UNIT V
CLOUD RESOURCE MANAGEMENT AND SCHEDULING

Policies and Mechanisms for Resource Management, Applications of Control Theory to Task Scheduling on a Cloud, Stability of a Two-Level Resource Allocation Architecture, Feedback Control Based on Dynamic Thresholds. Coordination of Specialized Autonomic Performance Managers, Resource Bundling, Scheduling Algorithms for Computing Clouds, Fair Queuing, Start Time Fair Queuing Borrowed Virtual Time Cloud Scheduling Subject to Deadlines , Scheduling Map Reduce Applications Subject to Deadlines.

Resource management is a core function of any man-made system. It affects the three basic criteria for the evaluation of a system: performance, functionality, and cost. An inefficient resource management has a direct negative effect on performance and cost and an indirect effect on the functionality of a system. A cloud is a complex system with a very large number of shared resources subject to unpredictable requests and affected by external events it cannot control. Cloud resource management requires complex policies and decisions for multi-objective optimization. Cloud resource management is extremely challenging because of the complexity of the system, which makes it impossible to have accurate global state information, and because of the unpredictable interactions with the environment. The strategies for resource management associated with the three cloud delivery models, *IaaS*, *PaaS*, and *SaaS*, differ from one another.

POLICIES AND MECHANISMS FOR RESOURCE MANAGEMENT

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

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

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

Admission control: The explicit goal of an admission control policy is to prevent the system from accepting workloads in violation of high-level system policies.
Capacity allocation: Capacity allocation means to allocate resources for individual instances; an instance is an activation of a service.
Load balancing and Energy optimization: The common meaning of the term *load balancing* is that of evenly distributing the load to a set of servers. **Load** balancing and energy optimization can be done locally, but global load-balancing and energy optimization policies encounter the same difficulties.
Quality-of-service (QoS) guarantees: The performance models are very complex, analytical solutions are intractable, and the monitoring systems used to gather state information for these models can be too intrusive and unable to provide accurate data. Many techniques are concentrated on system performance in terms of throughput and time in system, but they rarely include energy tradeoffs or QoS guarantees.

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

Control theory. Control theory uses the feedback to guarantee system stability and predict transient behavior , but can be used only to predict local rather than global behavior.
Machine learning: This technique could be applied to coordination of several autonomic system managers; they do not need a performance model of the system.
Utility-based: Utility-based approaches require a performance model and a mechanism to correlate user-level performance with cost.
Market-oriented/economic mechanisms*:* Such mechanisms do not require a model of the system, e.g., combinatorial auctions for bundles of resources.

APPLICATIONS OF CONTROL THEORY TO TASK SCHEDULING ON A CLOUD

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

Control Theory Principles: Optimal control generates a sequence of control inputs over a look-ahead horizon while estimating changes in operating conditions. A convex cost function has arguments *x(k)*, the state at step *k*, and *u(k)*, the control vector; this cost function is minimized, subject to the constraints imposed by the system dynamics. The discrete-time optimal control problem is to determine the sequence of control variables *u(i)*, *u(i* + 1*)*, . . . , *u(n* − 1*)* to minimize the expression

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

Where **ϕ(n, x(n))** is the cost function of the final step, *n*, and *Lk (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))*,

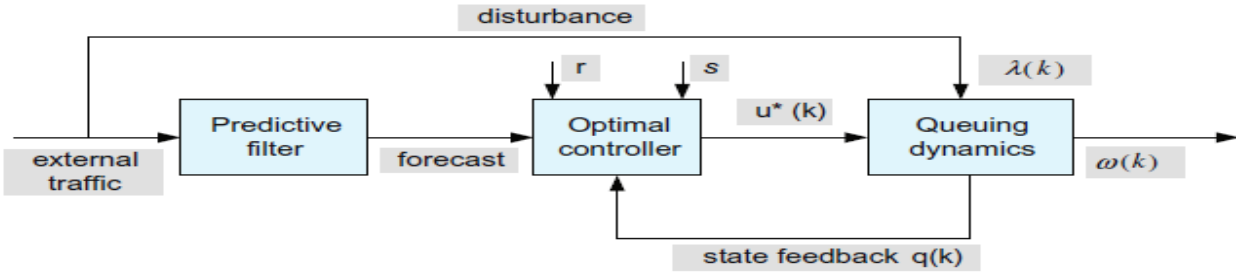


Fig: The structure of an optimal controller

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

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

A Model Capturing Both QoS and Energy Consumption for a Single-Server System

The behavior of a single processor is modeled as a nonlinear, time-varying, discrete-time state equation. If *Ts* 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

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

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

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

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

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

STABILITY OF TWO LEVEL RESOURCE ALLOCATION ARCHITECTURE

Two-level resource allocation architecture based on control theory concepts for the entire cloud. The automatic resource management is based on two levels of controllers, one for the service provider and one for the application. The main components of a control system are the inputs, the control system components, and the outputs. The system components are ***sensors*** used to estimate relevant measures of performance and ***controllers*** that implement various policies; the output is the resource allocations to the individual applications.

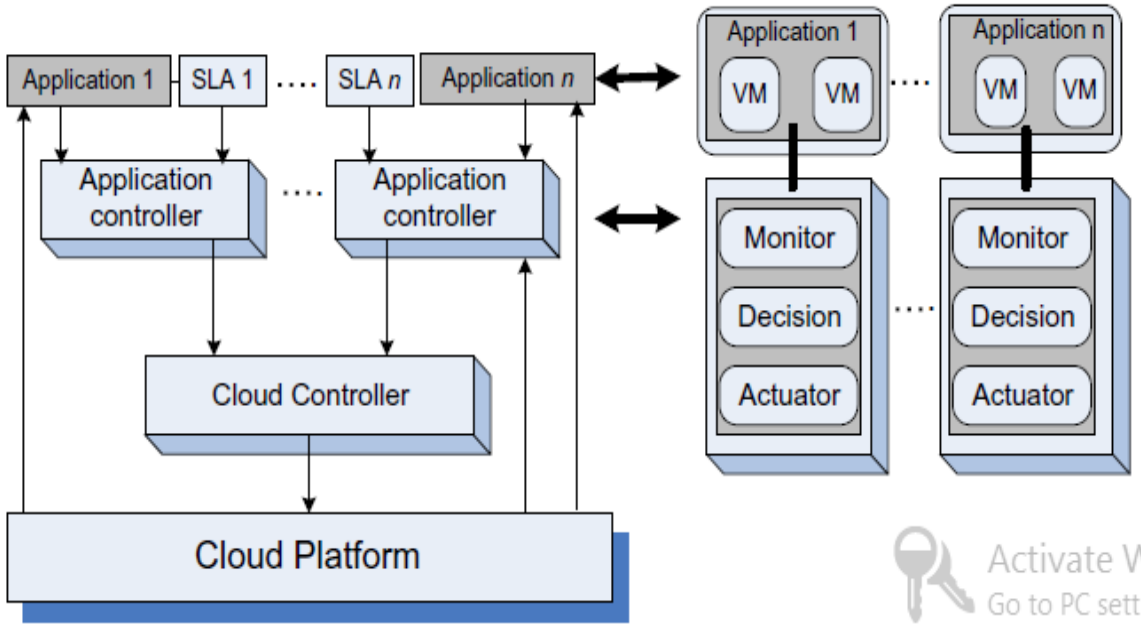


Fig: two-level control architecture.

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

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

2. The granularity of the control, the fact that a small change enacted by the controllers leads to very large changes of the output.

3. Oscillations, which occur when the changes of the input are too large and the control is too weak, such that the changes of the input propagate directly to the output.

Two types of policies are used in autonomic systems:

(i) Threshold-based policies and

(ii) Sequential decision policies based on Markovian decision models.

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

FEEDBACK CONTROL BASED ON DYNAMIC THRESHOLDS

Sensors, monitors and actuators are the elements which are involved in the control systems, where the **sensors** measure the parameters of interest, and then transmit the measures values to a **monitor**¸ which determines whether the system behavior must be changed; it requests that the ***actuators*** carry out the necessary actions.

Thresholds: A ***threshold*** is the value of a parameter related to the state of a system that triggers a change in the system behavior. Thresholds are used in control theory to keep critical parameters of a system in a predefined range. The threshold could be ***static***, defined once and for all, or it could be ***dynamic*.** A dynamic threshold could be based on an average of measurements carried out over a time interval, a so-called ***integral control*.** The dynamic threshold could also be a function of the values of multiple parameters at a given time or a mix of the two. a high and a low threshold are defined in order to maintain the system parameters in the given range.

Proportional Thresholding: it can be discussed as follows with the help of two question,

• Is it beneficial to have two types of controllers ?

(1) ***Application controllers*** that determine whether additional resources are needed and

(2) ***Cloud controllers*** that arbitrate requests for resources and allocate the physical resources?

• Is it feasible to consider *fine control*.

The **importance of the proportional thresholding** is captured by the following algorithm:

1. Compute the integral value of the high and the low thresholds as averages of the maximum and, respectively, the minimum of the processor utilization over the process history.

2. Request additional VMs when the average value of the CPU utilization over the current time slice exceeds the high threshold.

3. Release a VM when the average value of the CPU utilization over the current time slice falls below the low threshold.

The conclusions obtained based on experiments with three VMs are as follows:

(a) Dynamic thresholds perform better than static ones and

(b) Two thresholds are better than one

COORDINATION OF SPECIALIZED AUTONOMIC PERFORMANCE MANAGERS

Virtually all modern processors support dynamic voltage scaling (DVS) as a mechanism for energy saving. Indeed, the energy dissipation scales quadratically with the supply voltage. The power management controls the CPU frequency and, thus, the rate of instruction execution. For some compute-intensive workloads the performance decreases linearly with the CPU clock frequency, whereas for others the effect of lower clock frequency is less noticeable or nonexistent. The clock frequency of individual. Blades/servers is controlled by a power manager, typically implemented in the firmware; it adjusts the clock frequency several times a second. The approach to coordinating power and performance management i is based on several ideas:

• Use a joint utility function for power and performance. The joint performance-power utility function,

- *Upp(R, P)*, is a function of the response time, *R*, and the power, *P*, and it can be of the form

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

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.

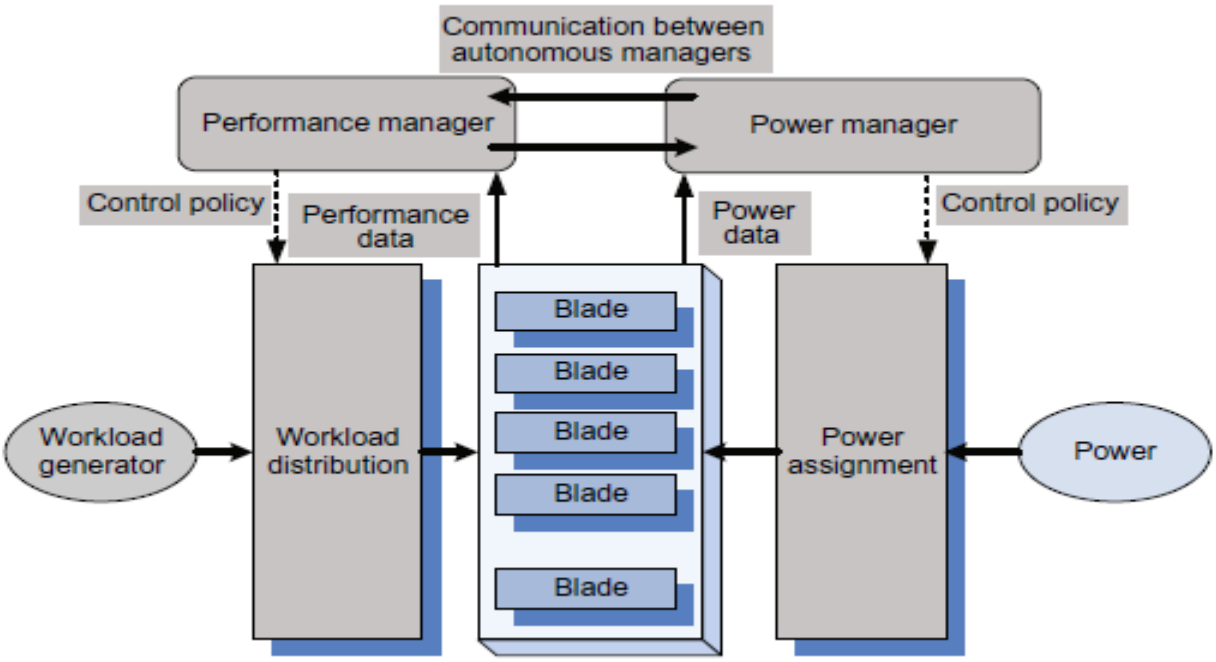


fig: Experimental support for performance mangers

Use a standard performance manager modified only to accept input from the power manager regarding the frequency determined according to the power management policy.
Use standard software systems

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

RESOURCE BUNDLING: Combinatorial auctions for cloud resources

Resources in a cloud are allocated in ***bundles***, allowing users get maximum benefit from a specific combination of resources. Indeed, along with CPU cycles, an application needs specific amounts of main memory, disk space, network bandwidth, and so on. Resource bundling complicates traditional resource allocation models and has generated interest in economic models and, in particular, auction algorithms.

Combinatorial Auctions. Auctions, in which participants can bid on combinations of items, or *packages*, are called *combinatorial auctions*. Such auctions provide a relatively simple, scalable, and tractable solution to cloud resource allocation. Two recent combinatorial auction algorithms are the ***simultaneous clock auction*** and the ***clock proxy auction.***

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

1. Be computationally tractable. Traditional combinatorial auction algorithms such as Vickey-Clarke- Groves (VLG) fail these criteria, because they are not computationally tractable.

2. Scale well. Given the scale of the system and the number of requests for service, scalability is a necessary condition.

3. Be objective. Partitioning in winners and losers should only be based on the price πu of a user's bid. If the price exceeds the threshold, the user is a winner; otherwise the user is a loser.

4. Be fair.Make sure that the prices are *uniform*. All winners within a given resource pool pay the same price.

5. Indicate clearly at the end of the auction the unit prices for each resource pool.

6. Indicate clearly to all participants the relationship between the supply and the demand in the system.

The function to be maximized is

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

The ASCA Combinatorial Auction Algorithm: Informally, in the ASCA algorithm the participants at the auction specify the resource and the quantities of

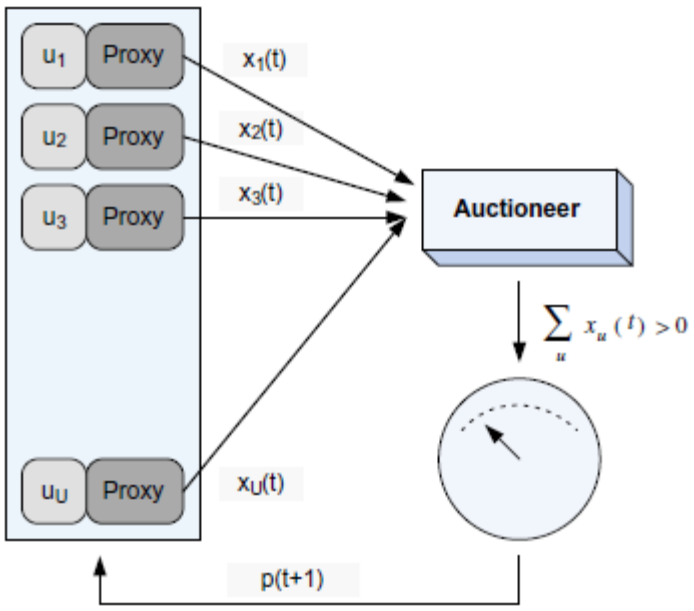
that resource offered or desired at the price listed for that time slot. Then the ***excess vector.***

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

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

There is a slight complication as the algorithm involves user bidding in multiple rounds. To address this problem the user proxies automatically adjust their demands on behalf of the actual bidders, These proxies can be modeled as functions that compute the "best bundle" from each *Qu* set given the current price.

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



An auctioning algorithm is very appealing because it supports resource bundling and does not require a model of the system. At the same time, a practical implementation of such algorithms is challenging. First, requests for service arrive at random times, whereas in an auction all participants must react to a bid at the same time. Periodic auctions must then be organized, but this adds to the delay of the response. Second, there is an incompatibility between cloud elasticity,

which guarantees that the demand for resources of an existing application will be satisfied immediately, and the idea of periodic auctions.
The above **fig. shows** the schematics of the ASCA algorithm.

SCHEDULING ALGORITHMS FOR COMPUTING CLOUDS

Scheduling is a critical component of cloud resource management. Scheduling is responsible for resource sharing/multiplexing at several levels. A server can be shared among several virtual machines, each virtual machine could support several applications, and each application may consist of multiple threads.
CPU scheduling supports the virtualization of a processor, the individual threads acting as virtual processors; a communication link can be multiplexed among a number of virtual channels, one for each flow.

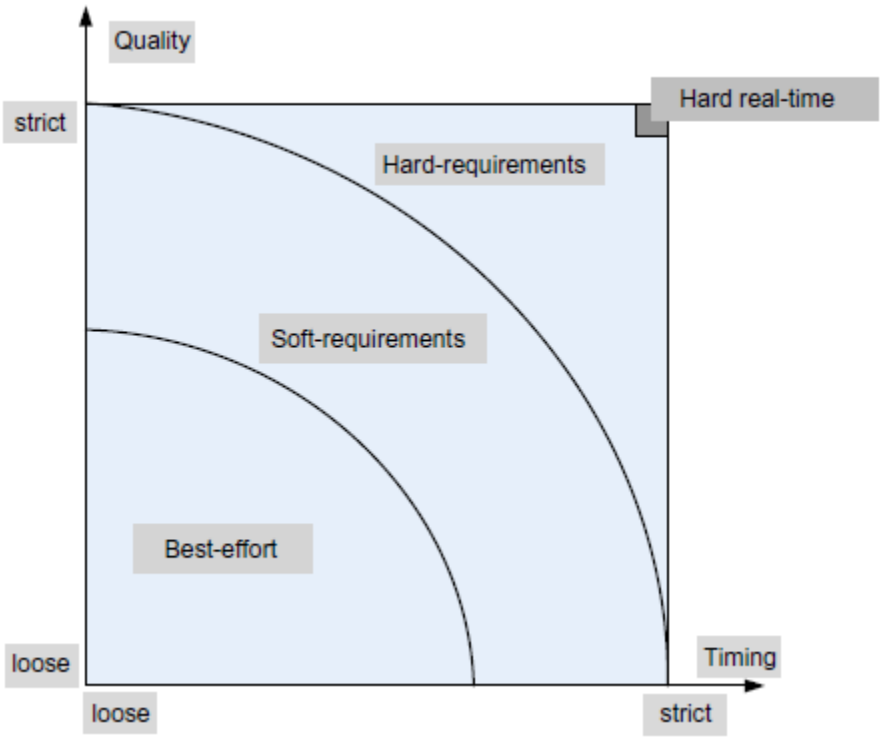
The objectives of a scheduler for a batch system are to maximize the throughput (the number of jobs completed in one unit of time, e.g., in one hour) and to minimize the turnaround time (the time between job submission and its completion). For a real-time system the objectives are to meet the deadlines and to be predictable. Schedulers for systems supporting a mix of tasks – some with hard real-time constraints, others with soft, or no timing constraints – are often subject to contradictory requirements. Some schedulers are *preemptive*, allowing a high-priority task to interrupt the execution of a lower-priority one; others are *non preemptive.*
Two distinct dimensions of resource management must be addressed by a scheduling policy:
(a) The amount or quantity of resources allocated and
(b) The timing when access to resources is granted

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, *Bi* _ *bi* .
*C*2**.** If the minimum allocation of any user is *Bmin* no allocation satisfying condition *C*1 has a higher *Bmin* than the current allocation.

*C*3**.** When we remove the user receiving the minimum allocation *Bmin* and then reduce the total amount of the resource available from *B* to *(B − Bmin)*, the condition *C*2 remains recursively true

from t_1 to t_2 of two runnable threads a and b, $\Omega_a(t_1, t_2)$ and $\Omega_b(t_1, t_2)$, respectively, minimize the expression

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



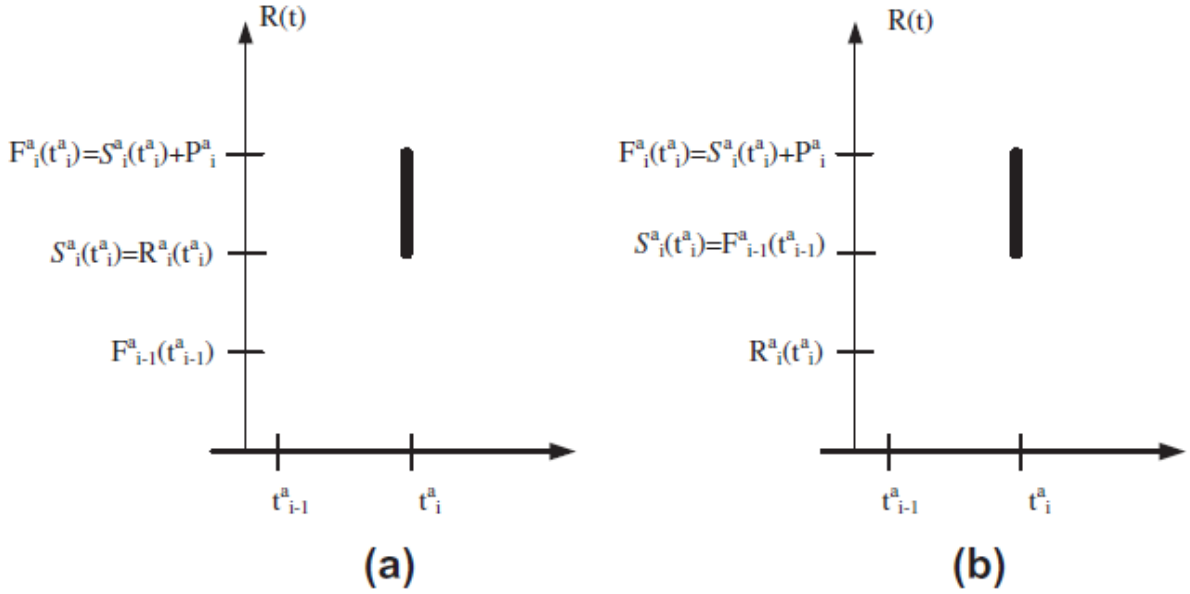
FAIR QUEUING

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

There arises two problems here with respect to ***bandwidth and buffer space*** its solution can be given by using FCFS, but algorithm does not guarantee fairness;

Greedy flow sources can transmit at a higher rate and benefit from a larger share of the bandwidth. To address this problem, a fair queuing algorithm proposed This algorithm guarantees the fairness of buffer space management, but does not

guarantee fairness of bandwidth allocation. Indeed, a flow transporting large packets will benefit from a larger bandwidth .



The *fair queuing (FQ)* algorithm proposes a solution to this problem. First, it introduces a *bit-by-bit round-robin (BR)* strategy; as the name implies, in this rather impractical scheme a single bit from each queue is transmitted and the queues are visited in a round-robin fashion.

Let *R(t)* be the number of rounds of the BR algorithm up to time *t* and *Nactive(t)* be the number of active flows through the switch. When the first and the last bit, respectively, of the packet *i* of flow *a* are transmitted. Then

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

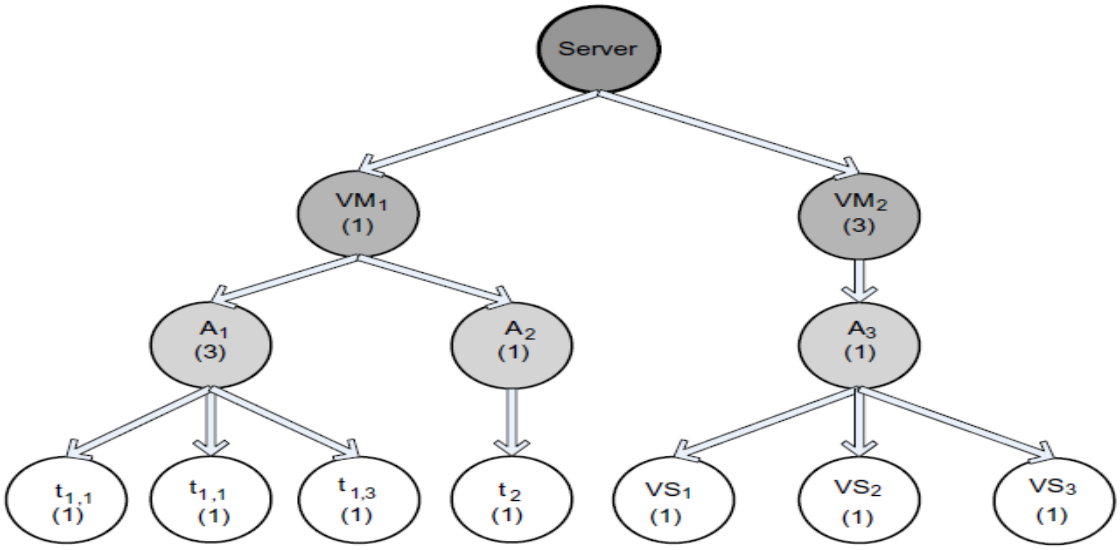
The quantities $R(t)$, $N_{active}(t)$, S_i^a, and F_i^a depend only on the arrival time of the packets, t_i^a, and not on their transmission time, provided that a flow a is active as long as

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

START TIME FAIR QUEUING

The basic idea of the ***start-time fair queuing (SFQ)*** algorithm is to organize the consumers of the CPU bandwidth in a tree structure; the root node is the processor and the leaves of this tree are the threads of each application. A scheduler acts at each level of the hierarchy. The fraction of the processor bandwidth, *B*, allocated to the intermediate node *i* is, with *wj* , $1 \leq j \leq n$, the weight of the *n* children of node *i*

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



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

An SFQ scheduler follows several rules:

R1. The threads are serviced in the order of their virtual start-up time; ties are broken arbitrarily.

R2. The virtual startup time of the *i* -th activation of thread *x* is

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

BORROWED VIRTUAL TIME

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

Thread *i* has an *effective virtual time*, *Ei*, an *actual virtual time*, *Ai* , and a *virtual time warp*, *Wi* . The scheduler thread maintains its own *scheduler virtual time (SVT)*, defined as the minimum actual virtual time *Aj* of any thread. The threads are dispatched in the order of their effective virtual time, Ei, a policy called the earliest virtual time (EVT).

The virtual time warp allows a thread to acquire an earlier effective virtual time

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

The algorithm measures the time in *minimum charging units* (*mcu*) and uses a time quantum called *context switch allowance (C)*, which measures the real time a thread is allowed to run when competing with other threads, measured in multiples of *mcu*. Typical values for the two quantities are *mcu* = 100 *μsec* and *C* = 100 *msec*. A thread is charged an integer number of *mcu*.

each thread *τi* maintains a constant *ki* and uses its weight *wi* to compute the amount _ used to advance its actual virtual time upon completion of a run.

$$A_i \leftarrow A_i + \Delta.$$

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

CLOUD SCHEDULING SUBJECT TO DEADLINES

SLA specifies when the results of computations done in the cloud are available,

Task Characterization and Deadlines: Real time applications involve periodic or a periodic tasks with deadlines, A task is characterized by a tuple ($A_{i,}$ σ_i, D_i) where $A_{i,}$ is arrival time

σ_i, >0 is the data size of the task

D_i is the relative deadline.

There also exist ***hard deadlines and soft deadlines***

In hard deadline 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 a strict and expressed precisely as milliseconds or possible in seconds

In soft deadlines there are no penalties if soft deadlines are missed by fractions of units used to express them.

System model: 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, . . . , *Sn*. The system is homogeneous, all workers are identical, and the communication time from the head node to any worker node is the same, here there are two important problems

- the order of execution of the tasks Πi
- The wordload partitioning and task mapping to worker nodes.

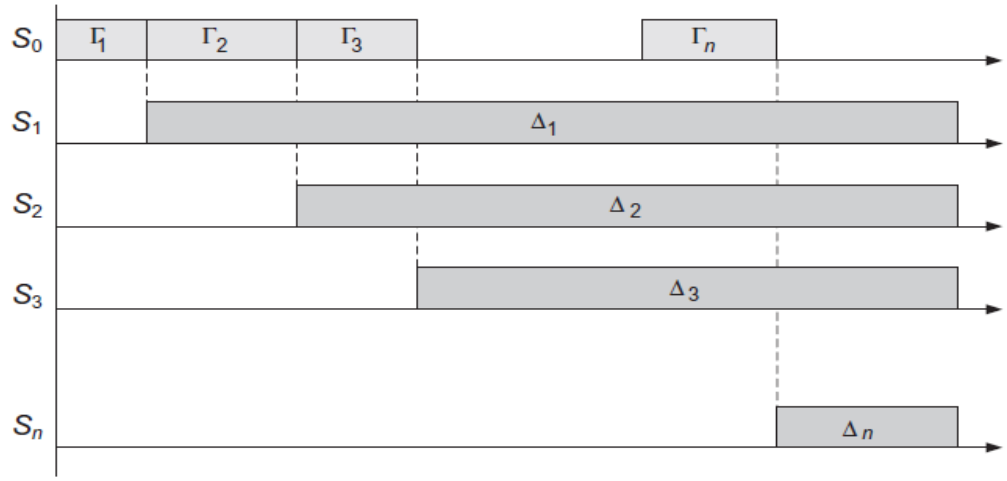
Scheduling policies: the most common scheduling policies are used to determine the order Of Execution Of The Tasks Is:

- First In First Out (Fifo) : Here The Tasks Are Scheduled For Execution In Order Of Their Arrival.
- Earliest Deadline (Edf) : Here The Task With Earliest Deadline Is Scheduled First

There exist two kinds of portioning rule which are

Optimal partitioning rule (OPR) : OPR refers to the execution times, where the workload is partitioned to ensure the earliest possible completion time and all the other tasks are required.

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



Equal Partitioning Rule (ERP) : ERP assigns an equal workload to individual worker nodes

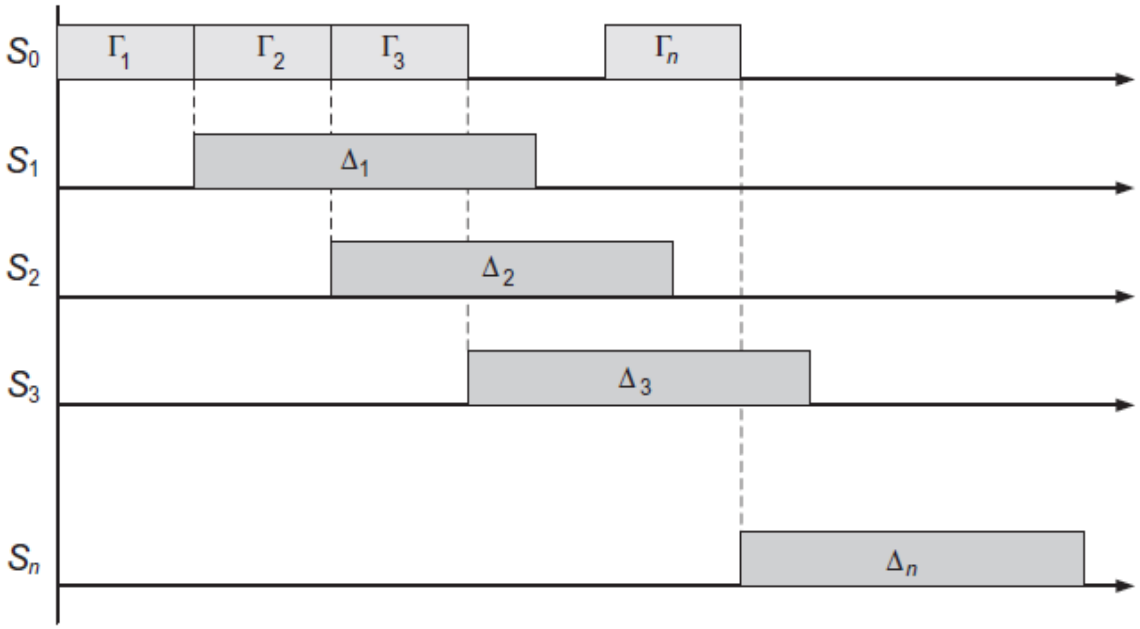
$$\mathcal{E}(n,\sigma) = \sum_{i=1}^{n} \Gamma_i + \Delta_n = n \times \frac{\sigma}{n} \times \tau + \frac{\sigma}{n} \times \rho = \sigma \times \tau + \frac{\sigma}{n} \times \rho.$$

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

$$t_0 + \sigma \times \tau + \frac{\sigma}{n} \times \rho \leqslant A + D \text{ or } n \geqslant \frac{\sigma \times \rho}{A + D - t_0 - \sigma \times \tau}.$$

Thus,

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



SCHEDULING MAP REDUCE APPLICATIONS SUBJECT TO DEADLINES

There exist several options for scheduling Apache Hadoop of the Map Reduce algorithm are

- The default FIFO Schedule
- The Fair Scheduler
- The Capacity Scheduler
- The Dynamic Proportional Scheduler

Two assumptions are made for the initial derivation

- the systems is homogeneous, which means that ρ_m and ρ_r which indicate the cost of processing a unit data by the map and the reduce task respectively
- load equipartition

Under these conditions the duration of the job J with the input of size σ is

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

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

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

Then the maximum value for the start up time of the reduce task is

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

Then the expression of maximum value for the start up time of the reduce task with respect to meeting the deadline is given as

$$t_m^0 + \sigma \frac{\rho_m}{n_m} \leqslant t_r^{max}$$

Now applying the condition for minimum number of slots as n_m^{min} should satisfy the condition

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