Module 4
Cloud Resource Management and Scheduling

Visualize a Cloud Office:

1. Admission Control: The receptionist decides who can enter the office (accept tasks).
2. Capacity Allocation: The manager assigns desks (resources) to employees (tasks).
3. Load Balancing: The team lead ensures all employees have equal workloads.
4. Energy Optimization: The facility team turns off lights when not needed (saves energy).
5. Quality of Service (QoS): The client manager ensures all client agreements are met.

Policies and mechanisms for resource management

Cloud resource management policies:

Policies	
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.	To allocate resources for individual instances; an instance is an activation of a service.
Load balancing.	Distribute the workload evenly among the servers.
Energy optimization.	Minimization of energy consumption.
Quality-of-service (QoS) guarantees.	Ability to satisfy timing or other conditions specified by a Service Level Agreement.

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

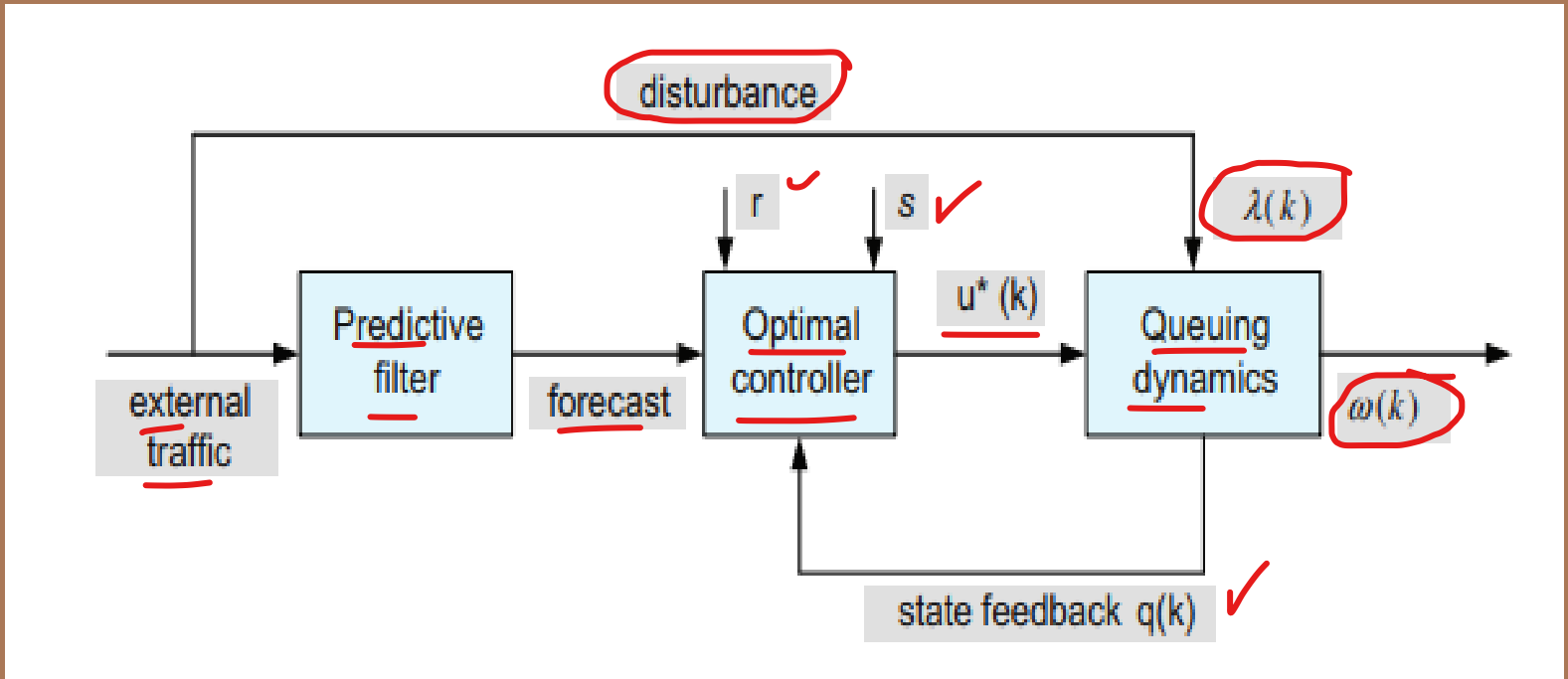
CPU Speed (GHz)	Normalized Energy (%)	Normalized Performance (%)
0.6	0.44	0.61
0.8	0.48	0.70
1.0	0.52	0.79
1.2	0.58	0.81
1.4	0.62	0.88
1.6	0.70	0.90
1.8	0.82	0.95
2.0	0.90	0.99
2.2	1.00	1.00

cumm

Mechanisms for the implementation of resource management policies

- **Control theory :** uses the feedback to guarantee system stability and predict transient behavior.
- **Machine learning :** does not need a performance model of the system.
- **Utility-based :** require a performance model and a mechanism to correlate user-level performance with cost.
- **Market-oriented/economic :** 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



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

Predictive Filter:

- The system starts by analyzing the "external traffic," which might represent the amount of incoming tasks or requests.
- The predictive filter uses this traffic information to create a "forecast." This forecast predicts future traffic, helping the system prepare for what's coming.

Optimal Controller:

- The forecast from the predictive filter, along with some other inputs labeled r (reference or target) and s (settings or constraints), goes into the optimal controller.
- The optimal controller's job is to make the best decisions on how to manage the incoming traffic based on the forecast, target, and constraints. It outputs $u*(k)u^*(k)u*(k)$, which could represent the optimal actions or settings to keep the system running smoothly.
- The controller also receives "state feedback," q(k) which tells it the current state of the queue. This feedback helps it adjust its decisions in real-time.

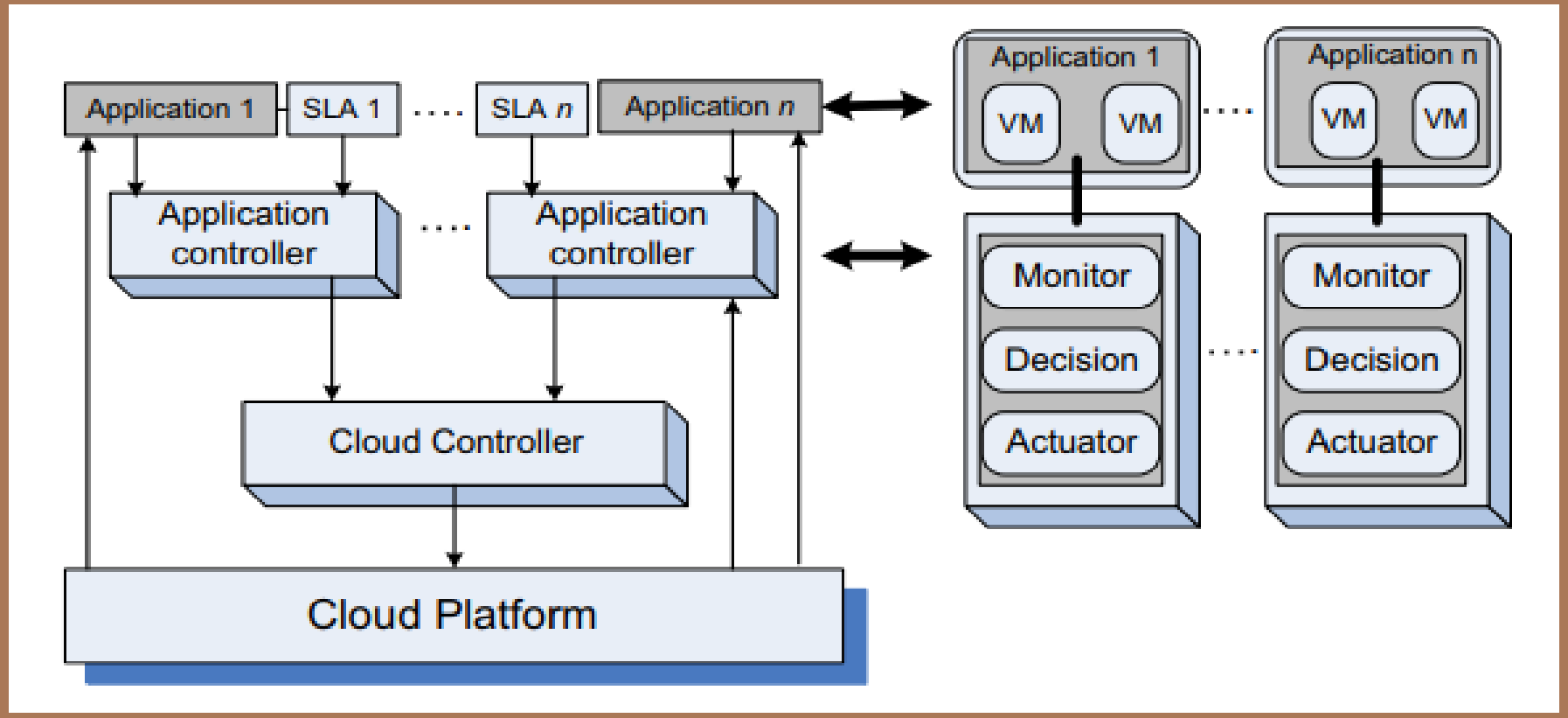
Queuing Dynamics:

- The optimal actions, $u*(k)$, from the controller go to the queuing dynamics block. This part models the actual queue, handling how tasks are processed and managed.
- The queuing dynamics block has two outputs:

$\lambda(k)$: the rate at which tasks are being processed or entering the system.

$\omega(k)$: some other state or measurement related to the queue (e.g., waiting time or queue length).

Stability of a two-level resource allocation architecture



Key Components of the Control System:

Control System Components:

- The system has two levels of control:
  - **Application Level (Local Controllers)**: Each application has its own controller (Application Controller) that manages resources for that specific application. These controllers work based on the application's Service Level Agreement (SLA) to ensure the application meets its performance requirements.
  - **Service Provider Level (Cloud Controller)**: This is the higher-level controller that oversees all applications on the cloud platform. It manages resources across applications, coordinating and balancing resources to ensure efficient use of the cloud platform.

Each application-level controller is equipped with:

- **Monitor**: Tracks performance metrics like CPU usage, memory utilization, etc., for its respective application.
- **Decision Maker**: Analyzes the data from the monitor and decides whether to allocate or deallocate resources.
- **Actuator**: Executes the resource adjustments, like adding or removing virtual machines (VMs) for the application.

Feedback Loop and Stability

- This control system uses feedback from the **Monitors** to keep the system stable. For example:
  - If an application's workload increases, the monitor detects this, and the application controller might decide to allocate more VMs to handle the load.
  - The cloud controller receives feedback from all application controllers and ensures the cloud platform remains balanced overall.
- **Stability** is crucial. If adjustments (like adding or removing VMs) are made too quickly or too frequently, the system can become unstable, causing problems like: ***Thrashing, Instability Sources.***

Lessons Learned from the two-level experiment

- Controllers should wait for the system to stabilize before making adjustments, preventing rapid changes that could lead to instability.
- If upper and lower thresholds are set, they should be spaced sufficiently apart to prevent frequent oscillations. Adjustments like adding or removing VMs must be done carefully to avoid crossing thresholds repeatedly, which could destabilize the system.

Feedback control based on dynamic thresholds

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

- **Thresholds.**

A threshold is the value of a parameter related to the state of a system that triggers a change in the system behavior.

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

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

- **Proportional Thresholding**

In Proportional Thresholding The questions addressed are:

1. 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?
2. Is it feasible to consider fine control? Is course control more adequate in a cloud computing environment?
3. Are dynamic thresholds based on time averages better than static ones?
4. Is it better to have a high and a low threshold, or it is sufficient to define only a high threshold?

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

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

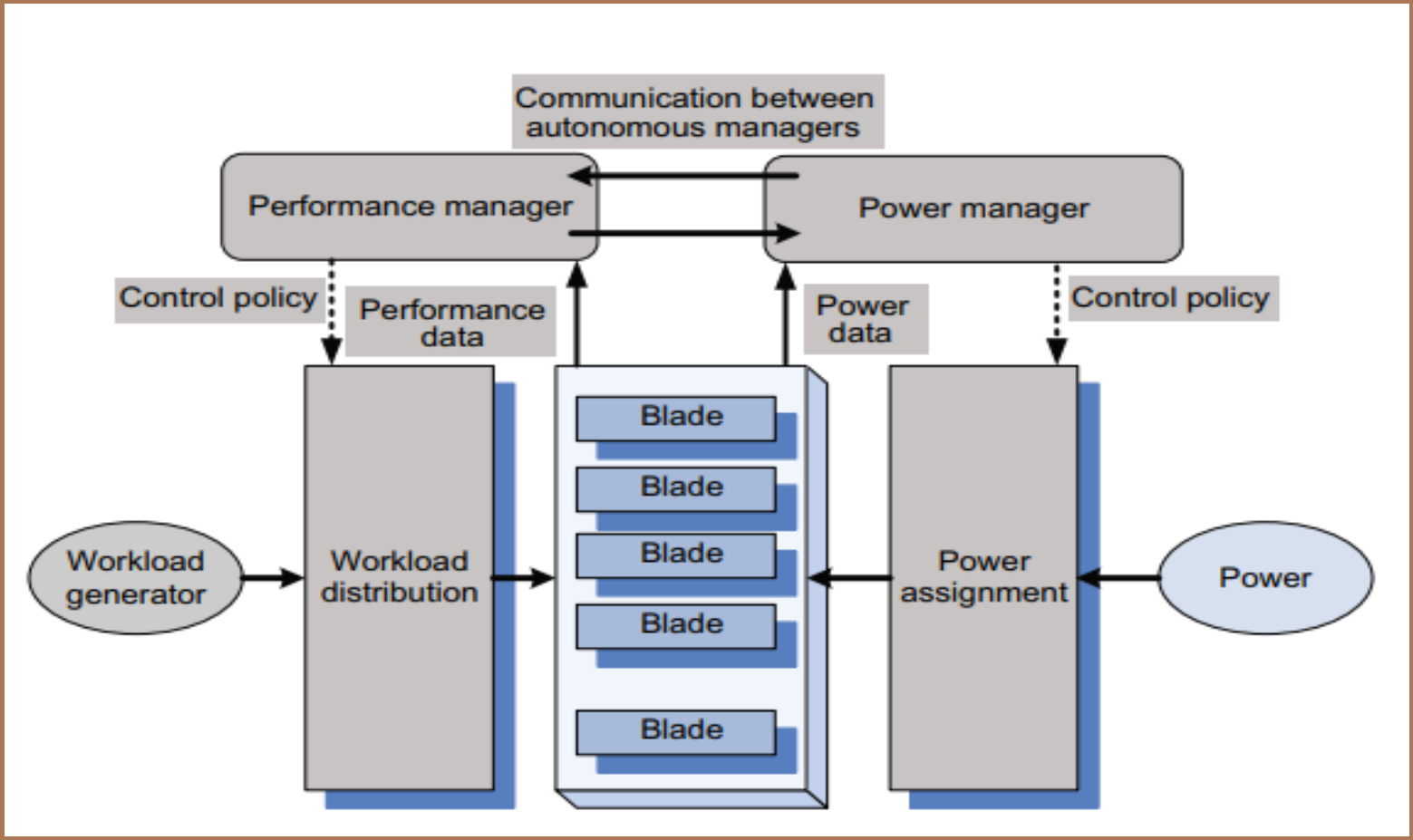
2. Request additional VMs when the average value of the CPU utilization over the current time slice exceeds the high threshold.

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

Conclusions

- Dynamic thresholds perform better than the static ones.
- Two thresholds are better than one.

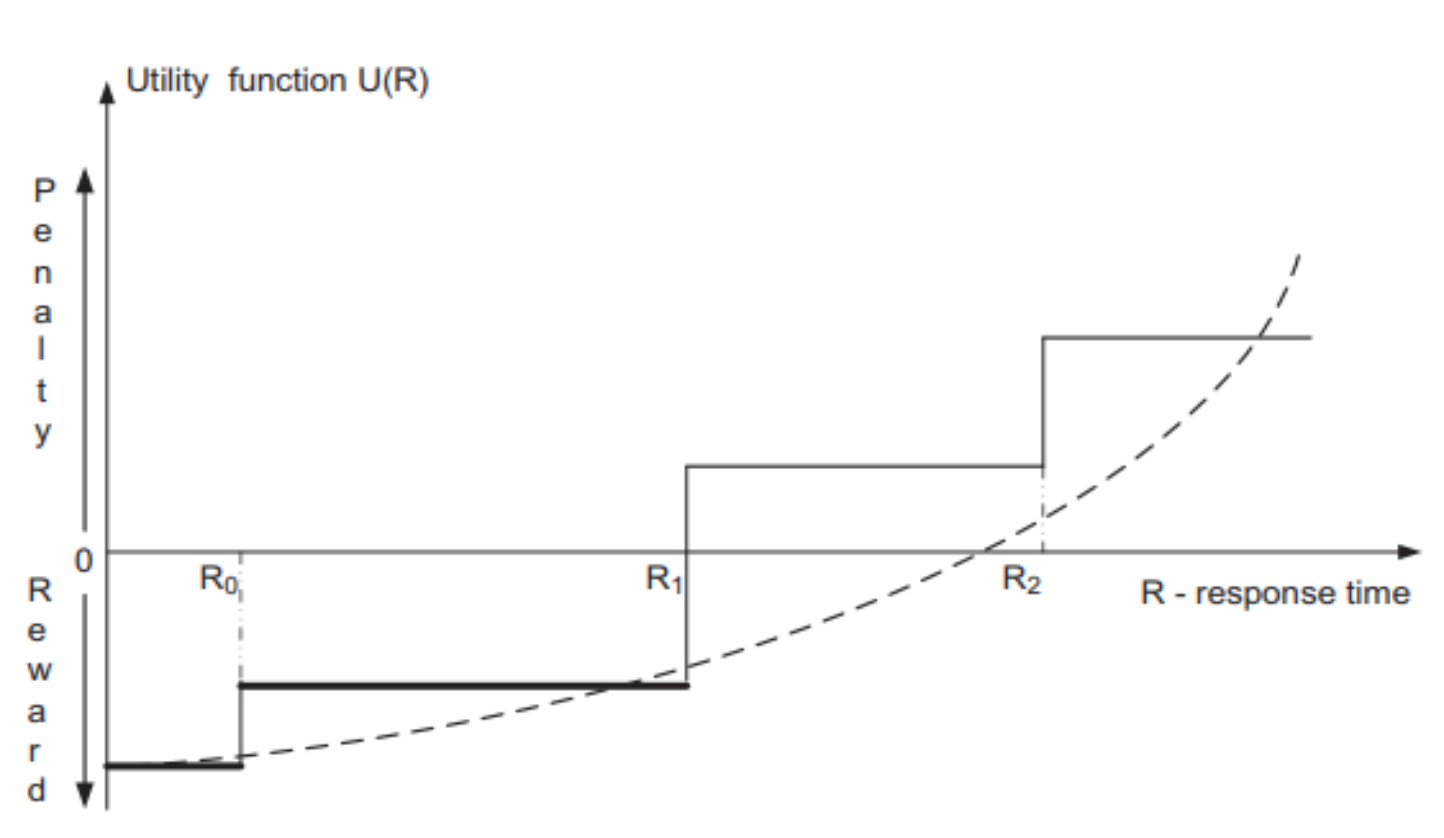
Coordination of specialized autonomic performance managers



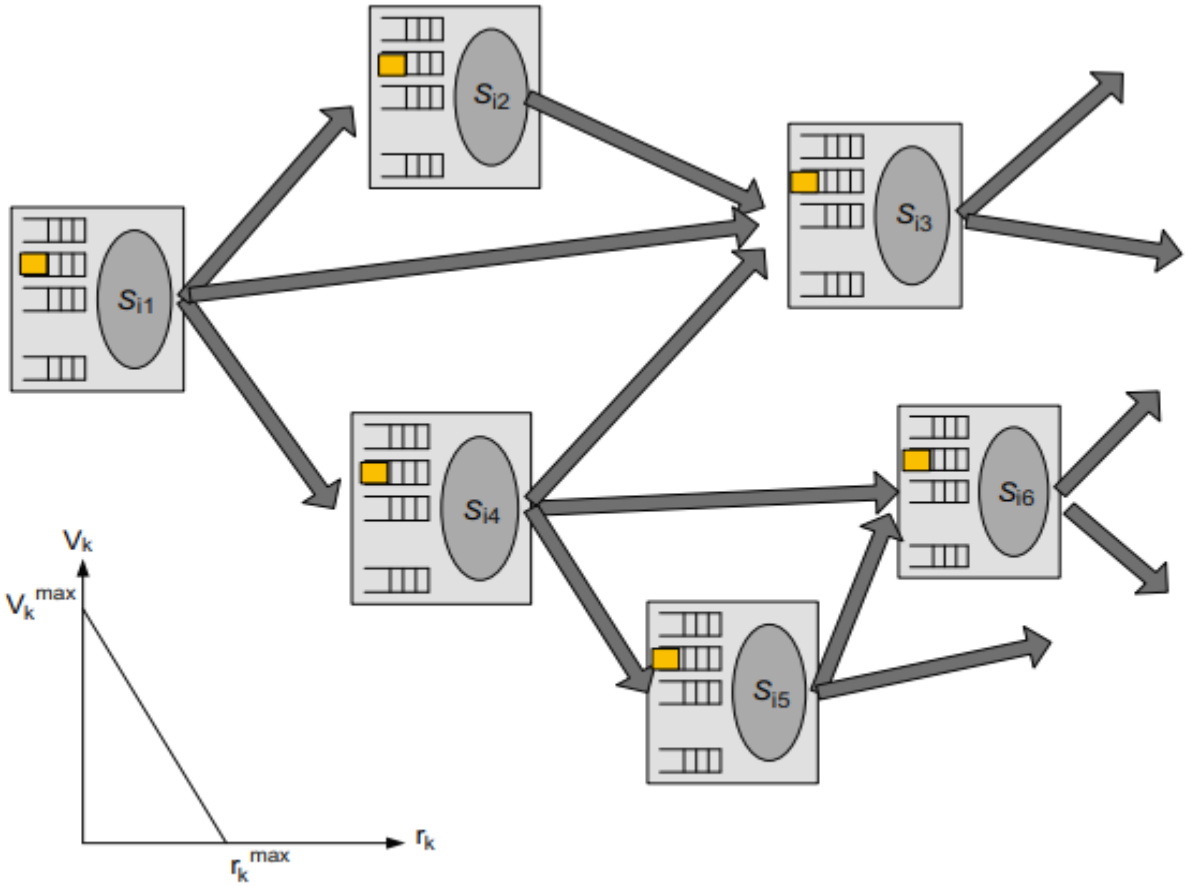
- Autonomous performance and power managers cooperate to ensure SLA prescribed performance and energy optimization.
- They are fed with performance and power data and implement the performance and power management policies, respectively.

- Use separate controllers/managers for the two objectives.
- Identify a minimal set of parameters to be exchanged between the two managers.
- Use a joint utility function for power and performance.
- 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.
- Use standard software systems.

A utility-based model for cloud-based Web services



The utility function U(R) is a series of step functions with jumps corresponding to the response time, R = R0|R1|R2, when the reward and the penalty levels change according to the SLA. The dotted line shows a quadratic approximation of the utility function.



(a) The utility function: v_k the revenue (or the penalty) function of the response time r_k for a request of class k.

(b) A network of multiqueues.

- **A service level agreement (SLA) :** specifies the rewards as well as penalties associated with specific performance metrics.
- The SLA for cloud-based web services uses the average response time to reflect the Quality of Service.
- We assume a cloud providing *K* different classes of service, each class *k* involving N_k applications.
- The system is modeled as a network of queues with multi-queues for each server.
- A delay center models the think time of the user after the completion of service at one server and the start of processing at the next server.

Resource bundling: Combinatorial auctions for cloud resources

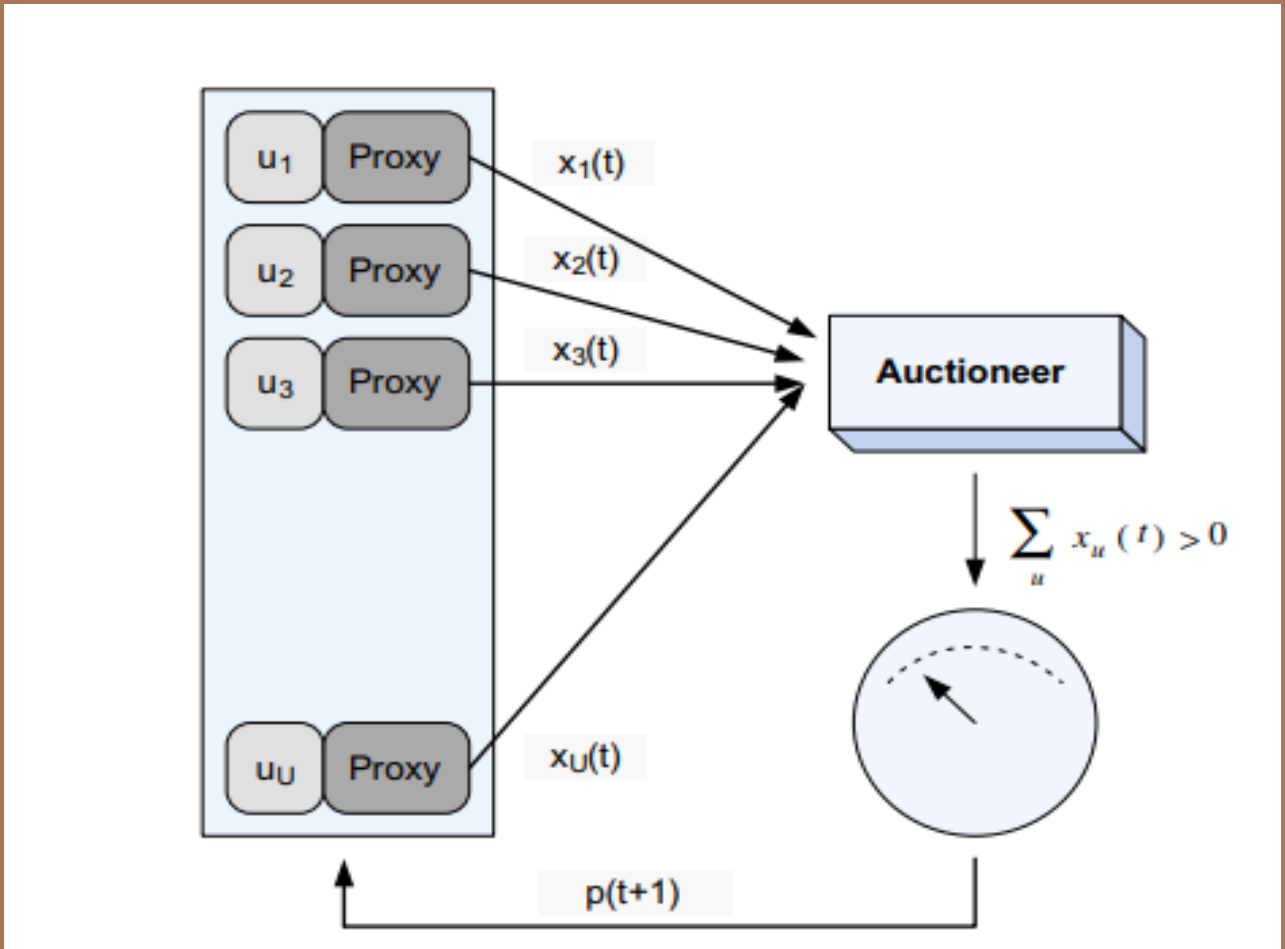
- Resources in a cloud are allocated in bundles, allowing users get maximum benefit from a specific combination of resources. Indeed, along with CPU cycles, an application needs specific amounts of main memory, disk space, network bandwidth, and so on.

Combinatorial Auctions.

In **combinatorial auctions**, users can bid on **combinations** (or **bundles**) of resources rather than individual items.

Types of Combinatorial Auctions

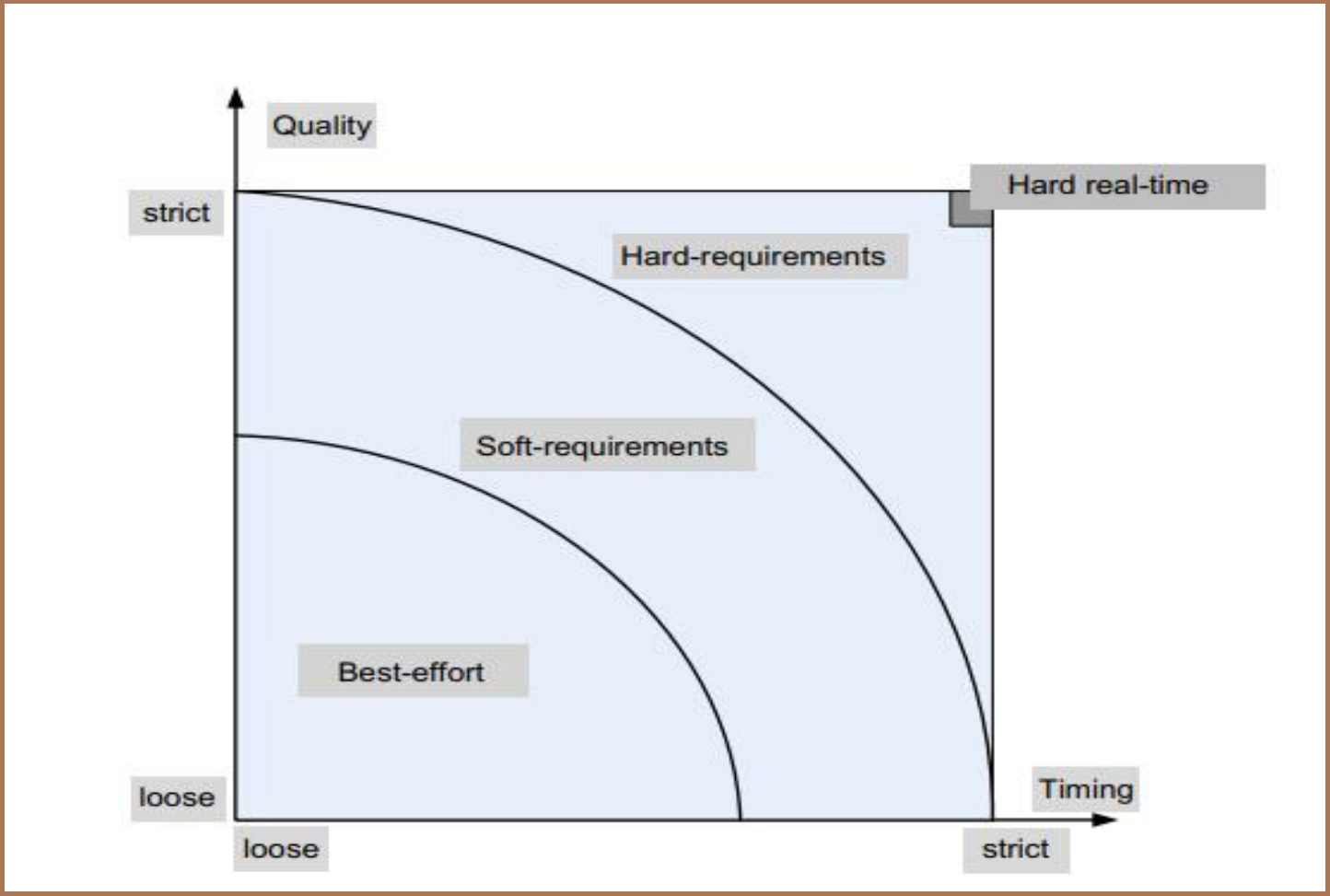
- Some recent combinatorial auction algorithms include:
- **Simultaneous Clock Auction:** All resource prices are visible and updated in real time like a "clock."
- **Clock Proxy Auction:** Users bid on bundles via proxies based on current clock prices.
- **Ascending Clock Auction (ASCA):** Prices start low and gradually increase as users place bids.
- In these auctions:
- The **current price** of each resource is displayed on a "clock" that everyone can see.
- Users compete by placing bids as prices increase.



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

- **Resource Bundling** solves the problem of allocating multiple resources at once.
- **Combinatorial Auctions** are a flexible and scalable way to allocate bundles of resources.
- The process ensures fairness, scalability, and efficient pricing, with clear winners and losers.

Scheduling algorithms for computing clouds



Scheduling in Cloud Systems

In **Resource Sharing Layers**: A **server** supports multiple virtual machines (VMs). Each **VM** supports multiple applications. Each **application** comprises threads, with resources dynamically allocated across these layers.

Quality-of-Service (QoS) Requirements

The **QoS needs vary by application type**, leading to different scheduling approaches:

Best-effort applications, **Soft real-time applications**, **Hard real-time applications**:

Objectives of Schedulers

Schedulers have different objectives depending on the type of system:

Best-effort applications: Examples: Batch jobs, data analytics.	Maximize throughput: Complete as many jobs as possible within a given time. Minimize turnaround time: Reduce the time between job submission and completion.	Scheduling policies: Algorithms like Round-robin, **FCFS** (First Come First Serve), and **SJF** (Shortest Job First) are used here.
Soft real-time applications: Examples: Multimedia streaming (audio/video)	QoS requirements are less rigid than hard real-time tasks but must still maintain performance.	Real-time algorithms like **EDF** (Earliest Deadline First) **or RMA** (Rate Monotonic Algorithm) are often used to prioritize tasks.
Hard real-time applications: Examples: Critical systems (medical, industrial automation).	Strict deadlines and resource guarantees. Tasks must be executed precisely within their timing constraints, or failure occurs.	Advanced algorithms like **RAD** (Resource Allocation/Dispatching) and **RBED** (Rate-Based Earliest Deadline) help integrate hard real-time scheduling in mixed workloads.

Fairness in Scheduling

A scheduler must ensure **fair resource allocation**, especially when multiple users or threads share the same resource. Two fairness criteria are discussed:

Max-min Fairness:

- Ensures no user gets more resources than they request.
- Maximizes the smallest allocation (Bmin) while satisfying constraints recursively.

Weighted Fairness:

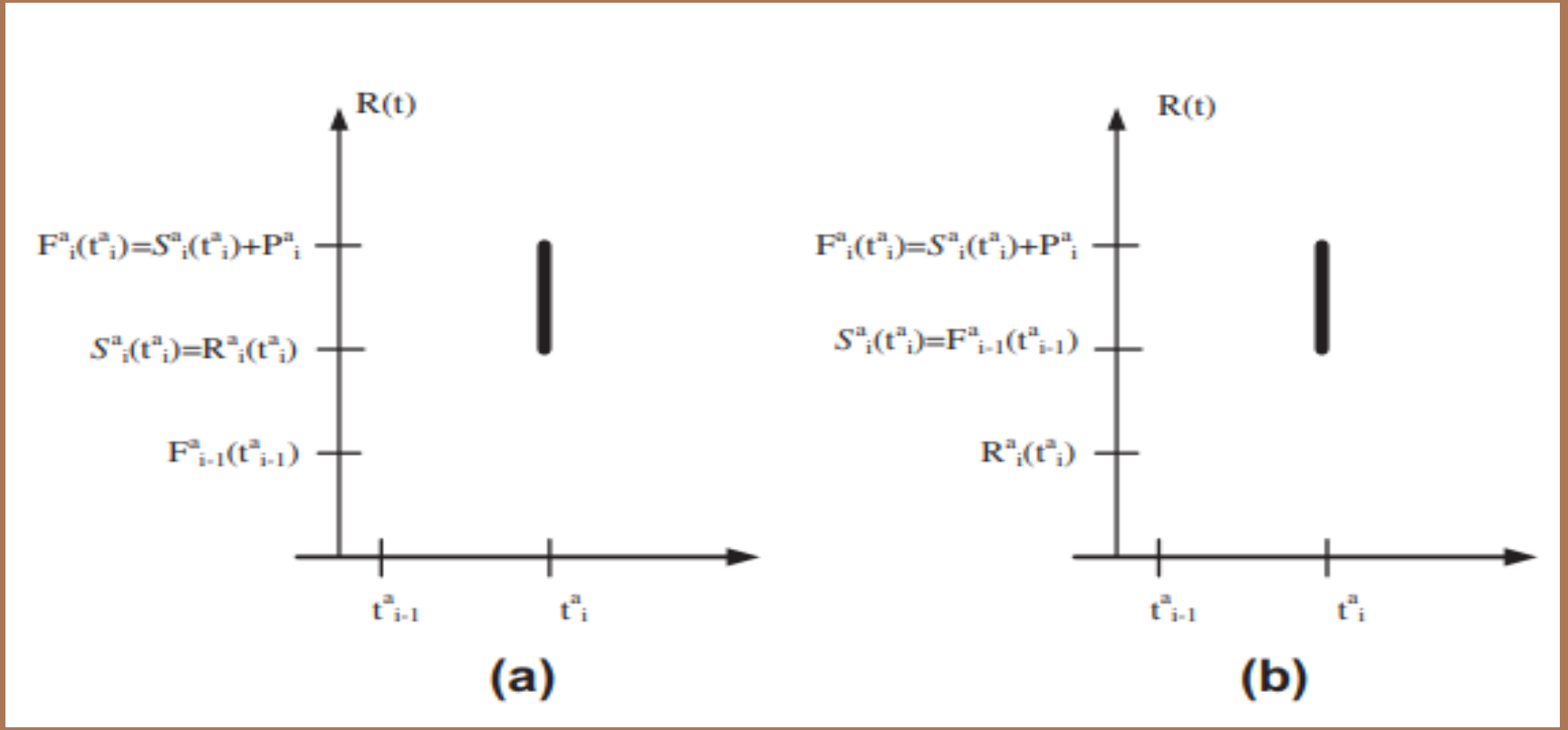
- Allocates resources proportionally based on task weights (e.g., priority or importance).
- Ensures tasks with higher weights receive more resources.

Fair queuing

The **Fair Queuing (FQ)** algorithm is a strategy designed to ensure fairness in resource management, particularly in scenarios where multiple flows or users compete for shared resources like bandwidth in a network or computing power in a cloud environment.

Fair Queuing works by:

- **Creating separate "queues"** for each user or flow, meaning every flow gets its own share of resources.
- **Scheduling resource use in a "round-robin" manner**, where each flow gets a turn to use the resources one by one, rather than letting one greedy flow hog all the resources.
- **Adjusting for fairness** by accounting for the size of the requests (like packet sizes in networking or workload sizes in cloud computing) to ensure no flow unfairly consumes more than its fair share.



The transmission of packet ***i*** of a flow can only start after the packet is available and the transmission of the previous packet has finished.

(a) The new packet arrives after the previous has finished.

(b) The new packet arrives before the previous one was finished.

(a) Case: $F_{i-1}^a < R(t_i^a)$

- **What it means:** The previous packet $(i-1)$ has already finished transmitting when the new packet i arrives. The current round $R(t_i^a)$ is ahead of F_{i-1}^a.
- **Start Time Calculation:**

  Since $F_{i-1}^a < R(t_i^a)$, the new packet i starts immediately at the current round:

$$S_i^a = R(t_i^a)$$

- **Finish Time Calculation:**

  The new packet finishes after being transmitted for its size P_i^a:

$$F_i^a = S_i^a + P_i^a = R(t_i^a) + P_i^a$$

Explanation: The new packet does not need to wait because the flow is idle. This situation corresponds to Diagram (b), where the transmission of the new packet begins immediately upon its arrival.

(b) Case: $F_{i-1}^a \geq R(t_i^a)$

- **What it means:** The previous packet $(i-1)$ has not yet finished transmitting when the new packet i arrives. The current round $R(t_i^a)$ is less than or equal to F_{i-1}^a.
- **Start Time Calculation:**

  Since $F_{i-1}^a \geq R(t_i^a)$, the new packet i must wait for the previous packet to finish:

$$S_i^a = F_{i-1}^a$$

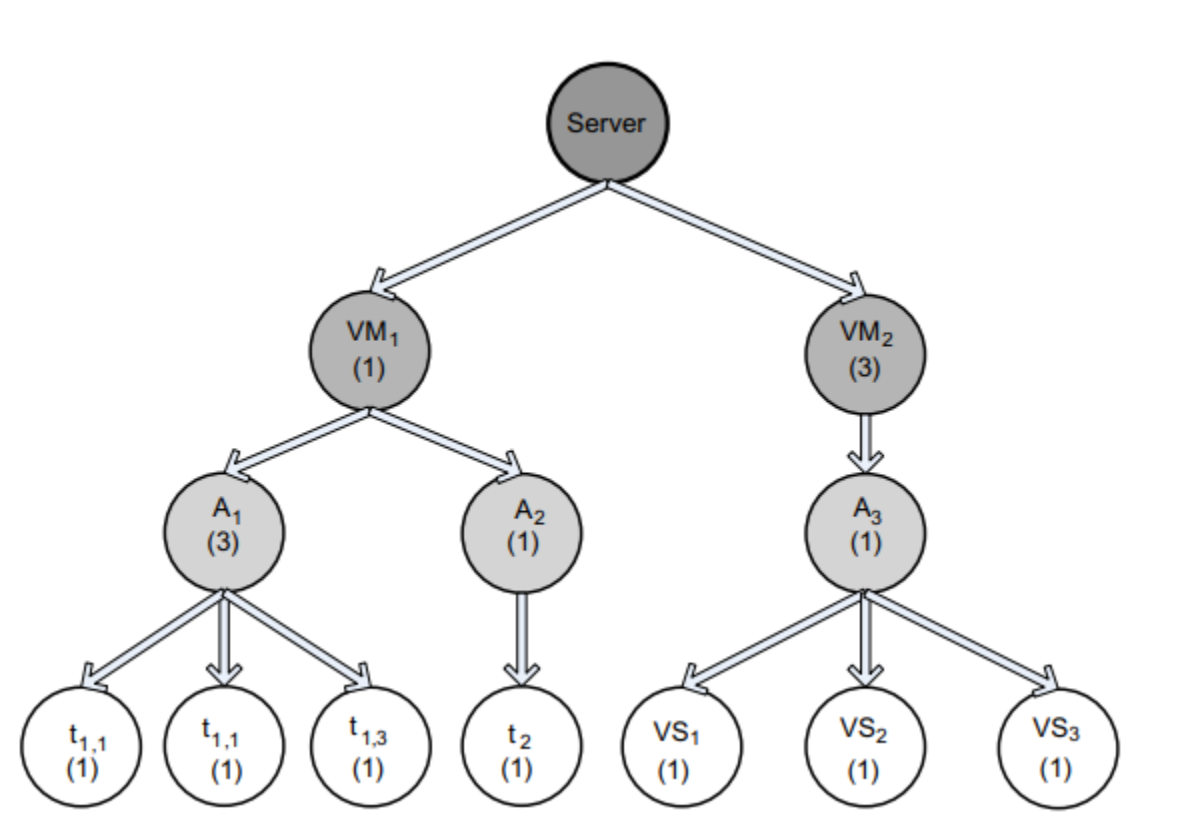
- **Finish Time Calculation:**

  Once the new packet starts, it finishes after its size P_i^a:

$$F_i^a = S_i^a + P_i^a = F_{i-1}^a + P_i^a$$

Explanation: The new packet must wait for the previous one to complete before starting its transmission. This corresponds to Diagram (a), where the start of the new packet is delayed until the previous packet finishes.

Start-time fair queuing



- The SFQ tree for scheduling when two virtual machines, VM1 and VM2, run on a powerful server.
- VM1 runs two best-effort applications A1, with three threads t1,1, t1,2, and t1,3, and A2 with a single thread, t2.
- VM2 runs a video-streaming application, A3, with three threads vs1, vs2, and vs3. The weights of virtual machines, applications, and individual threads are shown in parenthesis.

- 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.
  - 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.
  - If one of the threads of an application is not runnable then its allocation is transferred to the other threads of the applications.

Borrowed virtual time

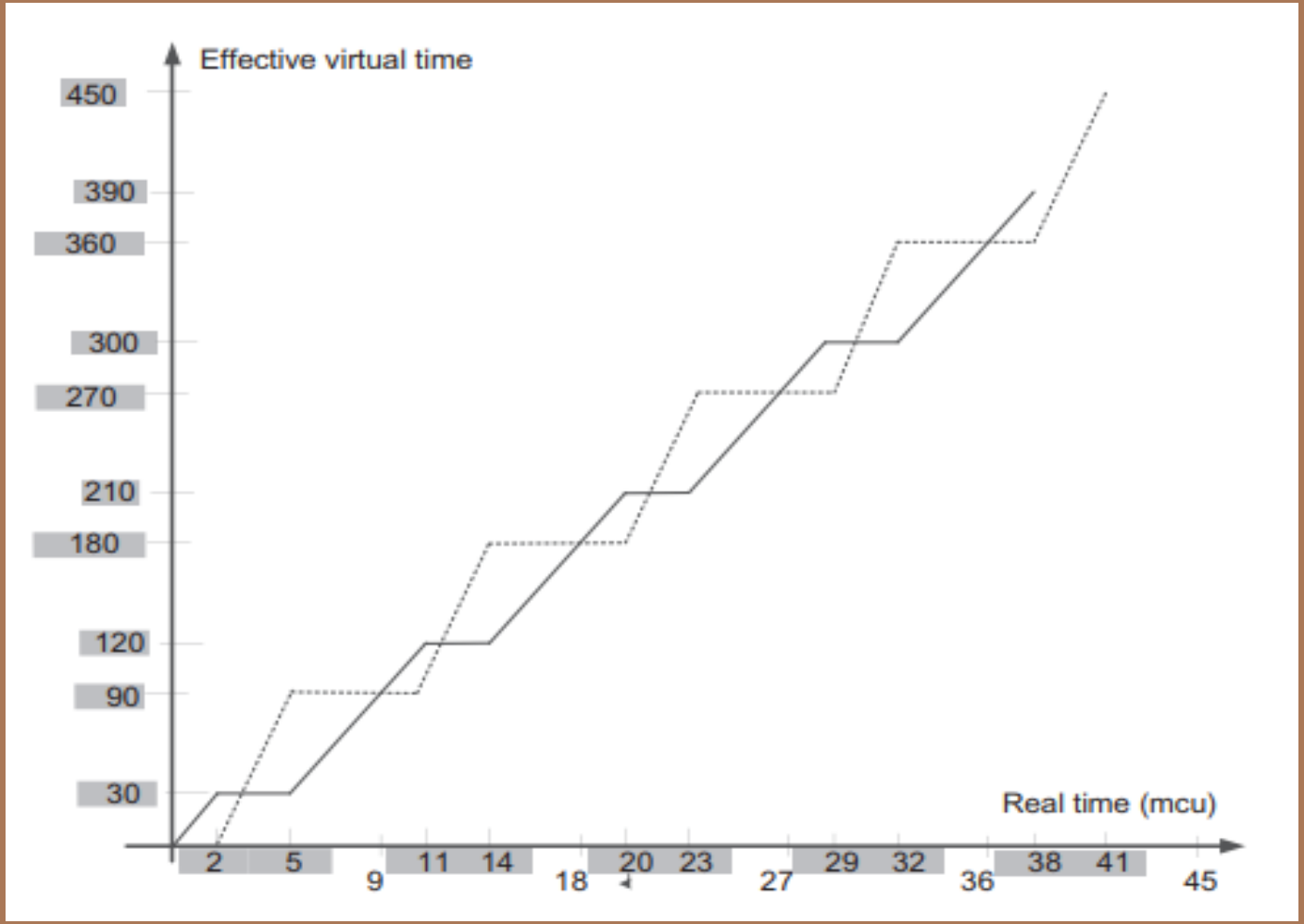
- The **Borrowed Virtual Time (BVT)** algorithm is a CPU scheduling technique designed to balance **low-latency dispatching** (for real-time applications) with **fair CPU sharing** (among various applications).A thread ***i*** has
  - an effective virtual time, E_i.
  - an actual virtual time, A_i.
  - a virtual time warp, W_i.
- The scheduler thread maintains its own scheduler virtual time (SVT) defined as the minimum actual virtual time of any thread.
- The threads are dispatched in the order of their effective virtual time, policy called the Earliest Virtual Time (EVT).
- Context switches are triggered by events such as:
  - the running thread is blocked waiting for an event to occur.
  - the time quantum expires.
  - an interrupt occurs.
  - when a thread becomes runnable after sleeping.

Example 1. The following example illustrates the application of the BVT algorithm for scheduling two threads a and b of best-effort applications. The first thread has a weight twice that of the second, wa = 2wb; when ka = 180 and kb = 90, then Δ= 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,...

  b : 2, 2 + 9 = 11, 11 + 9 = 20, 20 + 9 = 29, 29 + 9 = 38,...
- The 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, consists of only 3 mcu; a context switch occurs when a, which runs first, exceeds b by 2 mcu



The effective virtual time and the real time of the threads ***a*** (solid line) and ***b*** (dotted line) with weights $w_a = 2\ w_b$ when the actual virtual time is incremented in steps of 90 mcu.