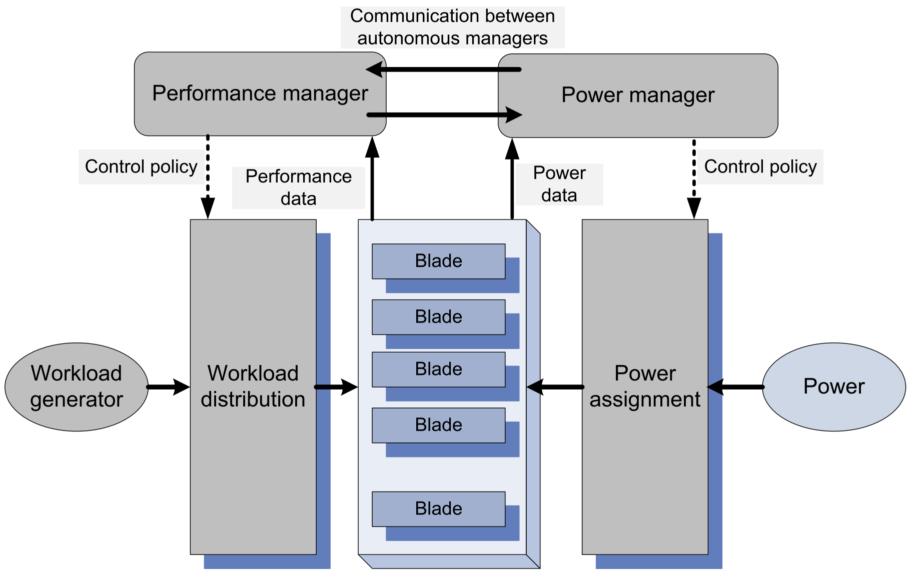

FIGURE 9.13

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.

via IPMI[14] to the firmware controlling the blade power. The power manager and the performance manager interact, but no negotiation between the two agents is involved.

- Use standard software systems. For example, use the WXD (WebSphere Extended Deployment), a 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 IBM Web Services Toolkit as a workload generator.

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

$$U'(p_\kappa, n_c) = U_{pp}(R(p_\kappa, n_c), P(p_\kappa, n_c)). \tag{9.76}$$

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

$$p_\kappa^{opt}(n_c) = \arg\max U'(p_\kappa, n_c). \tag{9.77}$$

[14] 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.

The hardware used for these experiments were the Goldensbridge blade with an Intel Xeon processor running at 3 GHz with 1 GB of level 2 cache and 2 GB of DRAM and with hyper threading enabled. A blade could serve 30 to 40 clients with a response time at or better than the 1 000 msec limit. When p_κ is lower than 80 Watts, the processor runs at its lowest frequency, 375 MHz, while for p_κ 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 power cap p_κ on n_c is derived via reinforcement-learning models.

The second type of experiments lead to the conclusion that both the response time and the power consumed are nonlinear functions of the power cap, p_κ, and the number of clients, n_c. More specifically, the conclusions of these experiments are: (i) at a low load, the response time is well below the target of 1 000 msec; (ii) at medium and high loads, the response time decreases rapidly when p_k increases from 80 to 110 wats; and (iii) the consumed power increases rapidly as the load increase for a given value of the power cap.

The machine learning algorithm used for the third type of experiments was based on the Hybrid Reinforcement Learning algorithm described in [474]. In the experiments using the machine learning model, the power cap required to achieve a response time lower than 1 000 msec for a given number of clients was the lowest when $\epsilon = 0.05$ and the first utility function given by Eq. (9.75) was used; for example, when $n_c = 50$ then $p_\kappa = 109$ Watts when $\epsilon = 0.05$, while $p_\kappa = 120$ when $\epsilon = 0.01$.

9.17 A utility model for cloud-based web services (R)

A *utility function* relates the "benefits" of a service to the "cost" to provide the service. For example, the benefit could be revenue and the cost could be the power consumption. In this section, we discuss a utility-based approach for autonomic management. The goal is to maximize the total profit computed as the difference between the revenue guaranteed by an SLA and the total cost to provide the services.

An SLA often specifies the rewards as well as penalties associated with specific performance metrics. Sometimes, the quality of service translates into average response time; this is the case of cloud-based web services when the SLA often specifies explicitly this requirement. For example, Fig. 9.14 shows the case when the performance metrics is R, the response time. The largest reward can be obtained when $R \le R_0$; a slightly lower reward corresponds to $R_0 < R \le R_1$; when $R_1 < R \le R_2$, 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 discussed in Section 9.13.

Formulated as an optimization problem, the solution discussed in [11] 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 $\mid K \mid$ 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 $v_k = v_k^{max}\left(1 - r_k / r_k^{max}\right)$; see Fig. 9.15(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, which models the think time of the user after the completion of service at one server and the start

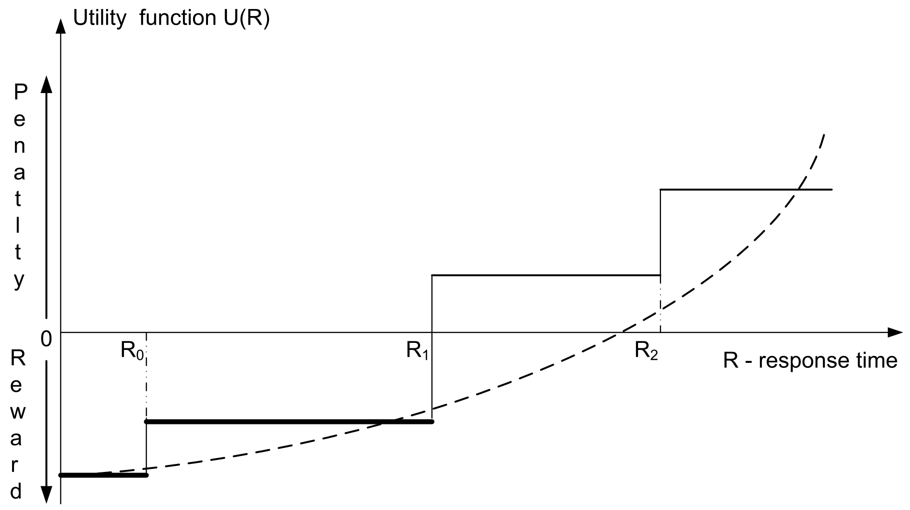

FIGURE 9.14

A 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. Dotted lines show a quadratic approximation of the utility function.

of processing at the next server; see Fig. 9.15(b). Upon completion, a class k request either completes with probability $1 - \sum_{k' \in K} \pi_{k,k'}$ 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 Λ_k the aggregate rate for class k, $\Lambda_k = \lambda_k + \sum_{k' \in K} \Lambda_{k'} \pi_{k,k'}$.

Typically, CPU and memory are considered as 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 DVFS model; the power consumption on a server is a function of the clock frequency. The scheduling of a server is work-conserving[15] and is modeled as a Generalized Processor Sharing (GPS) scheduling [536]. Analytical models [6], [384] are too complex for large systems.

The optimization problem formulated in [11] involves five terms: A and B reflect revenues, C is the cost for servers in a low power, stand-by mode, D is the cost of active servers given their operating frequency, E is the cost for switching servers from low-power, stand-by mode, to active state, and F is the cost for migrating VMs from one server to another. There are 9 constraints $\Gamma_1, \Gamma_2, \ldots, \Gamma_9$ for this mixed integer non-linear programming problem. The decision variables for this optimization problem are listed in Table 9.7 and the parameters used are shown in Table 9.8.

The expression to be maximized is:

$$(A + B) - (C + D + E + F) \tag{9.78}$$

[15] A scheduling policy is work conserving if the server cannot be idle while there is work to be done.

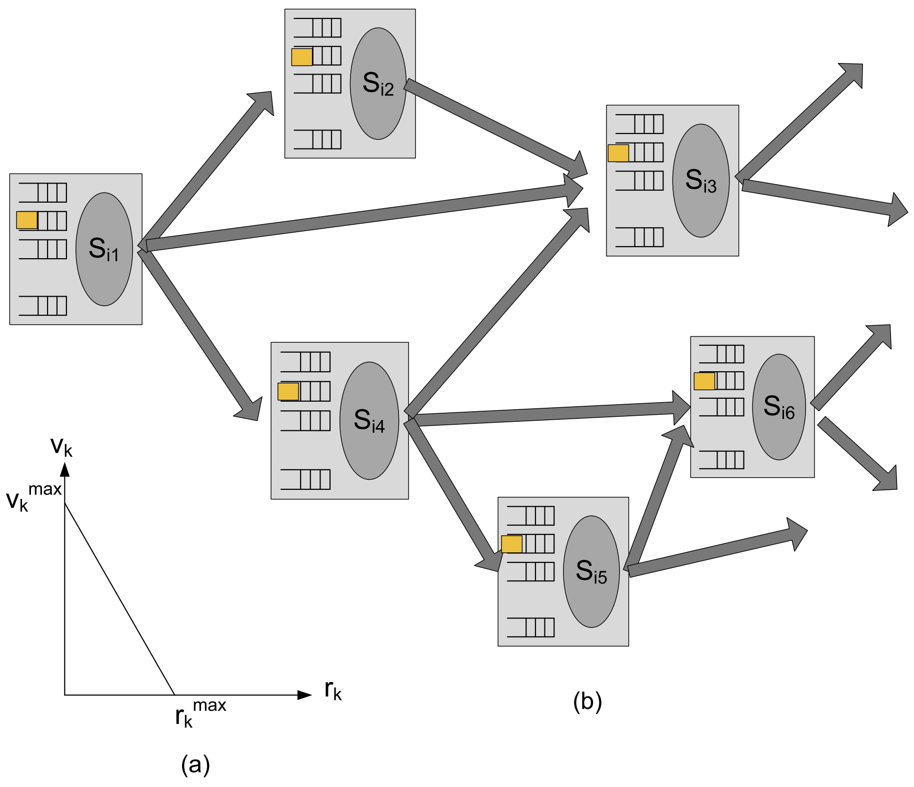

FIGURE 9.15

(a) 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) At each server S_i, there are $| K |$ queues for each one of $k \in K$ classes of requests. A tier consists of all requests of class $k \in K$ at all servers S_{ij}, $i \in I, 1 \leq j \leq 6$.

Table 9.7 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

with

$$A = \max \sum_{k \in K} \left(-m_k \sum_{i \in I, j \in N_k} \frac{\lambda_{i,k,j}}{\sum_{h \in H_i} (C_{i,h} \times y_{i,h})\, \mu_{k,j} \times \phi_{i,k,j} - \lambda_{i,k,j}} \right), \quad B = \sum_{k \in K} u_k \times \Lambda_k, \tag{9.79}$$

Table 9.8 The parameters used for the A, B, C, D, E, and F terms and the constraints Γ_i 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 VM from one server to another
cs_i	the cost for switching server i from the stand-by mode to an active state
$RAM_{k,j}$	the amount of main memory for tier j of class k request
$\overline{RAM}_i$	the amount of memory available on server i

$$C = \sum_{i \in I} \bar{c}_i, \quad D = \sum_{i \in I, h \in H_i} c_{i,h} \times y_{i,h}, \quad E = \sum_{i \in I} cs_i \max(0, x_i - \bar{x}_i), \tag{9.80}$$

and

$$F = \sum_{i \in I, k \in K, j \in N_j} cm \max(0, z_{i,j,k} - \bar{z}_{i,j,k}). \tag{9.81}$$

The nine constraints are:

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

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

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

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

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

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

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

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

(Γ_9) $\sum_{j=1}^{N_k} z_{i,k,j} \geq N_k \times w_{i,k}, \forall i \in I, k \in K$
$\lambda_{i,j,k}, \phi_{i,j,k} \geq 0, \forall i \in I, k \in K, j \in N_k$
$x_i, y_{i,h}, z_{i,k,j}, w_{i,k} \in \{0,1\}, \forall i \in I, k \in K, j \in N_k \quad \Rightarrow$ 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 unfeasible for a realistic cloud computing resource management strategy.

9.18 Cloud self-organization

Computer clouds are complex systems and should be analyzed in the context of the environment they operate in. The more diverse the environment, the more challenging is the cloud resource management. Cloud self-organization offers a glimpse of hope [332].

Two most important concepts for understanding complex systems are *emergence* and *self-organization*. *Emergence* lacks a clear and widely accepted definition, but it is generally understood as *a property of a system that is not predictable from the properties of individual system components.* There is a continuum of emergence spanning multiple scales of organization. Simple emergence occurs in systems at, or near thermodynamic equilibrium while complex emergence occurs only in nonlinear systems driven far from equilibrium [224].

Physical phenomena which do not manifest themselves at microscopic scales but occur at macroscopic scale are manifestations of emergence, e.g., temperature is a manifestation of microscopic behavior of large ensembles of particles. For such systems at equilibrium, the temperature is proportional to the average kinetic energy per degree of freedom. This is not true for ensembles of a small number of particles. Even the laws of classical mechanics can be viewed as limiting cases of quantum mechanics applied to large masses.

Emergence is critical for complex systems such as financial systems, the air-traffic system, and the power grid. The May 6, 2010, event when the Dow Jones Industrial Average dropped 600 points in a short period of time was a manifestation of emergence. The failure of the trading systems is attributed to interactions of trading systems developed independently and owned by organizations which work together but are motivated by self interest.

A recent paper [453] points out that dynamic coalitions of software-intensive systems used for financial activities pose serious challenges because there is no central authority, and there are no means to control the behavior of individual trading systems. The failures of the power grid (for example, the northeast blackout of 2003) can also be attributed to emergence. Indeed, during the first few hours of this event, the cause of the failure could not be identified due to the large number of independent systems involved. It was established only later that multiple causes, including the deregulation of the electricity market and the inadequacy of transmission lines of the power grid, contributed to this failure.

Informally, self-organization means synergetic activities of elements when no single element acts as a coordinator and the global patterns of behavior are distributed [191,441]. The intuitive meaning

Table 9.9 Attributes associated with self-organization and complexity.

Simple systems; no self-organization	Complex systems; self-organization
Mostly linear	Non-linear
Close to equilibrium	Far from equilibrium
Tractable at component level	Intractable at component level
One or few scales of organization	Many scales of organization
Similar patterns at different scales	Different patterns at different scales
Do not require a long history	Require a long history
Simple emergence	Complex emergence
Unaffected by phase transitions	Affected by phase transitions
Limited scalability	Scale-free

of self-organization is captured by the observation of Alan Turing [482]: "global order can arise from local interactions."

Self-organization is prevalent in nature; for example, in chemistry this process is responsible for molecular self-assembly, for self-assembly of monolayers, for the formation of liquid and colloidal crystals, and in many other instances. Spontaneous folding of proteins and other biomacromolecules, formation of lipid bilayer membranes, the flocking behavior of different species, creation of structures by social animals, are all manifestation of self-organization of biological systems. Inspired by biological systems, self-organization was proposed for organization of computing and communication systems [243], including sensor networks [328], for space exploration [239], and economical systems [284].

Generic attributes of complex systems exhibiting self-organization are summarized in Table 9.9. Nonlinearity of physical systems used to build computing and communication systems has countless manifestations and consequences. For example, when the clock rate of a microprocessor doubles, the power dissipation increases from $2^2 = 4$ to $2^3 = 8$ times, depending of the solid state technology used. This means that the heat removal system of much faster microprocessors has to use a different technology when we double the speed.

Nonlinearity is ultimately the reason why recently we have seen the clock rate of general-purpose microprocessors increasing only slightly. The increased number of transistors postulated by Moore's law is now used for multicore processors. This example illustrates also the so called *incommensurate scaling,* another attribute of complex systems. Incommensurate scaling means that when the size of the system, or when one of its important attributes such as speed increases, different system components are subject to different scaling rules.

The fact that computing and communication systems operate far from equilibrium is clearly illustrated by the traffic carried out by the Internet; there are patterns of traffic specific to the time of the day, but there is no steady-state. The many scales of the organization and the fact that there are different patterns at different scales is also clear in the Internet which is a collection of networks where, in turn, each network is also a collection of smaller networks, each one with its own specific traffic patterns.

The concept of *phase transition* comes from thermodynamics and describes the transformation, often discontinuous, of a system from one phase/state to another, as a result of a change in the environment. Examples of phase transitions are: *freezing*, transition from liquid to solid, and its reverse, *melting*; *deposition*, transition from gas to solid, and its reverse, *sublimation*; *ionization*, transition from gas to plasma, and its reverse, *recombination*.

Phase transitions can occur in computing and communication systems due to avalanche phenomena, when the process designed to eliminate the cause of an undesirable behavior leads to a further deterioration of the systems state. A typical example is thrashing due to competition among several memory-intensive processes that lead to excessive page faults.

Another example is an acute congestion which can cause a total collapse of a network; the routers start dropping packets, and, unless congestion avoidance and congestion control means are in place and operate effectively, the load increases as senders retransmit packets and the congestion increases. To prevent such phenomena some form of *negative feedback* has to be built into the system.

Scalability, the ability of the system to grow without affecting its global function(s), is a defining attribute of *self-organization*. Complex systems encountered in nature, or man-made, exhibit an intriguing property, *scale-free organization* [46,47]. This property reflects one of the few attributes of self-organization that can be precisely quantified.

Scale-free organization can be best explained in terms of the network model of the system, a random graph with vertices representing the entities and the links representing the relationships among them. In a scale-free organization the probability $P(m)$ that a vertex interacts with m other vertices decays as a power law $P(m) \approx m^{-\gamma}$ with γ a real number, regardless of the type and function of the system, the identity of its constituents and the relationships between them [67].

Empirical data available for social networks, power grids, the web, or the citation of scientific papers, confirm this trend. As an example of a social network, consider the collaborative graph of movie actors where links are present if two actors were ever cast in the same movie; in this case $\gamma \approx 2.3$. The power grid of the Western US has some 5 000 vertices representing power generating stations and in this case $\gamma \approx 4$.

The exponent of the World Wide Web scale-free network is $\gamma \approx 2.1$. This means that the probability that m pages point to one page is $P(m) \approx m^{-2.1}$ [47]. Recent studies indicate that $\gamma \approx 3$ for the citation of scientific papers. The larger the network, the closer a power law with $\gamma \approx 3$ approximates the distribution [46].

9.19 Cloud interoperability

Vendor lock-in is a concern, therefore cloud interoperability is a topic of great interest for the cloud community [314,337]. This section addresses several questions: What are the challenges? What is realistic to expect now? What could be done in the future for cloud interoperability? It makes only sense to discuss interoperability of PaaS and IaaS cloud delivery models; the expectation that SaaS services offered by one CSP will be offered by others, e.g., that Google's Gmail will be supported by Amazon, is not realistic.

Now, a workload can migrate from one server to another server in the same data center or among data centers of the same CSP. Migrating a workload to a different CSP is not feasible at this time. To use multiple clouds, data must be replicated and application binaries must be created for all targeted clouds. This is a costly proposition, thus, unfeasible in practice.

There is already a significant body of work on cloud standardization carried out at NIST, but it may take some time before the standards are adopted. First, cloud computing is a fast-changing field and early standardization would hinder progress and slowdown or stifle innovation. It is also likely that the CSPs will resist the standardization efforts.

CSPs are adamant to share information about internal specifications of the software stack, their policies, the mechanisms implementing these policies, and data formats. Each CSP is confident that such information gives it an advantage over the competition. There are also technical reasons why cloud interoperability poses a fair number of challenges, some insurmountable due to current limitations of computing and communication technologies.

So why a complex system such as the Internet is so successful while the development of an Intercloud, a worldwide organization allowing CSPs to share load is so challenging? The Internet is a network of networks, and its architecture is based on two simple ideas: (i) every communicating entity must be identified by an address, thus, a host at the periphery of the Internet, or a router at its core must have one or more IP address; (ii) data sent should be able to reach its destination in this maze of networks, therefore each network should route packets using the same protocol, the IP.

The function of a digital network, regardless of its physical substrate used for communication, is to transport bits of data regardless of what their provenance, music, voice, images, data collected by a sensor, text, or any other conceivable type of information. To make matters even easier, these bits can be packaged together in blocks of small, large, medium, or of any desirable size and can be repackaged whenever the need arises. All that matters is to deliver these bits from a source to a destination in a finite amount of time or in a very short time if the application so requires.

Things cannot be more different for a SuperCloud. First of all, most applications running on clouds need a large volume of data as input to produce results. Transferring say 1 TB of data over a 10 Gbps network takes 8×10^5 seconds, slightly less than a day. Increasing the network speed by an order of magnitude will still require a few hours to transfer this relatively modest volume of data for most cloud applications. Often, we need to transfer more than 1 TB and it is unlikely that Internet speeds of 100 Gbps will soon be available to connect data centers to one another.

What we expect from a SuperCloud is different from what is expected from the Internet where the only function required is to transport data and where all routers, regardless of their architecture, run software implementing the IP protocol. On the other hand, the spectrum of computations done on a cloud is extremely broad, and the heterogeneity of the cloud infrastructure cannot be ignored. Clouds use processors with different architectures, different configurations of cache, memory, and secondary storage, support different operating systems, and use different hypervisors.

The architecture of the server matters; one can only execute code on a server with the same ISA as the one the code was compiled for. The operating system running on a server matters because user code makes system calls to carry out privileged operations and a binary running under one OS cannot be migrated to another OS. The hypervisor running on the server matters because each hypervisor supports only a set of operating systems.

Fortunately, there is a glimpse of hope. A VM including the OS and the application can be migrated to a system with similar architecture and the same hypervisor. Nested virtualization, discussed in Section 5.10, allows a hypervisor to run another hypervisor and this idea discussed later in the section adds to the degrees of freedom for VM migration. Container technologies, such as Docker and LXC, are useful but one cannot move a Docker container from one host to another. What can be done to preserve data that an application has created inside the container is to commit the changes in the container to an image using Docker *commit*, move the image to a new host, and then start a new container with Docker *run*. Moreover, a Docker container is intended to run a single application. There are Docker containers to run an application such as MySQL. A new back-end Docker engine the *libcontainer* can run any application. LXC containers run an application under an instance of Linux.

9.20 Further readings

Cloud resource management poses new and extremely challenging problems, so there should be no surprise that this is a very active area of research. A fair number of papers, including [96], [432], and [27], are dedicated to different resource management policies. Several papers are concerned with QoS [171] and Service Level Agreements [199]. SLA-driven capacity management and SLA-based resource allocation policies are covered in [6] and [29], respectively. [169] analyzes performance monitoring for SLAs. Dynamic request scheduling of applications subject to SLA requirements is presented in [68]. Clouds QoS is analyzed in [68]. Semantic resource allocation using a multiagent system is discussed in [161].

The autonomic computing era is presented in [187]. Energy-aware resource allocation in autonomic computing is covered in [30]. Policies for autonomic computing based on utility functions are analyzed in [269]. Coordination of multiple autonomic managers and power-performance tradeoffs are dissected in [268]. Autonomic management of cloud services subject to availability guarantees is presented in [11]. The use of self-organizing agents for service composition in cloud computing is the subject of [219].

An authoritative reference on fault-tolerance is [38]; applications of control theory to resource allocation, discussed in [156] and [91], cover resource multiplexing in data centers. Admission control policies are discussed in [216]. Optimal control problems are analyzed in [226], and system testing is covered in [231]. Verification of performance assertion on the cloud is the subject of [278]. Power and performance management are the subject of [287] and performance management for cluster based web services is covered in [384]. Autonomic management of heterogeneous workloads is discussed in [467] and application placement controllers is the topic of [470]. Applications of pattern matching for forecasting on-demand resources needs is discussed in [88]. Economic models for resource allocations are covered in [327,330], and [431].

Scheduling and resource allocation are also covered by numerous papers: a batch queuing system on clouds with Hadoop and HBase is presented in [538]; data flow-driven scheduling for business applications is covered in [151]. Scalable thread scheduling is the topic of [516]. Reservation-based scheduling is discussed in [126]. Anti-caching in database management is the subject of [132]. Scheduling of real time services in cloud computing is presented in [310]. The OGF (Open Grid Forum) OCCI (Open Cloud Computing Interface) is involved in the definition of virtualization formats and APIs for IaaS. Flexible memory exchange, bag of tasks scheduling and distributed low latency scheduling are covered in [370], [377], and [382], respectively. Reference [421] analyzes capacity management for pools of resources.

9.21 Exercises and problems

Problem 1. Analyze the benefits and the problems posed by resource management implementations based on control theory, machine learning, utility-based, and market-oriented policies.

Problem 2. Can optimal strategies for the five classes of policies, admission control, capacity allocation, load balancing, energy optimization, and QoS guarantees be actually implemented in a cloud? The term "optimal" is used in the sense of control theory. Support your answer with solid arguments. Optimal strategies for one may be in conflict with optimal strategies for one or more of the other classes. Identify and analyze such cases.

Problem 3. Analyze the relationship between the scale of a system and policies and mechanisms for resource management. Consider also the geographic scale of the system.

Problem 4. What are the limitations of the control theoretic approach discussed in Section 9.13? Do the approaches discussed in Sections 9.14 and 9.15 remove or relax some of these limitations? Justify your answers.

Problem 5. Multiple controllers are probably necessary due to the scale of the cloud. Is it beneficial to have system and application controllers? Should the controllers be specialized? For example, some to monitor performance, others to monitor power consumption. Should all the functions we want to base the resource management policies on be integrated in a single controller and one such controller be assign to a given number of servers, or to a geographic region? Justify your answers.

Problem 6. In a scale-free network, the degrees of the nodes have an exponential distribution. A scale-free network could be used as a virtual network infrastructure for cloud computing. *Controllers* represent a dedicated class of nodes tasked with resource management; in a scale-free network nodes with a high connectivity can be designated as controllers. Analyze the potential benefit of such a strategy.

Problem 7. Use the start-time fair queuing (SFQ) scheduling algorithm to compute the virtual startup and the virtual finish time for two threads a and b with weights $w_a = 1$ and $w_b = 5$ when the time quantum is $q = 15$ and thread b blocks at time $t = 24$ and wakes up at time $t = 60$. Plot the virtual time of the scheduler function of the real time.

Problem 8. Apply the borrowed virtual time (BVT) scheduling algorithm to the problem in Example 2 of Section 9.7 but with a time warp of $W_c = -30$.

Problem 9. In Section 9.11, we introduced the concept of energy-proportional systems, and we saw that different system components have different dynamic ranges. Sketch a strategy to reduce the power consumption in a lightly loaded cloud and discuss the steps for placing a computational server in a standby mode and then for bringing it back up to an active mode.

Problem 10. Overprovisioning is the reliance on extra capacity to satisfy the needs of a large community of users when the average-to-peak resource demand ratio is very high. Give an example of a large-scale system using overprovisioning and discuss if overprovisioning is sustainable in that case and what are the limitations of it. Is cloud elasticity based on overprovisioning sustainable? Give the arguments to support your answer.