Chapter 12

Resource Management and Quality of Service

In a general sense, Quality of Service (QoS) is defined as a set of quality requirements (i.e., desirable properties) of an application, which are not explicitly formulated in its functional interfaces (1.4.1). In that sense, QoS includes fault tolerance, security, performance, and a set of "ilities" such as availability, maintainability, etc. In a more restricted meaning (considered in this chapter), QoS characterizes the ability of an application to satisfy performance-related constraints. This specific meaning of QoS is specially relevant in areas such as multimedia processing, real-time control, or interactive services for end users.

Performance control is achieved through resource management, the main theme of this chapter. We examine the main abstractions and patterns for managing resources, including the use of feedback control methods.

12.1 Introducing Resource Management

The function of a computing system is to provide *services* to its users. Each service is specified by a contract between a service provider and a service requester. This contract defines both the functional interface of the service and some extra-functional aspects, collectively known as Quality of Service (QoS), which include performance, availability, security, and need to be accurately specified for each application or class of applications. The part of the contract that defines QoS is called a *Service Level Agreement* (SLA). The technical expression of an SLA usually consists of a set of *Service Level Objectives* (SLO), each of which defines a precise objective for one of the specific aspects covered by the SLA. For instance, for an SLA on the performance of a web server, an SLO can specify a maximum response time to be achieved for 95% of the requests submitted by a certain class of users.

12.1.1 Motivation and Main Definitions

In order to perform its function, a computing system uses various resources such as processors, memory, communication channels, etc. Managing these resources is an important function, and there are several strong reasons for performing it accurately:

- Maintaining quality of service. Various indicators related to QoS in user applications, specially performance factors, are directly influenced by resource allocation decisions.
- Resource accounting. The users of a shared facility should be charged according to their actual resource consumption. Therefore any consumed resource must be traced back to a user activity.
- Service differentiation and resource pricing. The resource management system may allow users to pay for improved service. The system should guarantee this differentiated form of service, and the pricing scheme should adequately reflect the added value thus acquired.
- Detecting and countering Denial of Service (DoS). A DoS attack aims at preventing useful work from being done, by undue massive acquisition of resources such as CPU time, memory, or network bandwidth.
- Tracking and eliminating performance bugs. Without accurate monitoring of resource usage, a runaway activity might invisibly consume large amounts of resources, leading to performance degradation in user applications; or a regular activity may reserve unnecessary resources, thus hampering the progress of other activities.

In the traditional view of a computing system, resource allocation, i.e., the sharing of a common set of resources between applications contending for their use, was a task performed by the operating system, and user applications had little control over this process. This situation has changed due to the following causes:

- The increasing number of applications subject to strong constraints in time and space, e.g., embedded systems and applications managing multimedia data.
- The growing variability of the environment and operating conditions of many applications, e.g., those involving mobile communications.
- The trend towards more open systems, and the advent of open middleware.

Thus an increasing part of resource management is being delegated to the upper levels, i.e., to the middleware layers and to the applications themselves. In this chapter, we examine some aspects of resource allocation in these upper levels. We start by recalling a few basic definitions (see also 10.1.1).

The term *resource* applies to any identifiable entity (physical or virtual) that is used by a system for service provision. The entity that actually implements service provision, using resources, is called a *resource principal*. Examples of physical resources are processors, memory, disk storage, routers, network links, sensors. Examples of virtual resources are virtual memory, network bandwidth, files and other data (note that virtual resources are abstractions built on top of physical resources). Examples of resource principals are processes in an operating system, groups of processes dedicated to a common task (possibly across several machines), various forms of "agents" (computational entities that may move or spread across the nodes of a network). One may define a hierarchy of principals: for example, a virtual machine provides (virtual) resources to the applications it supports, while requesting (physical or even virtual) resources from a hypervisor.

A general principle that governs resource management is the separation between policies and mechanisms, an instance of separation of concerns (1.4.2). A *policy* defines overall goals for resource allocation, and the algorithms or rules of usage best suited to reach these goals. Such algorithms are implemented using *mechanisms*, which are defined at a low level, with direct access to the individual resources. These mechanisms must be neutral with respect to policies; i.e., several policies may be implemented using the same set of mechanisms, and the design of a mechanism should not preclude its use by different policies. An example of a general policy is to ensure that a given resource (e.g., CPU time) is allocated to a set of processes so that each process gets a share of that resource proportional to a predefined ratio. This policy may be implemented by various mechanisms, e.g., round robin with priorities or lottery scheduling (12.3.2). Conversely, these mechanisms may be used to implement different policies, e.g., including time-varying constraints.

12.1.2 Goals and Policies

The role of resource management is to allocate resources to the service providers (principals), subject to the requirements and constraints of both service providers and resource providers. The objective of a service provider is to respect its SLA, the contract that binds it to service requesters (clients). The objective of a resource provider is to maximize the utilization rate of its resources, and possibly the revenue it draws from their provision. The relationships between clients, service providers, and resource providers are illustrated on Figure 12.1 (as noted before, a resource provider may itself request resources from a higher level provider).

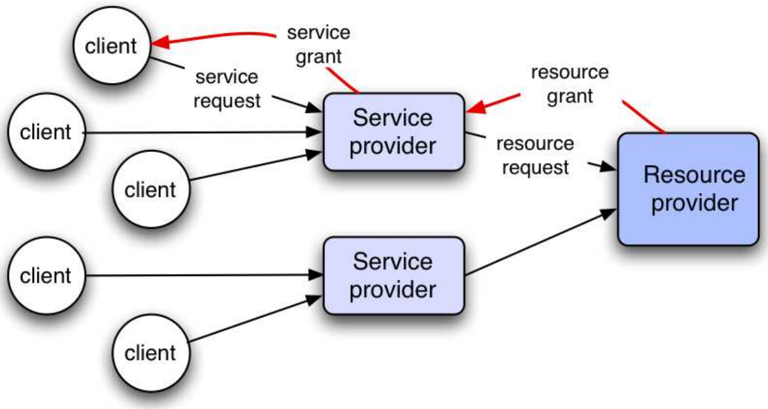

Figure 12.1. Service providers and resource providers

The requirements are the following, as seen by a service provider.

- The service provider should offer QoS guarantees to its clients, as specified by an SLA. The SLA may take various forms: strict guarantee, probabilistic (the agreed levels will be reached with a specified probability, or for a specified fraction of the demands), best effort (no guarantee on the results). The SLA is a contract that goes both ways, i.e., the guarantees are only granted if the clients respects some conditions on their requests. These conditions may apply to each client individually or to a population of clients as a whole.

- The service provider allocates resources for the satisfaction of clients' requests. It should ensure equitable treatment (*fair share*), in the sense that each client needing a resource should be guaranteed a share of that resource proportional to its "right" (or "priority"), as defined by a global policy. The share should be guaranteed in average over a specified period of time. If all clients have the same right, then each should be guaranteed an equal share. Other situations may occur:

  – *Service differentiation* is a means of specifying different classes of clients with different rights, which may be acquired by purchase or by negotiation.
  – The rights may be time-dependent, e.g., the right of a client may be increased to allow it to meet a deadline.

  More generally, the guarantee may be in terms of a minimal rate of progress, which prevents *starvation*, a situation in which a client's requests are indefinitely delayed.

- If several classes of clients are defined, the service provider should guarantee *service isolation*: the allocation of resources for a class should not be influenced by that of other classes. This means that a misbehaving client (one that does not respect its side of the SLA) may only affect other clients of its class, not clients from other classes.

The main requirement of a resource provider is to optimize the usage of the resources, according to various criteria, e.g., ensuring maximal utilization rate over time (possibly with a different weight for each resource), or minimizing energy consumption, etc. This implies that no resource should be idle while there exist unsatisfied requests for that resource (unless this situation is imposed by a higher priority policy, e.g. energy economy). This requirement is subject to the constraint of resource availability: the supply of most resources is bounded. There may be additional constraints on resource usage, e.g., due to administrative reasons, or to locality (e.g., usage and cost conditions are different for a local and a remote resource).

In all cases, a *metric* should be defined to assess the satisfaction of the requirements, both for QoS and for resource utilization factors. Metrics are examined in 12.3.1.

The above requirements may be contradictory. For instance, ensuring guarantees through worst case reservation may entail sub-optimal resource utilization. Such conflicts may only be resolved according to a higher level policy, e.g., through priority setting or through negotiation. Another approach is to pool resources, in order to amortize their cost among several applications, provided that peak loads do not occur at the same time for all of them. Also note that the fairness requirement should be met both at the global level (sharing resources among service providers) and for each service provider with respect to its own service requesters.

Resource management policies may be classified using various criteria (based on prediction and/or observation, using open loop or closed loop) and may face various operating conditions: if resources are globally sufficient to meet the global demand, combinatorial optimization is the relevant tool; if resources are insufficient (the common case), control methods are more appropriate. Examples throughout this chapter illustrate these situations.

Resource allocation is subject to a number of known risks, which any resource management policy should take into account.

- *Violation of fairness*, examples of which are the above-mentioned starvation, and priority inversion, a reversal of prescribed priorities due to unwanted interference between synchronization and priority setting (see 12.3.2).
- *Congestion*, a situation in which the available resources are insufficient to meet the demand. This may be due to an inadequate policy, in which resources are overcommitted, or to a peak in the load. As a result, the system's time is essentially spent in overhead, and no useful work can be done, a situation known as *thrashing*. Thrashing is usually avoided or delayed by an admission control policy (see 12.3.1).
- *Deadlock*, a situation of circular wait, in which a set of processes are blocked, each of them waiting for a resource that is held up by another member of the set. Deadlock may be prevented by avoiding circular dependencies (e.g., through ordered allocation), or detected and resolved, usually at some cost in progress rate.

In the context of this book, we are specifically interested in resource management for middleware systems; among these, Internet services are the subject of an intense activity, due to their economic importance. We conclude this section with a review of the main aspects of resource management for this class of systems.

12.1.3 Resource Management for Internet Services

An increasing number of services are available over the Internet, and are subject to high demand. Internet services include electronic commerce, e-mail, news diffusion, stock trading, and many other applications. As these services provide a growing number of functions to their users, their scale and complexity have also increased. Many services may accept requests from millions of clients. Processing a request submitted by a client typically involves several steps, such as analyzing the request, looking up one or several databases to find relevant information, doing some processing on the results of the queries, dynamically generating a web page to answer the request, and sending this page to the client. This cycle may be shortened, e.g., if a result is available in a cache. To accommodate this interaction pattern, a common form of organization of Internet services is a multi-tier architecture, in which each tier is in charge of a specific phase of request processing.

To answer the demand in computational power and storage space imposed by large scale applications, clusters of commodity, low cost machines have proved an economic alternative to mainframes and high-performance multiprocessors. In addition to flexibility, clusters allow high availability by replicating critical components. Thus each tier of a cluster-based application is deployed on a set of nodes (Figure 12.2). How the application components running on the servers of the different tiers are connected together depends on the architecture of the application; examples may be found in the rest of this chapter. Nodes may be reallocated between different tiers, according to the resource allocation policy.

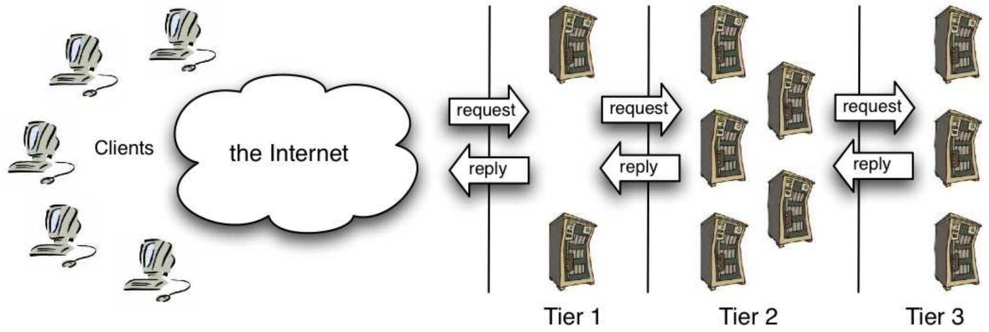

Figure 12.2. A cluster-based multi-tier application

A service provider may deploy its own cluster to support its applications. An alternative solution, in increasing use, is for the provider to host the application on a general purpose platform (or data center) owned by a computing facility provider, and shared with other service providers. The drawback of this solution is that the service provider has less control on the fine-grain tuning of the infrastructure on which its application is running. However, there are several benefits to using shared platforms.

- The service provider is freed from the material tasks of maintaining the infrastructure.
- The fixed cost of ownership (i.e., the part of the cost that is not proportional to the amount of resources) is shared between the users of the common facility.
- Mutualizing a large pool of resources between several applications allows reacting to load peaks by reallocating resources, provided the peaks are not correlated for the different applications.
- Resource sharing improves global availability, because of redundancy in hosts and network connections.

The load imposed on a service is defined by the characteristics of the requests and by their distribution in time. A request is characterized by the resources it consumes at each stage of its processing. Trace analysis of real loads has allowed typical request profiles to be determined for various classes of services (e.g., static content web server, electronic stores, auctions, etc.), which in turn allow building realistic load generators (see examples in [TPC 2006], [Amza et al. 2002]). Predicting the time distribution of the load is much more difficult, since the demand on Internet services is subject to huge, unexpected, variations. In the so-called *flash crowd* effect, the load on a service may experience a 100-fold or 1000-fold increase in a very short time, causing the service to degrade or to crash. Dealing with such overload situations is one of the main challenges of resource allocation.

We conclude this section by a summary of the main issues of resource allocation for Internet services.

Determining the amount of resources of a computing infrastructure is known as the *capacity planning* problem (see e.g., [Menascé and Almeida 2001]). Its solution relies on an estimation of the expected load. However, due to the possibility of wide variations in the load, dimensioning the installation for the maximum expected load leads to over-provisioning. Given that peak loads are exceptional events, a better approach is to define SLAs for a "normal" load (determined after observation, e.g., as the maximum load over 95% of the time). Relating resources to load is done by using a model, such as those described in 12.5. Overload situations are dealt with by admission control, as discussed below.

The problem of *resource provisioning* is that of allocating the resources of an installation in the face of competing demands. These demands may originate from different applications in the case of a shared facility, or from different stages of an application, or from different clients. In all cases, provisioning may be static or dynamic. Static provisioning, also called reservation, is based on an a priori estimate of the needs. Dynamic provisioning varies with time and is adapted to the current demand through a control algorithm.

Admission control (sometimes called *session policing*[1]) is a response to an overload situation, which consists in turning away a fraction of the requests to ensure that the remaining requests meet their SLA. Admission control needs to take into account some measure of the "value" of the requests, which may have different forms: class of service (if service differentiation is supported), estimated resource consumption of the request, etc.

A complementary way of dealing with overload situations is *service degradation*, which consists in providing a lower quality of service to some requests, thus consuming less resources. The measure of quality is application-dependent; examples include lowering image resolution, stripping down a web page by eliminating images, reducing requirements on the freshness or consistency of data, etc.).

In the rest of this chapter, we examine some approaches to solving these problems. We first examine the main abstractions relevant to resource management. We then consider resource management from the point of view of control and we present the main mechanisms and policies used for resource management at the middleware or application level, using both feed-forward and feedback forms of control. This presentation is illustrated by several case studies.

12.2 Abstractions for Resource Management

Resource management involves three main classes of entities: the resources to be allocated, the resource principals (resource consumers), and the resource managers. The first task in developing a resource management framework is to define suitable abstractions for these entities and for their mutual relationships, and to design mechanisms for their implementation.

[1] This term comes from network technology: *traffic policing* means controlling the maximum rate of traffic sent or received on a network interface, in order to preserve network QoS.

12.2.1 Resources

The physical resources of a system are the identifiable items that may be used for the execution of a task, at a certain level of visibility. Usually this level is that of executable programs, for which the resources are CPUs, memory, disk space; communication related resources, such as disk or network I/O, are quantified in terms of available bandwidth.

An operating system creates an abstract view of these resources through virtualization, in order to hide low-level allocation mechanisms from its users. Thus CPU time is abstracted by threads, physical memory by virtual memory, and disk storage by files. These abstractions may be considered as resources when allocated to users through processes; but, from the point of view of the operating system kernel, they are principals for the corresponding physical resources.

The whole set of physical resources that makes up a machine may itself be abstracted in the form of a virtual machine, which provides the same interface as the physical machine to its users, and runs its own operating system. This is again a two-level allocation mechanism: at the lower level, a hypervisor multiplexes the physical resources between the virtual machines; on each virtual machine, the operating system's allocator shares the virtual resources between the users. The interplay between the two levels of resource management is examined in Section 12.2.2. The notion of a virtual machine has been extended to multiprocessors [Govil et al. 2000] and to clusters (as discussed in more detail in Section 12.2.2).

A resource is defined by a number of properties, which influence the way it may be used.

- Exclusive or shared use. Most resources are used in exclusive mode, i.e., they may only be allocated to one principal at a time. Such is the case of a CPU, or a block of private memory. In some infrequent cases, a resource may be shared, i.e., simultaneously used by several principals. This is the case of a shared, read-only memory page or disk file. Even in that case, a maximal degree of sharing may be specified to avoid performance degradation.
- Stateful or stateless. A resource may have a state related to the principal that currently uses it. This state needs to be set up when the resource is allocated; it should be cleaned up, and possibly stored for later reuse, when the resource is reallocated. This operation may be simple and cheap (e.g., in the case of a CPU, in which the state consists of a set of registers), or fairly costly, in the case of a whole computer for which a new operating system may need to be installed.
- Individual or pooled. A resource may exist as a single instance, or it may be part of a pool of identical resources (such that any resource of the pool may be allocated to satisfy a request). Typical pooled resources are memory pages, CPUs in a multiprocessor, or machines in a homogeneous cluster. Note that, if a pooled resource is stateful, there may be a preference to reallocate an instance previously used by the same principal, in order to spare state cleanup and restoration (this is called *affinity scheduling* in the case of CPUs).

  For virtual resources, pooling is often used as an optimization device. Contrasting with physical resources, virtual resources may be created and deleted; these operations have a cost. In order to reduce the overhead of creation and deletion, a pool of resources is initially created. A request is satisfied by taking a resource from the pool; when released, the resource is returned to the pool. Various policies may be used to adapt the size of the pool to the mean demand (if the pool is too small, it will frequently be exhausted; if too large, resources will be wasted). Such pools are frequently used for threads, for network connections, or for components in container-based middleware.

In addition to the above characteristics, a resource may have a number of attributes (e.g., for a printer, its location, its throughput, its color characteristics, etc.) and may allow access to some current load factors (e.g., number of pending requests, etc.). All the properties of a resource are typically grouped in a data structure, the resource descriptor, and are used for resource discovery, the process of looking up for a resource satisfying some constraints (e.g., the closest color printer, or the one with the shortest job queue, etc.). Techniques used for service discovery (3.2.3) are relevant here.

In a large scale system, providing an accurate, up to date, view of the current state of all resources may in itself be a challenging task. Such a global view is seldom needed, however, since most distributed resource management systems aim at satisfying a request with a resource available in a local environment.

12.2.2 Resource Principals

In current operating systems, resources such as virtual memory or CPU time are allocated to processes. A process, in turn, is created on behalf of a user. Thus the resources consumed by a process are charged to the user for which it executes.

This scheme is not adequate for fair and accurate resource management, for several reasons: the time spent in the system kernel, e.g., for servicing interrupts, is not taken into account; a single system process may do work for a number of different users, e.g., for network related activities; conversely, a single user task may be split across several processes. Therefore there is a need for defining a separate notion of a resource principal.

This is yet another application of the principle of separation of concerns (1.4.2). A resource principal is defined as a unit of independent activity devoted to the execution of a well-identified service; all the resources that is uses should also be well-identified. There is no reason why resource principals should coincide with entities defined using different criteria such as unit of protection or unit of CPU scheduling.

Note that resource principals are essentially mechanisms; they may be used to implement any globally defined policy for resource sharing among the independent activities that they represent.

Several attempts have been done towards defining resource principals and relating them to other entities. We briefly review some of these efforts. This subject is still an area of active research.

Resource Containers

On a single machine, *resource containers* have been introduced in [Banga et al. 1999], specifically in the context of networked servers, such as Web or database servers. A resource container is an abstract entity that contains all the system resources used by an application to achieve a particular independent activity (in the case of Web servers, an activity is the servicing of a particular HTTP connection).

Consider the case of CPU time. A resource container is bound to a number of threads, and this binding is dynamic. Thus a given thread may do work for several containers and its binding changes accordingly over time. Each container receives CPU time according to some policy (e.g., fixed share, etc.), and distributes it among the threads that are currently bound to it. Thus a thread that is multiplexed among several containers receives resources, at different times, from all these containers[2].

Other resources are managed in a similar manner. Thus a container may be dynamically bound to a number of sockets and file descriptors, to implement the management of network and disk bandwidth, respectively.

Containers may be organized in a hierarchy to control resource consumption at a finer grain: a container may divide its resources among child containers. Resource allocation within a container, managed at a certain level, is independent of resource allocation between the containers at that level. Containers defined at the top level get their resources from the system, according to a global resource management policy.

In the experiments described in [Banga et al. 1999], resource containers have been used to implement prioritized handling of clients, to control the amount of resources used by CGI processing, and to protect a server against SYN-flooding, a DoS attack that attempts to monopolize the use of the server's network bandwidth.

Cluster Reserves and Related Work

Cluster reserves [Aron et al. 2000] are an extension of resource containers to cluster-based servers, running on a set of commodity workstations connected by a network. Typically, a request for a service is sent to a node designated as the front-end, which in turn assigns the execution of the request to one or several nodes, according to a load sharing policy (12.3.2). Such a policy defines a set of service classes (units of independent activity), together with rules for sharing the cluster's resources among the service classes. An implementation of a global policy should ensure performance isolation, i.e., it should guarantee that these rules are actually enforced (for instance, an application should not "steal" resources from another one, leading to a violation of the global allocation policy). Assigning a request to a service class may be done on various criteria, such as request content, client identity, etc.

Service classes are implemented by cluster reserves. A *cluster reserve* is a cluster-wide resource principal that aggregates resource containers hosted on the nodes of the cluster. A resource globally allocated to a cluster reserve may be dynamically split among a set of resource containers on different nodes.

The problem of partitioning the resources among the individual containers is mapped to a constrained optimization problem. The goal is to compute a resource allocation on each node, while satisfying several constraints: the total allocation of each resource for

[2]If the binding changes frequently, rescheduling the thread at each change may prove costly; thus the scheduling of a shared thread is based on a combined allocation of the set of containers to which it is bound, and this set is periodically recomputed; thus the rescheduling period may be controlled, at the expense of fine grain accuracy.

each reserve should minimally deviate from that specified by the global allocation policy; the total allocation of any resource on each node should not exceed the available amount; no resource should be wasted (i.e., allocated to a container that has no use for it); minimal progress should be guaranteed for each service class. A resource manager node runs a solver program that recomputes the optimal resource allocation (periodically or on demand) and notifies it to the local manager of each node.

Experiments reported in [Aron et al. 2000] show that cluster reserves can achieve better resource usage than a policy that reserves a fixed set of nodes for each service class, and that they provide good performance isolation between a set of service classes.

An extension of the principle of cluster reserves has been developed in the Sharc system [Urgaonkar and Shenoy 2004]. An application running on a cluster is decomposed in individual components, or *capsules*, each running on a single node. Each capsule imposes its own requirements in terms of performance and resource needs (the resources controlled by Sharc are CPU and network bandwidth, while cluster reserves only control CPU). Several application may concurrently share the cluster. A two-level resource manager (see 12.2.3) allocates the resources among the capsules. When a new application is started, the manager determines the placement of its capsules on the nodes, according to available resources. During execution, resources may be traded between capsules on a node (within a single application), depending on their current needs; thus a capsule that has spare resources may temporarily lend them to another capsule. Experience shows that this system effectively shares resources on a moderate size cluster.

Virtual Clusters

Clusters on Demand (COD) [Chase et al. 2003, Moore et al. 2002] is another attempt towards global resource management on clusters and grids. The hardware platform is a (typically large) collection of machines connected by a network. The approach taken here is to consider this platform as a utility, to be shared by different user communities with different needs, possibly running different software. Thus each such community receives a *virtual cluster* (in short, *vcluster*), i.e., an autonomous, isolated partition composed of a collection of nodes. The users of a *vcluster* have total control on it (subject to authorization), and may install a new software environment, down to the operating system.

The nodes that compose a *vcluster* are dynamically allocated. A node is usually a physical machine, but it may also be a virtual machine hosted by a physical node. A *vcluster* may expand or shrink, by acquiring or releasing nodes. This resizing may be at the initiative of the *vcluster*, to react to variations in load, or at the initiative of the system manager, according to the global management policy. Thus a negotiation protocol takes place between these two levels to resolve conflicts (e.g., if several *vclusters* need to expand at the same time).

A node is a stateful resource. When a node is reallocated, some of its state needs to be saved, and it may possibly be entirely reinstalled. A bootstrap mechanism allows an entirely new operating system to be installed at node reallocation.

Vclusters, like other forms of resource principals, are mechanisms. They may be used to implement various policies, both for resource allocation within a *vcluster* (reacting to load variations) and for global cluster management.

Another system based on the same principle is Oceano [Appleby et al. 2001].

12.2.3 Resource Managers

A *resource manager* is the entity responsible for managing resources. Depending on the system structure, it may take the form of a server process or of a passive object. In the first case, the requests are sent as messages; in the second case, they take the forms of procedure or method calls. The call may be implicit, i.e., triggered by an automatic mechanism; such is the case for operating systems resource managers (e.g., a memory manager is called by a page fault hardware trap). A resource manager may also be distributed, i.e., it is composed of several managers that collaborate using a group protocol (peer to peer or master-slave), while providing a single API to its users.

The role of a manager is to allocate resources at a certain level, by multiplexing resources available at a lower level. Thus, a complex system usually relies on a hierarchy of managers. The lowest level managers allocate physical resources such as CPU or memory, and deliver corresponding resources in a virtual form. These virtual resources are in turn allocated by higher level managers. For example, a low-level scheduler allocates CPU to kernel-level threads; these in turn are used as resources by user-level threads. A resource manager is usually associated with a management *domain* (10.1.2), which groups a set of resources subject to a common policy.

These hierarchical schemes may be exploited in various ways, according to the overall organization of resource management. In an organization using an elaborate model for resource principals, a global management level allocates resources between these principals. Then a local manager is associated with each principal (e.g., a virtual cluster) to allocate resources within that principal, which achieves isolation. In an organization based on multi-node containers (as described in Section 12.2.2) the manager hierarchy may correspond to a physical organization: a manager for a resource in a multi-node container relies on sub-managers for that resource on each node. The already mentioned Sharc system [Urgaonkar and Shenoy 2004], for instance, uses this two-level manager scheme.

12.3 Resource Management as a Control Process

Like other aspects of system administration, resource management may be viewed as a control process. The principles of control were introduced in 10.2.1. Recall that control may take two forms: open loop (or feed-forward), and closed loop (or feedback). Both approaches are used for resource management. However, open loop control relies on a prediction of the load and needs an accurate model of the controlled system, two requirements that are difficult to meet in practice. Therefore, most open-loop policies either make strong assumptions on the resource needs of an application, or apply to local decisions for which a policy may be defined a priori.

In 12.3.1, we examine the main issues of resource management algorithms. In 12.3.2, we discuss the main basic open loop policies used to manage resources. In 12.3.3, we present market-based approaches, another form of open loop management relying on a decentralized, implicit, form of global control. Resource management using feedback control is the subject of 12.4.

12.3.1 Resource Management Algorithms

Assume that the managed system is in a state in which its current resource principals meet a prescribed QoS specification using the resources already acquired (call it a "healthy" state). A control algorithm for resource management attempts to keep the system in a healthy state, using the three means of action outlined in 12.1.3: dynamic resource provisioning, admission control, and service degradation. These may be controlled through feed-forward, feedback, or through a mixed approach. In addition, the problem of bringing the system in an initial healthy state should also be solved.

Several common questions arise in the design of a control algorithm. We examine them in turn.

1) System States and Metrics

The first question is how to define a healthy state. In other words, what metric is used to assess the state of the system?

Recall (12.1) that QoS is specified by Service Level Objectives (SLOs), which are the technical expression of an SLA. SLOs involve high-level performance factors, such as global response time or global throughput. While these factors are related to client satisfaction, they cannot be directly used to characterize the state of a system, for which resource allocation indicators are more relevant. These indicators are more easily measured, and can be used for capacity planning and for controlling the system during operation.

The problem of SLA decomposition for performance QoS [Chen et al. 2007] is to derive low-level resource occupation factors from Service Level Objectives. Note that equivalent versions of this problem exist for other aspects of QoS, such as availability and security. These are discussed in Chapters 11 and 13, respectively. Here we only consider the performance aspects of QoS.

To illustrate this issue, consider a 3-tier implementation of an Internet service, for which SLOs are expressed as a maximum mean response time R and a minimum throughput T. For a given system infrastructure, the problem is to map these requirements onto threshold values for resource occupation at the different tiers:

$$(R, T) \mapsto (\eta_{http-cpu}, \eta_{http-mem}, \eta_{app-cpu}, \eta_{app-mem}, \eta_{db-cpu}, \eta_{db-mem})$$

where η_{*-cpu} and η_{*-mem} are the occupation rates of CPU and memory, respectively, for the three tiers: HTTP, Application, and Database.

Two main approaches have been proposed to solve the SLA decomposition problem.

- Using a model of the system to derive low-level resource occupation thresholds from high-level SLOs.
- Using statistical analysis to infer relevant system-related metrics from user-perceived performance factors.

The model-based approach relies on the ability to build a complete and accurate model of the system. This is a difficult task, due to the complexity of real systems, and to the widely varying load conditions. However, progress is being made; the most promising approach seems to be based on queueing network models. [Doyle et al. 2003] uses a simple queueing model to represent a static content Web service. [Chen et al. 2007] use a more elaborate queueing network to describe a multi-tier service with dynamic web content. This model is similar to that of [Urgaonkar et al. 2007], also described in Section 12.5.5.

The statistical analysis approach is fairly independent of specific domain knowledge, and may thus apply to a wide range of systems and operating environments. An example of this approach is [Cohen et al. 2004]. The objective is to correlate system-level metrics and threshold values with high-level performance factors such as expressed in SLOs. To that end, they use Tree-Augmented Naive Bayesian Networks, or TANSs [Friedman et al. 1997], a statistical tool for classification and data correlation. Experiments with a 3-tier e-commerce system have shown that a small number of system-level metrics (3 to 8) can predict SLO violations accurately, and that combinations of such metrics are significantly more predictive than individual metrics (a similar conclusion was derived from the experiments described in 12.5.4). The method is useful for prediction, but its practical use for closed loop control has not been validated.

2) Predictive vs Reactive Algorithms

Are decisions based on prediction or on reaction to real-time measurement through sensors? In the predictive approach, the algorithm tries to assess whether the decision will keep the system in a healthy state. This prediction may be based on estimated upper limits of resource consumption (using a model, as described above), or on the prior observation of a typical class of workload. Both approaches to prediction are useful for estimating mean values and medium-term evolution, but does not help in the case of load peaks. Thus a promising path seems to design algorithms that combine prediction with reaction and thus implement a mixed feed-forward-feedback control scheme.

3) Decision Instants

What are the decision instants? The decisions may be made periodically (with a predefined period), or may be linked to significant events in the system, such as the arrival or the termination of a request, or depending on measurements (e.g., if some load factor exceeds a preset threshold). These approaches are usually combined.

4) Heuristics and Strategies

While the design of a resource management algorithm depends on the specific features of the controlled system and of its load, a few heuristic principles apply to all situations.

- *Allocate resources in proportion of the needs.* As discussed above, the needs may be estimated by various methods. Techniques for proportional allocation are discussed in 12.3.2.

- In the absence of other information, attempt to *equally balance resource occupation.* An example illustrating this principle (load balancing algorithms) is presented in 12.3.2.

- *Shed load to avoid thrashing.* Experience shows that the best way of dealing with a peak load is to keep only a fraction of the load that can be serviced within the SLA, and to reject the rest. This is the main objective of admission control (see 12.5 for detailed examples).

Recall (12.1.3) that three forms of resource management algorithms may be used, in isolation or combined. The base for their decisions is outlined below.

In the case of resource provisioning, the decision is to allocate a resource to a principal (acting on behalf of a request or set of requests), either from an available resource pool, or by taking it from another principal. The decision is guided by the "proportional allocation" principle, possibly complemented by an estimate of the effect of the action on the measured QoS.

In the case of admission control, a request goes through an admission filter. The filtering decision (admit or reject) is based on an estimate of whether admitting the request would keep or not the system in a healthy state. There may be a single filter at the receiving end, or multiple filters, e.g., at the entry of each tier in a multi-tiered system. A rejected request may be kept by the system to be resubmitted latter, or to be granted when admission criteria are met again, e.g., after resources have been released. Alternatively, in an interactive system such as a web server, the request may be discarded and a rejection message may be sent to the requester, with possibly an estimate of a future acceptance time.

In the case of service degradation, the decision is to lower the service for some requests, with the goal of maintaining the system in a healthy state with respect to other requests. There are two aspects to the decision: how to select the "victim" requests, and by what amount to degrade the service.

In all cases, the decision involves an estimate of its impact on the state of the system. As explained in the discussion on system states, this estimate involves correlating resource utilization thresholds with SLOs, through the use of a model and/or statistical analysis.

12.3.2 Basic Open Loop Policies

Open-loop policies for resource management make decisions based on the knowledge of the resource needs, which may either be given a priori, or predicted by one of the techniques discussed above. Some of these policies are a base on which more elaborate resource management systems using feedback have been developed.

In this section, we first briefly present static reservation. We then go on to a discussion of some usual techniques for implementing proportional scheduling: allocating resources in proportion of specified needs. We finally illustrate balanced resource allocation with a brief overview of load balancing.

1) Reservation

One way to ensure that a service provider respects its SLA is to reserve for it all the resources that it may need, assuming the resource needs (or at least an upper bound of the needs for each kind of resource) are known in advance. Thus the resources of the system are statically divided between the different classes of services. Each provider also may use reservation to allocate its resource pool among the service requesters. If a service provider closes its activity, its resources become available, and may be reallocated to a new provider, if they satisfy its needs.

This method assumes that the total available amount of each resource is large enough to satisfy the aggregated demand. In addition, each reservation should be made by considering the worst case situation. Thus, if there is a large variation in resource demand, or if the worst case demand cannot be accurately estimated, a significant amount of resources may be wasted. This limits the applicability of static reservation; however, this drawback may be reduced by temporary trading unused resources, such as proposed in Sharc (12.2.2 and [Urgaonkar and Shenoy 2004]).

The same principle is used for capacity planning, i.e., estimating the global amount of resources of a platform, given its expected load and the SLAs. Capacity planning thus relies on a mapping of SLAs to resource occupation, as discussed above. If worst case estimates are used, overestimating the amount of resources leads to underutilization. To counter this risk, [Urgaonkar et al. 2002] propose a controlled overbooking strategy.

A common use of reservation is to guarantee QoS in network communications, by statically reserving network bandwidth and buffer space.

2) Proportional Scheduling

A frequent situation is one in which the policy to be implemented for sharing a resource is one of *proportional-share*, i.e., each requester gets a share of the resource proportional to a predefined ratio. This ratio may be fixed, or it may vary over time. This may be used at the global level, to allocate resources among various service classes, or within a service provider, to share resources among the requesters of that service.

A more precise statement of the requirements is as follows. Let there be n contenders for a (non-shared) resource, and define a set of ratios $r_1, \ldots r_n$, such that $0 \leq r_i \leq 1$ and $\sum_{i=1}^{n} r_i = 1$. The goal is that, at any time, each contender c_i should be granted a fraction r_i of the resource. If the resource is, for example, CPU time, this means that c_i gets a virtual processor whose speed is r_i times that of the actual CPU. If the resource is memory, c_i gets a fraction r_i of the available memory (possibly rounded to the nearest page limit if the memory is paged). This holds even if the r_i or n vary over time.

Priority-based methods. The problem of proportional scheduling has initially been attacked by assigning priorities to the contenders, in proportion to their assigned ratios. This is the classical method used for CPU scheduling in operating systems. In the simplest case, all processes have equal priority, and are ordered in a single run queue, scheduled by round robin with a fixed quantum q. This approximates the "shared processor" model (the limit case when q goes to 0), in which each process is granted a virtual processor of equal speed (the speed of the physical processor divided by the number of processes). If priorities are introduced, the scheduler orders the run queue by decreasing priority, with FIFO ordering within each priority.

A stronger bias in favor of high priority processes may be achieved by several means.

- Preemption. A new process preempts the processor if its priority is higher than that of the active process.
- Multiple queues. A different queue is used for each priority, with lower quantum values for high priority. The processor is granted to the first process of the non-empty queue with the highest priority, and preemption is applied.

While widely used in commercial systems, priority-based methods suffer from several drawbacks.

- The relative proportion of priorities has no simple relationship with the prescribed rates; thus priorities must be set by trial and error to approximate these rates.
- The system is subject to starvation. This may be prevented by having the priority of a process increase with its waiting time (to be restored to its initial value after it has been served), but this further complexifies the behavior of the system.
- For real-time systems, in which processes must meet deadlines, there is a risk of priority inversion. Consider a process *P0* of priority 0 (the highest), which is blocked by a mutex held by a process *P7* of priority 7. *P7* in turn is delayed by processes of intermediate priorities (4, 5, ...) and thus cannot release the mutex. As a consequence, *P0* may miss a deadline. Ad hoc techniques have been proposed to avoid priority inversion (e.g., increasing the priority of a process while it holds a mutex), but the competition between processes is not always apparent, as the mutex may be hidden under several layers of abstraction in an operating system routine.

Therefore alternative methods have been proposed for achieving proportional rate resource allocation.

Lottery scheduling. A more accurate and efficient solution is provided by *lottery scheduling* [Waldspurger and Weihl 1994], a mechanism that has initially been proposed for proportional-share CPU scheduling. Each contender c_i gets a number of tickets proportional to the ratio r_i; each ticket carries a different value. Each time an allocation should be made (e.g., at periodic instants, or at each new request, etc.), a lottery (i.e., a random drawing in the set of tickets, with equal chances) is held, and the resource is allocated to the holder of the winning ticket.

Lottery scheduling has been extended and adapted to other resources such as memory or network bandwidth, and to the allocation of multiple resources [Sullivan et al. 1999].

It can be shown that the allocation is probabilistically fair, in the sense that the expected allocation to clients is indeed proportional to the number of tickets they hold, i.e., to the assigned ratios. While a discrepancy may exist between the actual proportion of the resource allocated to a client and the expected one, the accuracy improves like $\sqrt{N}$, where N is the number of drawings. The overhead cost of a random drawing is very small.

Several additional mechanisms may be implemented, such as temporary transfer of tickets between clients, e.g., when a client cannot use its share of tickets because it waits for a message or another resource. This method may be used to prevent priority inversion.

3) Load balancing

In the context of cluster computing, *load balancing* is a technique used to distribute a set of tasks among the nodes of a cluster. A typical situation is that of a cluster-based web server which receives requests from clients. A common architecture consists of a front-end node (the switch) and a set of homogeneous back-end (or server) nodes on which the actual work is performed. Client requests are directed to the switch, which distributes them among the server nodes (Figure 12.3(a)). This scheme is also used between the successive tiers of a multi-tier system, in which each tier runs on a set of nodes (Figure 12.3(b)).

From the point of view of a client, the front-end node and the servers may be considered together as a virtual server, which globally provides the specified service. The clients are unaware of the distribution of requests among the servers.

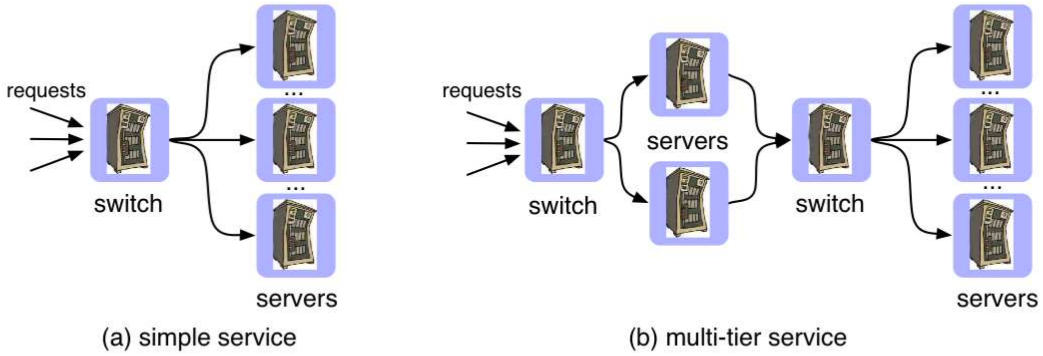

Figure 12.3. Load balancing

The objectives of a load balancing policy are the following.

- Keeping the servers active; no server should be idle while unsatisfied requests are pending. A heuristic to achieve this goal is to distribute the load "evenly" among the servers (in a sense to be illustrated by the examples that follow).
- Increasing the overall performance, e.g., reducing the mean response time of the requests, or maximizing the throughput.
- As an accessory goal, exploiting the existence of multiple servers to increase availability; however, for this goal to be achieved, the switch itself should be replicated to preserve its function in case of failure.

These objectives are subject to a number of constraints, among which the following.

- If the requests are not independent (e.g., if they activate subtasks of a parallelized global task), the precedence constraints between the subtasks should be respected.
- If client-specific state is maintained by the server, the requests from a given client should be run on the same server (this property is called persistence[3]).

[3]This term has a different meaning from that used in Chapter 8.

The simplest load balancing policy is Round-Robin: successive requests are sent to servers in a sequential, circular order ($S_1, S_2, \ldots, S_n, S_1, \ldots$). This policy is static, in the sense that it depends neither on the characteristics of the requests nor on the current load of the servers. Round-Robin gives acceptable results for a “well-behaved” load, in which arrival and service times are regularly distributed (e.g., following an exponential law). However, as pointed out in 12.1.3, real loads usually exhibit a less predictable behavior, with high variability in both arrival rates and service times. For such irregular loads, Round-Robin may cause the load on the servers to be heavily unbalanced. For this reason, dynamic policies have been investigated. Such policies base their decisions on the current load of the back-end servers and/or on the content of the requests.

A first improvement on basic Round-Robin is to take the current server load into account, by assigning each server a weight proportional to its current load (server load estimates are periodically updated). Thus lightly loaded nodes are privileged for accepting new requests. Server load is estimated by CPU or disk utilization, by the number of open connections, or by a combination of these factors. This policy, called Dynamic Weighted Round-Robin, is simple to implement, since it does not involve request analysis. The same principle is applied for balancing the load between heterogeneous nodes with different performance characteristics: the nodes with higher performance are privileged.

In addition to server load, more elaborate policies also take request contents into account. One of the first attempts in this direction was the Locality Aware Request Distribution (LARD) proposed in [Pai et al. 1998]. The main objective is to improve cache hit rates in the servers, thus reducing mean response time. To do so, the name space that defines the data to be fetched from the database (e.g., the URLs) is partitioned, for instance through a hash-coding function, and each partition (or target) is assigned to a particular back-end server, thus improving cache locality. LARD combines load-aware balancing and high locality, by giving priority to lightly loaded servers for the first assignment of a target. In addition, server activity is monitored in order to detect substantial load imbalance, and to correct it by reassigning targets between servers.

More recent work, ADAPTLOAD [Zhang et al. 2005], attempts to improve on LARD by dynamically adjusting the parameters of the load distribution policy, in order to react to peaks in the load. To do so, ADAPTLOAD uses knowledge of the history of request distribution.

A survey of load balancing policies for clustered web servers may be found in [Cardellini et al. 2002].

12.3.3 Market-based Resource Management

The presentation of the goals and policies of resource allocation (12.1.2) suggests an analogy with an economic system in which customers and suppliers are trading goods and services. A number of projects have tried to exploit this analogy (see [Sutherland 1968] for an early attempt), but none of these proposals has actually achieved wide application.

In this section, we summarize the main problems of market-based resource management, and we illustrate them with a case study.

1) Issues in Market-based Resource Management

With the emergence of large distributed systems shared by a community of users (clusters, grids, PlanetLab [Bavier et al. 2004]), the idea of using market mechanisms for resource allocation has been revived in several recent proposals. Drawing on the analogy with an economic system, the goal is to assign resources to requests, while satisfying a set of constraints. These include limitations on both the amount of resources available and on the payment means of the requesters, as well as meeting the goals of both providers and requesters. Since these goals may be contradictory (as pointed out in 12.1.2), each party will attempt to maximize its own utility function, based on a valuation of resources, and a global compromise should be sought.

The following main issues need to be considered.

- How to assign a value to a request? Note that this value may be a function of time, since a request may prove of no value (or may even cause a penalty) if satisfied too late.
- How to assign a value to a resource? This value may again vary, depending on availability and on demand (e.g., through a form of bidding).
- Does there exist an equilibrium point between the requirements of resource providers and requesters?
- How is value measured? Economic analogies suggest bartering (trading a resource for another one), or using a form of currency, real or virtual.
- What mechanisms are used for contracting? Again, analogies suggest the notions of brokering (using intermediate agents), using leases (granting a resource for a fixed period of time), etc.
- How to prevent and detect misbehavior (cheating, stealing, overspending)?

These issues are still being explored, and there is no universally accepted solution. The following example gives an idea of some orientations of current research.

2) Case Study: Currency-based Resource Management

We present this case study in two steps. We first describe Sharp (Secure Highly Available Resource Peering) [Fu et al. 2003], a framework for distributed resource management in an Internet scale computing infrastructure. We then show the use of this framework to acquire resources by means of virtual currency.

Sharp defines abstractions and mechanisms for coordinated resource allocation on a system composed of a collection of sites, where a *site* is a local resource management domain, which may range from a single machine to a cluster. This framework is designed for building resource management systems that are flexible (allowing for various policies and trade-offs), robust (tolerant to partial failures), and secure (resisting to attacks).

Sharp relies on a hierarchy of resource managers, based on two roles: authority and broker. An *authority* is the entity that has actual control over a set of resources and maintains their status. A *broker*[4] is an intermediate entity that has a delegation from an authority and receives requests for the resources managed by that authority; an authority may grant delegation to several brokers.

A *claim* is an assertion stating that some principal has control on some set of resources over some time interval. A claim may be soft, i.e., it is only a promise that may possibly not be fulfilled; or it may be hard, i.e., the resources are actually granted. A soft claim is concretely represented by a *ticket*, while a hard claim is represented by a *lease*. Both tickets and leases are delivered by certified authorities, and are cryptographically protected to avoid forgery.

A typical situation is that represented on Figure 12.4 (ignore the left part, for the time being): the authority is a resource provider (e.g., a platform provider), and resource principals are hosted services. To get resources, a hosted service uses a two-phase process: it first gets a ticket from a local resource manager (a broker acting for the resource provider); it then uses this ticket to get a lease from that resource provider. It then may use the claimed resources until the end of the specified period.

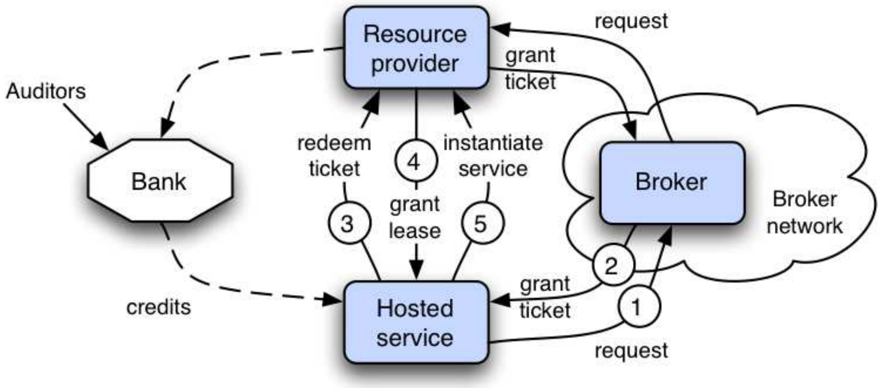

Figure 12.4. Resource allocation in SHARP (from [Fu et al. 2003, Irwin et al. 2005])

After the hosted service has successfully redeemed his ticket with the resource provider, it is granted the set of requested resources, and may then run (serve its own clients) using these resources until the end of the lease period.

The distinction between tickets and leases allows flexibility for resource allocation by delaying the resource allocation decision to the last possible time. Tickets may also be transferred from one resource principal to another one; for instance, a principal may create children and delegate part of its tickets to them; the children may do the same, thus creating a resource delegation tree.

Over-subscription (delivering more tickets than could simultaneously be redeemed) is possible, allowing better satisfaction and improving resource availability, at the risk of occasionally refusing or delaying the granting of a lease. Over-subscription may occur at all levels, i.e., a principal may delegate more tickets than it holds. When processing a tree of nested claims, a simple algorithm examines the claims in bottom up order and locates conflicts due to over-subscription at the lowest common ancestor, thus ensuring

[4] called *agent* in [Fu et al. 2003].

accountability for conflicts.

Tickets and leases are signed by the authority that delivered them. This makes them unforgeable, non-repudiable, and independently verifiable by third parties (e.g., for an audit). The finite duration of leases improves the system's resilience since any resource shall be freed in bounded time, even if its manager fails.

Experiments have shown that the proposed framework improves resource utilization. However, to achieve that goal, the various parameters (degree of over-subscription, duration of the claim, structure of the delegation trees) need to be carefully selected, using experience. The implementation of claims by tickets and leases provides flexibility and allows various trade-offs to be explored, while signatures improve security and traceability.

We now describe a virtual currency model built on top of Sharp, and used in a project called Cereus [Irwin et al. 2005]. While resources may be traded through bartering (identifying mutual coincidence of needs between parties), the introduction of currency allows external regulation of resource allocation policies within a community.

Cereus currency is self-recharging, i.e., spent amounts are restored to each consumer's budget after some time. Each consumer is allocated a number of *credits* in virtual currency (this initial allocation might itself be purchased with actual money). If a consumer has a budget of c credits, it may use them to acquire resources through contracts or auctions. Credits are automatically recharged after a fixed amount of time (say r) from the moment they are spent. A consumer may not spend or commit more than c over any interval of time of length r, and it may not accumulate more than c credits at any time. This prevents hoarding, which might allow malicious actions such as starving other users. Note that this system behaves like lottery scheduling (fair share proportional allocation, see 12.3.2) if the recharge time is set to a very small value. For larger recharge time, the users have more freedom to schedule their requests over time.

In a system in which resources are acquired through auctions, users spend their credits by bidding for resources. A sum committed to a bid at time t is recharged at time $t + r$, which encourages early bidding and discourages canceled bids. Cereus uses an auction protocol in which a broker posts a call price based on recent history, and returns resources in proportion to the bid if the demand exceeds the amount available.

In order to regulate the use of currency, Cereus coordinates all currency transactions through a trusted banking service. Currency is implemented using the claim mechanism provided by Sharp: c credits are represented by a claim for c units of a special type, virtual currency. This claim, called a credit note, is an instance of a Sharp ticket. Credit notes are issued to consumers by trusted banks (left part of Figure 12.4). To spend currency, a consumer delegates control of its credits to a supplier, or to a broker that conducts an auction. In return, the consumer (hosted service) gets resource tickets that it may redeem for leases on the acquired resources (middle part of Figure 12.4). The credits are spent by the brokers to get resources from the suppliers, and are ultimately returned to the banks to be reissued to consumers.

An auditing system allows malicious behavior (overspending, attempting to recharge credits before recharge time) to be detected, since all the needed information is contained in the credit notes.

Other aspects of the economic approach to resource management have been investigated.

The Tycoon project [Feldman et al. 2005] proposes a resource allocation mechanism for shared clusters, based on a continuous reevaluation of users' preferences through an ongoing auction. They explore existence conditions for an equilibrium (in the sense that resources may be allocated in a way that maximizes each user's utility function) and propose strategies for reaching equilibrium with small overhead. Tycoon is close to Cereus, but places emphasis on agility (fast reaction to change), while Cereus guarantees medium-term stability through leasing.

The work of [Balazinska et al. 2004] proposes a load management mechanism for loosely coupled distributed systems, through contracts between the participants. Their price-setting mechanism produces "acceptable" allocations, in the sense that *no* participant is overloaded if spare resources are available, and that overload, if it occurs, is shared by *all* participants. This achieves a form of fairness.

12.4 Feedback-controlled Resource Management

An overview of the control aspects of systems management has been presented in 10.2.1. Here we concentrate on the aspects related to performance and resource management, using feedback control. As noted in 10.2.1, the main advantage of using feedback vs. feedforward control is that it does need a detailed model of the controlled system, and is well adapted to an unpredictable behavior of the load.

In order to apply feedback control to resource management, one needs to specify the model used to represent the system, the variables, both observed and controlled, and the sensors and actuators that allow the controller to interact with the variables, and the policy implemented by the controller. We examine these aspects in turn. See [Hellerstein 2004], [Diao et al. 2005] for more details, and the book [Hellerstein et al. 2004] for an in-depth study of the subject.

12.4.1 Models

There are four main approaches to modeling the resource management and performance aspects of a computing system.

The first approach is empirical, and has been often used, due to the complexity of the actual systems. For instance, in an early effort to design a control algorithm to prevent thrashing in paged virtual memory systems [Brawn and Gustavson 1968], the evolution of the system was represented by a transition graph between three states (normal, underload, overload), characterized by value ranges of resource occupation rates. The controller attempted to prevent the system from getting into the overload state by acting on the degree of multiprogramming (the number of jobs admitted to share memory). Despite its simplicity, this approach proved remarkably efficient. Other empirical models are based on observation and represent the behavior of the system using curve-fitting methods (see an example in [Uttamchandani et al. 2005], using a piecewise linear approximation to model the behavior of a large scale storage utility).

The second approach is to consider the controlled system as a black box, i.e., a device whose internal organization is unknown, and which only communicates with the outside world through a set of sensors and actuators. One also needs to assume that the behavior of the system follows a known law. A common assumption is to consider that the system is linear and time-invariant (see 12.5.4, note 7 for a definition of these terms). Then the numerical values of the model's parameters are determined by performing appropriate experiments (e.g., submitting the system to a periodic input and measuring the response on the output channels), a technique known as system identification. An example of the use of this technique may be found in 12.5.4.

The third approach is based on queueing theory. It considers the system as a queueing network, i.e., a set of interconnected queues, each of which is associated with a specific resource or set of resources. A request follows a path in this network, depending on the resources it needs for its completion. In order to fully specify the model, one needs to define the service law and the inter-arrival law for each queue, and the transition probabilities from one queue to another one. Both analytical and computational solutions for queueing problems have been developed (a book on the applications of queueing theory to computing system is [Lazowska et al. 1984], also available on line[5]).

Models based on queueing systems are commonly used to build workload generators, in order to study the impact of various load parameters on the performance of an application. These models fall into two main classes depending on how job (or request) arrivals are organized. In a closed model, new arrivals are conditioned by job completions, since a job goes back to a fixed size pool upon completion, to be resubmitted later. In an open model, new jobs arrive independently of job completions and completed jobs disappear from the system. Partly open models combine the characteristics of open and closed models, as only part of the job population is recycled upon completion. Open and closed models exhibit vastly different behaviors, as stressed in [Schroeder et al. 2006]. Examples of the use of queueing models for QoS control are presented in this chapter: a closed model in 12.5.5, and an open model in 12.5.6.

The last approach is to represent the behavior of the system by an analytical model. This approach is only applicable when the organization of the controlled system is sufficiently simple and well known. This is often the case for single-resource problems, such as CPU scheduling, as illustrated by the example of 12.5.6

12.4.2 Sensors and Actuators

In order to apply feedback control to resource management, one needs to specify the variables, both observed and controlled, the sensors and actuators that allow the controller to interact with the variables, and the policy implemented by the controller. We first examine the common aspects related to variables.

Since the objective of a resource management policy is to improve QoS, using QoS-related factors as observed variables seems a natural choice. Such variables depend on the relevant aspect of QoS, such as response time (mean, variance, maximum value), throughput, jitter, etc. However, these variables are not always easy to observe (e.g., how to observe the mean response time for a large, widely distributed user population?). Therefore, variables related to resource utilization, which are more readily accessible, are often used. Resource utilization metrics include CPU load, memory occupation, power consumption, I/O channel rate, percentage of servers used in a pool, etc.

[5] http://www.cs.washington.edu/homes/lazowska/qsp/

The problem of relating these variables to the measure of QoS is discussed in 12.3.1.

Actuators for resource management may take a variety of forms:

- Allocate or release a unit of allocation, e.g., an area of storage, a time slice on a CPU, a whole server in a server farm, a share of bandwidth on a communication channel, etc.

- Change the state of a resource, e.g., turn off a server, or modify the frequency of a CPU, in order to reduce energy consumption.

- Add or remove a resource to or from a pool. The resource may be physical (e.g., a server or a memory board), or virtual (e.g., a virtual machine or a virtual cluster).

- Allow or deny a principal the right to compete for resource allocation (in a certain resource pool). One common example is adjusting the multiprogramming level, i.e., the number of jobs or tasks allowed to run concurrently on a node.

We briefly mention specific actuators associated with two resource management techniques: admission control and system reconfiguration.

Admission control relies on regulating a flow of incoming requests. If the decision (admit or reject) depends on the contents of the request, each request must be considered individually. Otherwise, for "anonymous" admission control, one may rely on a method only based on the incoming flow rate, such as the token bucket, a flow-throttling device used for network control. Each token in the bucket gives a right to admit a unit of flow (e.g., a fixed number of bytes); the token is consumed when the unit is admitted. Tokens may be added to the bucket, up to a maximum capacity; if the capacity is exceeded, the bucket "overflows" and no token is added. Thus the two controlling parameters are the token issue rate and the capacity of the bucket (Figure 12.5). Token buckets are used, for example, to regulate admission control in the SEDA system (12.5.2).

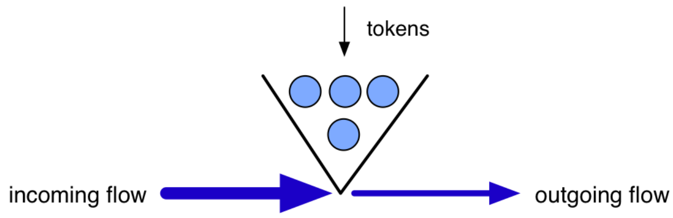

Figure 12.5. The token bucket: an actuator for anonymous admission control

Reallocating resources may involve system reconfiguration. For example, if additional resource units are added to a pool, connections between these units and the rest of the system need to be set up. Techniques for dynamic system reconfiguration are examined in 10.5.

12.4.3 Control Algorithms

In many instances, specially when no model of the system is available, resource management algorithms using feedback are not based on control theory, but use a rule-based approach. For instance, a simplistic admission control algorithm may use thresholds: if the measured QoS indicator (e.g., mean response time) is less than a lower limit, admit; if it exceeds a higher limit, reject. A number of experiments (see [Hellerstein et al. 2004]) have investigated more elaborate algorithms, based on control theory.

The control of computing systems is a new area, which presents several challenges. The controlled systems are usually non-linear, and they are submitted to a highly variable workload. In addition, sensors and effectors are themselves fairly complex systems, and their own dynamics should be integrated into the model of the controlled system.

While many common control algorithms are Single Input-Single Output (SISO), experience (see 12.5.4) has shown that Multiple Input-Multiple Output (MIMO) models more adequately reflect the behavior of complex computing systems. Control algorithms use a proportional (P) or Proportional Integral (PI) control law, assuming that the controlled system is linear, or possibly piecewise linear. Control laws based on the Derivative (D) are seldom used, because they tend to overreact to steep load variations, which are common in Internet services.

Case studies of resource management algorithms based on control theory are presented in 12.5.4 and 12.5.6.

12.4.4 Problems and Challenges

The application of feedback control methods to resource management and QoS in computer systems is still in its infancy. A number of problems still need to be solved.

- Accurate models of real, large scale computing systems are difficult to build, and the performance factors of these systems, as perceived by the clients, are difficult to measure. As a consequence, the correlation between QoS factors and controlled parameters (as set by actuators) is not easy to establish.
- The load on real system has complex, time dependent characteristics, and is difficult to model, even if realistic load generators are now available.
- Most of the methods and tools for solving control problems have been developed for linear systems. However, real systems such as Internet services exhibit highly non-linear behavior, because of such phenomena as saturation and thrashing.

More generally, the communities of computer science and control systems still have different cultures, and progress is needed in the mutual understanding of their respective concepts and methods. Joint projects, examples of which are mentioned in this chapter, should contribute to this goal.

12.5 Case Studies in Resource Management

In order to illustrate the concepts and techniques presented in the previous sections, we examine six case studies, which are representative of the variety of situations and approaches.

Their main features are described in Table 12.1.

sect.	appl.	infrastr.	model	sensors	actuators	diff.	reference
12.5.1	IS[1]	cluster	empiric	resp. time	AC[4], RP[5]	yes	[Blanquer et al. 2005]
12.5.2	IS[1]	cluster	empiric	resp.time	AC[4], RP[5]	yes	[Welsh and Culler 2003]
12.5.3	IS[1]	server	empiric	CPU util.	AC[4]	no	[Cherkasova and Phaal 2002]
12.5.4	WS[2]	server	black-box + ident.	CPU util. mem. util.	MaxClient KeepAlive	no	[Diao et al. 2002a]
12.5.5	IS[1]	cluster	queues	resp. time	AC[4], RP[5]	yes	[Urgaonkar et al. 2007]
12.5.6	RT[3]	server	analytic	CPU util. miss ratio	RP[5] (CPU scheduling)	yes	[Lu et al. 2002]

[1]Internet service
[2]Web server
[3]Real time
[4]Admission control
[5]Resource provisioning

Table 12.1. Case studies in resource management

The case studies are classified according to the following criteria: nature of the application (most applications are Internet services, with the special case of a simple web server); supporting infrastructure (single machine or cluster); model (empirical, analytic, "black-box" with parameter identification); observed and controlled variables (through sensors and actuators, respectively); support for differentiated service.

There are two main approaches to implement a resource management policy.

- Considering the managed system as a black box and installing the management system outside this black box. Examples of this approach are developed in 12.5.1 and 12.5.4.
- Embedding the management system within the managed system; this may be done at various levels, e.g., operating system, middleware, or application. Examples of this approach are developed in 12.5.5 and 12.5.6.

The first approach has the advantage of being minimally invasive, and to be independent of the internal structure of the managed system, which may possibly be unknown, as is the case with legacy systems. The second approach allows a finer grain control of the management policy, but involves access to the internals of the managed systems. The resource management programs also need to evolve together with the application, which entails additional costs.

A mixed approach is to consider the system as an assembly of black boxes, and to install the management system at the boundaries thus defined, provided that the architecture of the managed system is indeed explicit. An example is presented in 12.5.2.

12.5.1 Admission Control using a Black-Box Approach

As an example of the black-box approach, we present Quorum [Blanquer et al. 2005], a management system for cluster-based Internet services. Quorum aims at ensuring QoS guarantees for differentiated classes of service. For each class of service, the SLA is composed of two Service Level Objectives (SLO): minimum average throughput, and maximum response time for 95% of the requests. The SLA may only be guaranteed if it is feasible for the expected load and the cluster capacity, i.e., if 1) the clients of each class announce the expected computation requirements for the requests of the class; 2) the incoming request rate for each class is kept below its guaranteed throughput; and 3) the resources of the cluster are adequately dimensioned. In line with the black-box approach, SLA guarantees are defined at the boundary of the cluster.

QUORUM is implemented in a front-end node sitting between the clients and the cluster on which the application is running; this node acts as a load balancer (12.3.2), and is connected to the nodes that support the first tier of the application.

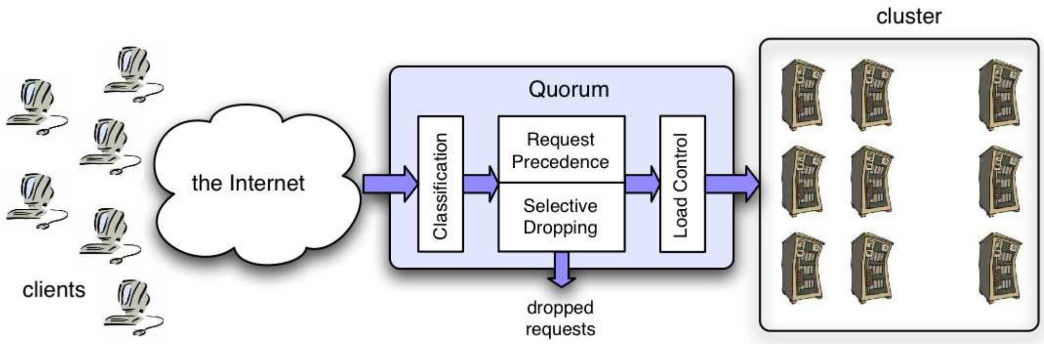

Figure 12.6. The architecture of QUORUM (from [Blanquer et al. 2005])

QUORUM is composed of four modules, which run on the front-end node (Figure 12.6). Their function is as follows (more details are provided further on).

- The *Classification* module receives all client requests, and determines the service class of each request.
- The *Request Precedence* module determines which proportion of the requests within each class will be transmitted to the cluster.
- The *Selective Dropping* module performs admission control, by discarding the requests for which the specified maximum response time cannot be guaranteed.
- The *Load Control* module releases the surviving requests into the cluster, and determines the rate of this request flow in order to keep the load of the cluster within acceptable bounds.

Thus QUORUM controls both the requests to be forwarded to the cluster and the flow rate of these requests.

Load Control regulates the input rate of the requests through a sliding window mechanism similar to that of the TCP transport protocol. The window size determines the number of outstanding requests at any time; it is recomputed periodically (a typical period is 500 ms, a compromise between fast reaction time and significant observation time). The observed parameter is the most restrictive response time for the set of service classes. The current algorithm increments or decrements linearly the window size to keep the response time within bounds.

Request Precedence virtually partitions the cluster resources among the service classes, thus ensuring isolation between classes. The goal is to ensure that the fraction of the global cluster capacity allocated to each class allows the throughput guarantee for this class to be met. The resource share allocated to each class is computed from the guaranteed throughput for this class and the expected computation requirements of the requests. Should the requests for a class exceed these requirements (thus violating the client's part of the contract), the throughput for that class would be reduced accordingly, thus preserving the other classes' guarantees.

In summary, Request Precedence ensures the throughput SLO for each class, while Load Control ensures the response time SLO. Selective Dropping discards the requests for which the response time SLO cannot be achieved under the resource partition needed to guarantee throughput.

Experience with a medium-sized cluster (68 CPUs) and a load replayed from traces of commercial applications shows that Quorum achieves its QoS objectives with a moderate performance overhead (3%), and behaves well under wide fluctuations of incoming traffic, or in the presence of misbehaving classes (i.e., exceeding their announced computation requirements).

Other systems have experimented with the black-box approach for 3-tier applications.

Gatekeeper [Elnikety et al. 2004] does not provide QoS guarantees, but improves the overall performance of the system by delaying thrashing, using admission control. In the absence of a client-side contract, Gatekeeper needs a preliminary setup phase in order to determine the capacity of the system.

Yaksha [Kamra et al. 2004] is based on a queueing model of the managed system, and uses a PI (Proportional Integral) feedback control loop to pilot admission control. It does not provide service differentiation. The SLA only specifies a maximum response time requirement; throughput is subject to best effort. The system is shown to resist to overload, and to rapidly adjust to changes in the workload.

12.5.2 Staged Admission Control

Seda (Staged Event-Driven Architecture [Welsh et al. 2001], [Welsh and Culler 2003]) is an architecture for building highly concurrent Internet services. High concurrency aims at responding to massive request loads. The traditional approach to designing concurrent systems, by servicing each request by a distinct process or thread, suffers from a high overhead under heavy load, both in switching time and in memory footprint. Seda explores an alternative approach, in which an application is built as a network of event-driven stages connected by event queues (Figure 12.7(a)). Within each stage, execution is carried out by a small-sized thread pool. This decomposition has the advantages of modularity, and the composition of stages through event queues provide mutual performance isolation between stages.

The structure of a stage is shown on Figure 12.7(b). A stage is organized around an application-specific event handler, which schedules reactions to incoming events, by activating the threads in the pool. Each invocation of the event handler treats a batch of events, whose size is subject to control, as explained later.

The event handler acts as a finite state machine. In addition to changing the local state, its execution generates zero or more events, which are dispatched to event queues of

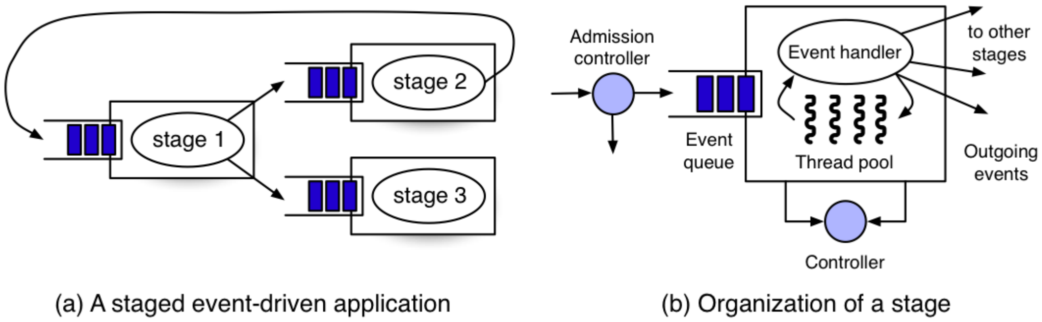

Figure 12.7. The architecture of SEDA (from [Welsh and Culler 2003])

other stages. To ensure performance, the event handling programs should be non-blocking; thus a non-blocking I/O library is used. Note that this structure is similar to that of Click [Kohler et al. 2000], the modular packet router described in 4.5.2.

Two forms of control are used within each stage, in order to maintain performance in the face of overload: admission control is applied to each event queue, and resource allocation within a stage is also subject to control.

Doing admission control on a per stage basis allows focusing on overloaded stages. The admission control policy usually consists in limiting the rate at which a stage accepts incoming events to keep the observed performance at that stage within specified bounds. If an event is rejected by this mechanism, it is the responsibility of the stage that emitted the event to react to the rejection (e.g., by sending the event to another stage, a form of load balancing). In the experiments described, performance metrics is defined by the 90th-percentile response time, smoothed to prevent over-reaction in case of sudden spikes. The policy also implements service differentiation, by defining a separate instance of the admission controller for each customer class. Thus a larger fraction of the lower class requests are rejected, ensuring service guarantees to the higher classes.

Within a stage, resources may be controlled by one or more specialized controllers. One instance is the thread pool controller, which adjusts the number of threads according to the current load of the stage, estimated by the length of the event queue. Another instance is the batching controller, which adjusts the size of the batch of events processed by each invocation of the event handler. The goal of this controller is to maintain a trade-off between the benefits of a large batching factor (e.g., cache locality and task aggregation, which increase throughput), and the overhead it places on response time. Thus the controller attempts to keep the batch size at the smallest value allowing high throughput.

Experiments on a SEDA-based mail server [Welsh and Culler 2003] have shown that the admission control controller is effective in reacting to load spikes, and keeps the response time close to the target. However, the rejection rate may be as high as 80% for massive load spikes. It has also been shown that per-stage admission control rejects less requests than single point admission control.

Another policy for responding to overload is to degrade the QoS of the request being served, in order to reduce the rejection rate at the expense of lower quality (see 12.1.3). Service degradation gives best results when coupled with admission control.

In its current state, SEDA does not attempt to coordinate the admission control decisions made at the different stages of an application. Therefore, a request may be dropped at a late stage, when it has already consumed a significant amount of resources. The work presented in the next section addresses this problem.

12.5.3 Session-Based Admission Control

Session-Based Admission Control [Cherkasova and Phaal 2002] is a method used to improve the performance of commercial web sites by preventing overload. This is essentially an improvement of a basic predictive admission control method based on the individual processing of independent requests, in which the admission algorithm keeps the server close to its peak capacity, by detecting and rejecting requests that would lead to overload.

This method is not suited for a commercial web site, in which the load has the form of sessions. A *session* is a sequence of requests separated by think intervals, each request depending on the previous ones (e.g., browsing the site, making inquiries, moving items into the cart, etc.). If the requests are admitted individually, request rejection may occur anywhere in a session. Thus some sessions may be interrupted and aborted before completion if retries are unsuccessful; for such a session, all the work done till rejection will have been wasted. As a consequence, although the server operates near its full capacity, the fraction of its time devoted to useful work may be quite low for a high server load (and thus a high rejection rate).

Therefore a more appropriate method is to consider sessions (rather than individual requests) as units of admission, and to make the admission decision as early as possible for a session. The admission criterion is based on server utilization, assuming that the server is CPU-bound (the criterion could be extended to include other resources such as I/O or network bandwidth).

A predefined threshold U_{max} specifies the critical server utilization level (a typical value is 95%). The load of the server is predicted periodically (typically every few seconds), based on observations over the last time interval:

$$U^{i+1}_{predicted} = f(i+1), \text{ where } f \text{ is defined by: } \quad \begin{cases} f(1) = U_{max} \\ f(i+1) = (1-k)f(i) + kU^{i}_{measured} \end{cases}$$

Thus the predicted load for interval T_{i+1} is estimated as a weighted mean of the previous estimated load and the load actually measured over the last interval T_i. The coefficient k (the weight of measurement versus extrapolation) is set between 0 and 1. The admission control algorithm is as follows.

- if $U^{i+1}_{predicted} > U_{max}$, then any new session arriving during T_{i+1} will be rejected (a rejection message is send to the client). The server will only process the already accepted sessions.
- if $U^{i+1}_{predicted} \leq U_{max}$, then the server will accept new sessions again.

The weighting factor k allows the algorithm to be tuned by emphasizing responsiveness (k close to 1) or stability (k close to 0). A responsive algorithm (strong emphasis on current observation) tends to be restrictive since it starts rejecting requests at the first sign of overload; the risk is server underutilization. A stable algorithm (strong emphasis on past history) is more permissive; the risk is a higher rate of abortion of already started sessions if the load increases in the future (a session in progress may be aborted in an overloaded server, due to timeouts on delayed replies and to limitation of the number of retries).

This algorithm has been extensively studied by simulation using SpecWeb, a standardized benchmark for measuring basic web server performance. Since sessions (rather than individual requests) are defined as the resource principals, an important parameter of the load is the average length (number of requests) of a session. Thus the simulation was done with three session populations, of respective average length 5, 15 and 50. We present a few typical results (refer to the original paper for a detailed analysis).

With a fully responsive admission control algorithm ($k = 1$), a sampling interval of 1 second, and a threshold $U_{max} = 95\%$

1. The throughput for completed sessions is improved for long sessions (e.g., from 80% to 90% for length 50 at 250% server load[6]); it is fairly stable for medium length sessions, and worse for short sessions. This may be explained by the fact that more short sessions are rejected at high load, with a corresponding rise of the rejection overhead.

2. The percentage of aborted sessions falls to zero for long and medium sessions, at all server loads. It tends to rise for short sessions at high loads (10% at 250% load, 55% at 300% load).

3. The most important effect is the increase in useful server utilization, i.e., the fraction of processor time spent in processing sessions that run to completion (i.e., are not aborted before they terminate). This factor improves from 15% to 70%, 85% and 88% for short, medium and long sessions, respectively, for a 200% server load. This is correlated with result 2 (low rate of aborted sessions).

Lowering the weighting factor k (going from responsive to stable) has the expected effect of increasing the useful throughput at moderate load, but of decreasing it at higher load, where more sessions are aborted due to inaccurate prediction (the threshold load value is about 170% for 15-request sessions). This suggests using an adaptive policy in which k would be dynamically adjusted by observing the percentage of refused connections and aborted requests. If this ratio rises, k is increased to make the admission policy more restrictive; if it falls to zero, k is gradually decreased.

Another dynamic strategy consists in trying to predict the number of sessions that the server will be able to process in the next interval, based on the observed load and on workload characterization. The server rejects any new session above this quota.

Both dynamic strategies (adapting the responsiveness factor k and predicting the acceptable load) have been tried with different load patterns drawn from experience ("usual day" and "busy day"). Both strategies were found to reduce the number of aborted sessions and thus to improve the useful throughput, the predictive strategy giving the best results.

[6]the server load is the ratio between the aggregated load generated by the requests and the server processing capacity.

12.5.4 Feedback Control of a Web Server

The work reported in [Diao et al. 2002a], [Diao et al. 2002b] is an early attempt to apply feedback control to the management of an Internet service (a single Apache web server delivering static content).

The controlled variables are CPU and memory utilization, denoted by CPU and MEM, respectively, which are easily accessible low-level variables, assuming that the settings of these variables are correlated to QoS levels such as expressed in an SLA. The controlling variables are those commonly used by system administrators, namely *MaxClients* (MC), the maximum number of clients that can connect to the server, and *KeepAlive Timeout* (KA), which determines how long an idle connection is maintained in the HTTP 1.1 protocol. This parameter allows an inactive client (or a client whose think time has exceeded a preset limit) to be disconnected from the server, thus leaving room to serve other clients.

The Apache web server is modeled by a black box, with two inputs (MC and KA) and two outputs (CPU and MEM). The behavior of this black box is assumed to be linear and time-invariant[7]. In a rough first approximation, CPU is strongly correlated with KA, and MEM is strongly correlated with MC. This suggests modeling the evolution of the server by the two Single Input, Single Output (SISO) equations:

$$\begin{aligned} CPU(k+1) &= a_{CPU} CPU(k) + b_{CPU} KA(k) \\ MEM(k+1) &= a_{MEM} MEM(k) + b_{MEM} MC(k) \end{aligned}$$

where time is discretized ($X(k)$ denotes the value of X at the k-th time step), and the as and bs denotes constant coefficients, to be determined by identification (least squares regression with discrete sine wave inputs).

A more general model, which does not assume a priori correlations, is the Multiple Input, Multiple Output (MIMO) model:

$$\mathbf{y}(k+1) = \mathbf{A}\mathbf{y}(k) + \mathbf{B}\mathbf{u}(k)$$

in which

$$\mathbf{y}(k) = \begin{bmatrix} CPU(k) \\ MEM(k) \end{bmatrix}, \mathbf{u}(k) = \begin{bmatrix} KA(k) \\ MC(k) \end{bmatrix},$$

and $\mathbf{A}$ and $\mathbf{B}$ are 2×2 constant matrices, again determined by identification.

The first step is to test the accuracy of the models. Experience with a synthetic workload generator, comparing actual values with those predicted by the model, gives the following results.

- While the SISO model of MEM is accurate, the SISO model of CPU provides a poor fit to actual data, specially with multiple step inputs.
- Overall, the MIMO model gives a more accurate fit than the dual SISO model.
- The accuracy of the models degrades near the ends of the operating region, showing the limits of the linearity assumption.

[7] A system with input $\mathbf{u}$ and output $\mathbf{y}$ is *linear* if $\mathbf{y}(a\mathbf{u}) = a\mathbf{y}(\mathbf{u})$ and $\mathbf{y}(\mathbf{u}_1 + \mathbf{u}_2) = \mathbf{y}(\mathbf{u}_1) + \mathbf{y}(\mathbf{u}_2)$. It is *time-invariant* if its behavior is insensitive to a change of the origin of time.

The second step is to use the model for feedback control. The controller is Proportional Integral (PI), with gain coefficients determined by classical controller design methods (pole placement for SISO, and cost function minimization for MIMO). Experience shows that the SISO model performs well in spite of its inaccuracy, due to the limited degree of coupling between KA and MC. However, the MIMO model does better overall, specially for heavy workload.

The contributions of this work are to show that control theory can indeed be applied to Internet services, and that good results can be obtained with a simple design. In that sense, this work is a proof of concept experiment. However, being one of the first attempts at modeling an Internet service, it suffers from some limitations.

- The model is linear, while experience shows that actual systems exhibit a non-linear behavior due to saturation and thrashing.
- The model uses low-level system parameters as controlled variables, instead of user-perceived QoS factors. In that sense, it does not solve the SLA decomposition problem, as presented in 12.3.1.
- The model considers a single server, while clusters are more common.
- The workload generator does not create wide amplitude load peaks.

More recent models, such as that presented in the next section, aim at overcoming these limitations.

Another early attempt at applying feedback control to web server performance is [Abdelzaher et al. 2002]. They use a SISO algorithm, in which the measured variable is system utilization and the controlled variable is a single index that determines the fraction of clients to be served at each service level (assuming the service to be available at various levels of degradation). This index is the input of an admission control actuator.

12.5.5 Resource Provisioning for a Multi-tier Service

The work reported in [Urgaonkar et al. 2007] differs from that presented in the previous sections, in that it is based on an analytical model of the managed system. This model may be used to predict response times, and to drive resource allocation, using both dynamic resource provisioning and admission control.

The class of systems under study is that of multi-tiered systems (12.1.3). A multi-tiered system is represented as a network of queues: queue Q_i represents the i-th tier ($i = 1, 2, \ldots, M$). In the first, basic, version of the model, there is a single server per tier. A request goes from tier 1 to tier M, receiving service at each tier, as shown on Figure 12.8, which represents a 3-tier service.

The model represents a session as a set of requests (cf 12.5.3). Sessions are represented by a special queueing system, Q_0, which consists of N "servers", which act as request generators. Each server in Q_0 represents a session: it emits successive requests, separated by a think time. When a session terminates, a new one is started at the same server; thus Q_0 represents a steady load of N sessions.

The model takes the following aspects into account:

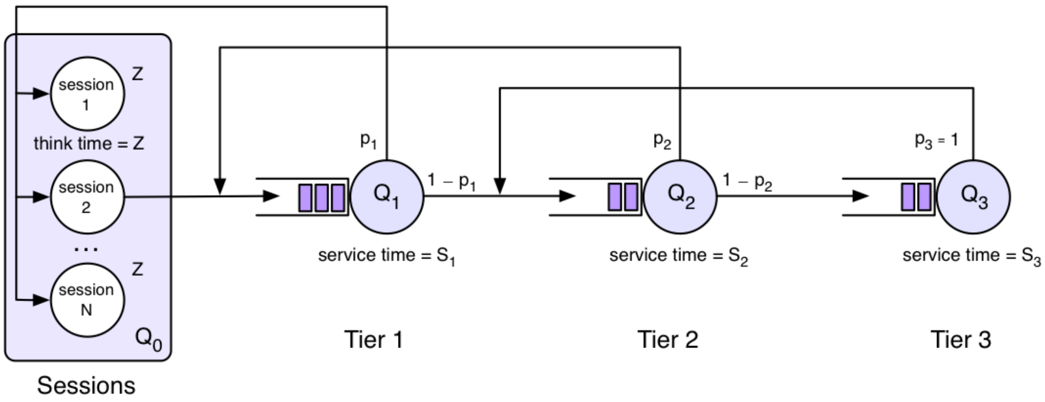

Figure 12.8. Queueing network for a 3-tier service (adapted from [Urgaonkar et al. 2007])

- A request may visit a tier (say number k) several times (e.g., if it makes several queries in a database).
- A request may go no further than a certain tier (e.g., because the answer was found in a cache in that tier, and no further processing is needed).

This is represented in the model by a transition from tier k to tier $k-1$. The two above cases are represented using different transition probabilities. A request terminates when it goes back from Q_1 to Q_0.

The parameters of the basic version of the model are the service times S_i at each tier i, the transition probabilities between queues (p_i is the transition probability from Q_i to Q_{i-1}), and the think time of the user, represented by the service time of the servers in Q_0. Numerical values of these parameters, for a specific application, are estimated by monitoring the execution of this application under a given workload. Once these parameters are known, the system may be resolved, using the Mean-Value Analysis (MVA) algorithm [Reiser and Lavenberg 1980] for closed queueing networks, to compute the mean response time.

Several enhancement have been added to the basic version.

- Modeling multiple servers. Internet services running on clusters use replicated tiers for efficiency and availability. This is represented by multiple queues at each tier: the single queue Q_i is replaced by r_i queues $Q_{i,1}, \ldots, Q_{i,r_i}$, where r_i is the degree of replication at tier i. As described in 12.3.2, a load balancer forwards requests from one tier to the next, and dispatches them across replicas. A parameter of the model captures the possible imbalance due to specific constraints such as stateful servers or cache affinity.
- Handling concurrency limits. There is a limit to the number of activities that a node may concurrently handle. This is represented by a drop probability (some requests are dropped due to concurrency limits).
- Handling multiple session classes. Given the parameters of each class (think time, service times, etc.), the model allows the average response time to be computed on

a per-class basis.

The model has some limitations: a) Each tier is modeled by a single queue, while requests actually compete for several different resources: CPU, memory, network, disk. b) The model assumes that a request uses the resources of one tier at a time. While this is true for many Internet services, some applications such as streaming servers use pipeline processing, not currently represented by the model.

Experiments with synthetic loads have shown that the model predicts the response time of applications within the 95% confidence interval. In addition, the model has been used to assist two aspects of resource management: dynamic resource provisioning (for predictable loads) and admission control (for unexpected peaks).

1. *Dynamic resource provisioning*. Assume that the workload of the application may be predicted for the short term, in terms of number of sessions and characteristics of the sessions. Then resource provisioning works as follows: define an initial server assignment for each tier (e.g., one server per tier). Use the model to determine the average response time. If it is worse than the target, examine the effect of adding one more server to each tier that may be replicated. Repeat until the response time is below the target. This assumes that the installation has sufficient capacity (total number of servers to meet the target). The complexity of the computation is of the order of $M.N$ (M number of tiers, N number of sessions), which is acceptable if resources are provisioned for a fairly long period (hours).

2. *Admission control*. Admission control is intended to react to unexpected workload peaks. In the model under study, it is performed by a front-end node, which turns out excess sessions to guarantee the SLA (here the response time). The criterion for dropping requests is to maximize a utility function, which is application dependent. For example, the utility function may be a global revenue generated by the admitted sessions; assigning a revenue to each session class allows service differentiation. In the experiments described, the system admits the sessions in non-increasing order of generated revenue, determining the expected response time according to the model, and drops a session if its admission would violate the SLA for at least one session class.

The interest of this work is that it provides a complete analytical model for a complex system: a multi-tiered application with replicated tiers, in the presence of a complex load. The system is able to represent such features as load unbalancing and caching effects, and allows for differentiated service. Its main application is capacity planning and dynamic resource provisioning for loads whose characteristics may be predicted with fair accuracy.

Similar queueing models have been used for correlating SLOs with system-level thresholds [Chen et al. 2007], and for identifying the bottleneck tier in a multi-tier system, in order to apply a control algorithm for service differentiation [Diao et al. 2006].

12.5.6 Real-Time Scheduling

[Lu et al. 2002] have explored the use of feedback control theory for the systematic design of algorithms for real-time scheduling. This is a single-resource problem: a number of tasks, both periodic and aperiodic, compete for the CPU. Each task must meet a deadline; for periodic tasks, the deadline is set for each period, and for non-periodic tasks, the deadline is relative to the arrival time. The arrival times (for aperiodic tasks) and CPU utilization rates are unknown, but estimates are provided for each task.

The controlled variables, i.e., the performance metrics controlled by the scheduler, are sampled at regular intervals (with a period W), and defined over the kth sampling window $[(k-1)W, kW]$. These variables include:

- The deadline miss ratio $M(k)$, the number of deadline misses divided by the number of completed and aborted tasks in the window.
- The CPU utilization $U(k)$, the percentage of CPU busy time in the window.

Reference values M_S and U_S are set for these variables (e.g. $M_S = 0$ and $U_S = 90\%$). Three variants of the control system have been studied: FC-U, which only controls $M(k)$, FC-M, which only controls $U(k)$, and FC-UM, which combines FC-U and FC-M as described later.

There is a single manipulated variable, on which the scheduler may act to influence the system's behavior: $B(k)$, the total estimated CPU utilization of all tasks in the system. The actuator changes the value of $B(k)$ according to the control algorithm. Then an optimization algorithm determines the precise allocation of CPU to tasks over the $(k+1)$th window so that their total CPU utilization is $B(k+1)$. In addition to being called at each sampling instant, the actuator is also invoked upon the arrival of each task, to correct prediction errors on arrival times..

The optimization algorithm relies on the notions of QoS level and value. Each task T_i is assigned a certain number (at least two) of QoS levels, which determine possible conditions of execution, defined by a few attributes associated with each level j, among which a relative deadline $D_i[j]$, an estimated CPU utilization $B_i[j]$ and a value $V_i[j]$. Level 0 corresponds to the rejection of the task; both $B_i[0]$ and $V_i[0]$ are set[8] to 0. The value $V_i[j]$ expresses the contribution of the task (as estimated by the application designer) if it meets its deadline $D_i[j]$ at level j; if it misses the deadline, the contribution is $V_i[0]$ (equal or less than 0). Both $B_i[j]$ and $V_i[j]$ increase with j.

The optimization algorithm assigns a (non-zero) QoS level to each task to maximize its value density (the ratio $V_i[j]/B_i[j]$, defined as 0 for level 0), and then schedules the tasks in the order of decreasing density until the total prescribed CPU utilization $B(k)$ is reached.

Intuitively, a system with two levels of QoS corresponds to an admission control algorithm: a task is either rejected (level 0) or scheduled at its single QoS level (level 1); the value density criterion gives priority to short, highly valued tasks. Increasing the number of levels allows for a finer control over the relative estimated value of a task's completion: instead of deciding to either run or reject a task, the algorithm may chose to run it at an intermediate level of QoS for which the ratio between contribution and consumed resources qualifies the task for execution.

The controller algorithm uses a simple proportional control function (Figure 12.9), i.e., the control function is $D(k) = K.E(k)$, where $E(k)$ (the error), is the difference between

[8] $V_i[0]$ may also be set to a negative value, which corresponds to a penalty.

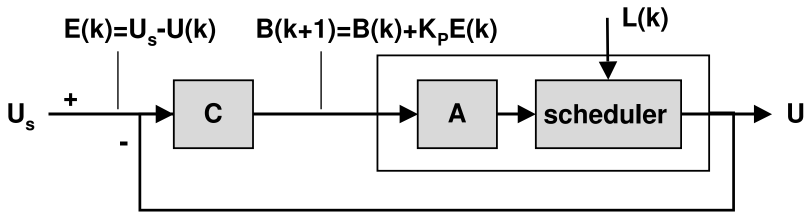

Figure 12.9. Feedback control of CPU scheduling

the reference value and the measured output, i.e., either $M_S - M(k)$ or $U_S - U(k)$ depending on the controlled variable. Then the command to the actuator is $B(k+1) = B(k) + D(k)$.

Combining FC-U and FC-M is done as follows: the two separate control loops are set up, then the two command signals $D(k)$ are compared, and the lower signal is selected.

The disturbance is the actual load, which introduces an uncertainty factor since the estimates of arrival times and CPU consumption may be incorrect.

A complete analysis of this system has been done, using the control theory methodology. The system is non-linear, due to saturation. This can be modeled by two different regimes separated by a threshold; each of the regimes can be linearized. Then the analysis determines stability conditions and allows the parameters to be tuned for various load characteristics.

The experimental results obtained with various synthetic loads show that the system is stable in the face of overload, that it has low overshoot in transient states, and that it achieves an overall better utilization of CPU than standard open-loop control algorithms such as EDF. In addition, the system has all the benefits of the availability of a model, i.e., performance prediction and help in parameter selection.

12.6 Conclusion

We summarize a few conclusions on resource management to achieve QoS requirements.

- Resource management has long been guided by heuristics. The situation is changing, due to a better understanding of SLA decomposition (the relationship between user-perceived QoS and internal resource occupation parameters). This in turn results from the elaboration of more accurate models of the managed systems, and from the development of better statistical correlation methods.

- The application of control theory to resource management is still in an early stage. This may be explained by the inherent complexity of the systems under study and by their highly non-linear and time-dependent character. Control-based techniques should also benefit from the above-mentioned progress in SLA decomposition. Most likely, successful methods will combine predictive (model-based) and reactive (feedback-controlled) algorithms.

- Admission control appears to be the dominant strategy for dealing with unexpected load peaks. It may be made minimally invasive, using a black box approach, and its actuators are easy to implement.

However, this apparent simplicity should not offset the difficulty of the design of a good admission control algorithm. As many experiments have shown, a good knowledge of the load can be usefully exploited, again mixing predictive and reactive approaches.

- A number of aspects need further investigation, e.g., allocation problems for multiple resources, interaction between resource management for multistage requests (such as multiple tiers), further elaboration of the contract satisfaction criteria (probabilistic, guaranteed).

12.7 Historical Note

Resource allocation was one the main areas of operating systems research in the mid-1960s. The pitfalls of resource management, such as deadlock and starvation, were identified at that time, and solutions were proposed. The influence of resource allocation on user-perceived performance was also investigated, using both queueing models and empirical approaches. Capacity planning models were used for dimensioning mainframes. The problems of resource overcommitment, leading to thrashing, were understood in the late 1960s, and the principle of their solution (threshold-based control of the multiprogramming degree) is still valid today.

The notion of Quality of Service was not present as such for operating systems (although similar concepts were underlying the area of performance management). QoS was introduced in networking, first in the form of support for differentiated service, then for application-related performance. QoS emerged as an important area of study with the advent of the first multimedia applications (e.g., streaming audio and video), for which the "best effort" philosophy of the Internet Protocol was inappropriate. Techniques such as channel reservation, traffic policing, scheduling algorithms, and congestion avoidance were developed. [Wang 2001] presents the main concepts and techniques of Internet QoS.

In the 1980s, heuristic-based feedback control methods were successfully applied to network operation (e.g., for adaptive routing in the Internet, for flow control in TCP). The extension of QoS to middleware systems was initiated in the 1990s (see e.g., [Blair and Stefani 1997]).

The rapid rise of Internet services in the 2000s, and the growing user demand for QoS guarantees, have stimulated efforts in this area. As a part of the autonomic computing movement (see 10.2), several projects have attempted to apply control theory to the management of computing systems, specially for performance guarantees. [Hellerstein et al. 2004] gives a detailed account of these early efforts. In addition to feedback control, current research topics are the development of accurate models for complex Internet services, and the use of statistical methods to characterize both the service request load and the behavior of the managed systems.

References

[Abdelzaher et al. 2002] Abdelzaher, T. F., Shin, K. G., and Bhatti, N. (2002). Performance guarantees for Web server end-systems: A control-theoretical approach. *IEEE Transactions on Parallel and Distributed Systems*, 13(1):80–96.

[Amza et al. 2002] Amza, C., Cecchet, E., Chanda, A., Cox, A., Elnikety, S., Gil, R., Marguerite, J., Rajamani, K., and Zwaenepoel, W. (2002). Specification and Implementation of Dynamic Web Site Benchmarks. In *WWC-5: IEEE 5th Annual Workshop on Workload Characterization*, Austin, TX, USA.

[Appleby et al. 2001] Appleby, K., Fakhouri, S., Fong, L., Goldszmidt, G., Kalantar, M., Krishnakumar, S., Pazel, D., Pershing, J., and Rochwerger, B. (2001). Oceano - SLA based management of a computing utility. In *Proceedings of the 7th IFIP/IEEE International Symposium on Integrated Network Management*.

[Aron et al. 2000] Aron, M., Druschel, P., and Zwaenepoel, W. (2000). Cluster reserves: a mechanism for resource management in cluster-based network servers. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 90–101, Santa Clara, California.

[Balazinska et al. 2004] Balazinska, M., Balakrishnan, H., and Stonebraker, M. (2004). Contract-Based Load Management in Federated Distributed Systems. In *First USENIX Symposium on Networked Systems Design and Implementation (NSDI'04)*, pages 197–210, San Francisco, CA, USA.

[Banga et al. 1999] Banga, G., Druschel, P., and Mogul, J. C. (1999). Resource containers: A new facility for resource management in server systems. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation (OSDI'99)*, New Orleans, Louisiana.

[Bavier et al. 2004] Bavier, A., Bowman, M., Chun, B., Culler, D., Karlin, S., Muir, S., Peterson, L., Roscoe, T., Spalink, T., and Wawrzoniak, M. (2004). Operating System Support for Planetary-Scale Network Services. In *First USENIX Symposium on Networked Systems Design and Implementation (NSDI'04)*, pages 253–266, San Francisco, CA, USA.

[Blair and Stefani 1997] Blair, G. and Stefani, J.-B. (1997). *Open Distributed Processing and Multimedia*. Addison-Wesley. 452 pp.

[Blanquer et al. 2005] Blanquer, J. M., Batchelli, A., Schauser, K., and Wolski, R. (2005). Quorum: Flexible Quality of Service for Internet Services. In *Second USENIX Symposium on Networked Systems Design and Implementation (NSDI'05)*, pages 159–174, Boston, Mass., USA.

[Brawn and Gustavson 1968] Brawn, B. and Gustavson, F. (1968). Program behavior in a paging environment. In *Proceedings of the AFIPS Fall Joint Computer Conference (FJCC'68)*, pages 1019–1032.

[Cardellini et al. 2002] Cardellini, V., Casalicchio, E., Colajanni, M., and Yu, P. S. (2002). The state of the art in locally distributed Web-server systems. *ACM Computing Surveys*, 34(2):263–311.

[Chase et al. 2003] Chase, J. S., Irwin, D. E., Grit, L. E., Moore, J. D., and Sprenkle, S. E. (2003). Dynamic virtual clusters in a grid site manager. In *Proceedings 12th IEEE International Symposium on High Performance Distributed Computing (HPDC'03)*, Seattle, Washington.

[Chen et al. 2007] Chen, Y., Iyer, S., Liu, X., Milojicic, D., and Sahai, A. (2007). SLA Decomposition: Translating Service Level Objectives to System Level Thresholds. Technical Report HPL-2007-17, Hewlett-Packard Laboratories, Palo Alto.

[Cherkasova and Phaal 2002] Cherkasova, L. and Phaal, P. (2002). Session-Based Admission Control: A Mechanism for Peak Load Management of Commercial Web Sites. *IEEE Transactions on Computers*, 51(6):669–685.

[Cohen et al. 2004] Cohen, I., , Goldszmidt, M., Kelly, T., Symons, J., and Chase, J. S. (2004). Correlating instrumentation data to system states: A building block for automated diagnosis and control. In *Proceedings of the Sixth Symposium on Operating Systems Design and Implementation (OSDI'04)*, pages 31–44, San Francisco, CA, USA.

[Diao et al. 2002a] Diao, Y., Gandhi, N., Hellerstein, J. L., Parekh, S., and Tilbury, D. M. (2002a). MIMO Control of an Apache Web Server: Modeling and Controller Design. In *Proceeedings of the American Control Conference*, volume 6, pages 4922–4927.

[Diao et al. 2002b] Diao, Y., Gandhi, N., Hellerstein, J. L., Parekh, S., and Tilbury, D. M. (2002b). Using MIMO feedback control to enforce policies for interrelated metrics with application to the Apache Web server. In *IEEE/IFIP Network Operations and Management Symposium (NOMS'02)*, pages 219–234, Florence, Italy.

[Diao et al. 2005] Diao, Y., Hellerstein, J. L., Parekh, S., Griffith, R., Kaiser, G., and Phung, D. (2005). Self-Managing Systems: A Control Theory Foundation. In *12th IEEE International Conference and Workshops on the Engineering of Computer-Based Systems (ECBS'05)*, pages 441–448, Greenbelt, MD, USA.

[Diao et al. 2006] Diao, Y., Hellerstein, J. L., Parekh, S., Shaikh, H., and Surendra, M. (2006). Controlling Quality of Service in Multi-Tier Web Applications. In *26st International Conference on Distributed Computing Systems (ICDCS'06)*, pages 25–32, Lisboa, Portugal.

[Doyle et al. 2003] Doyle, R. P., Chase, J. S., Asad, O. M., Jin, W., and Vahdat, A. M. (2003). Model-based Resource Provisioning in Web Service Utility. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems (USITS-03)*, Seattle, WA, USA.

[Elnikety et al. 2004] Elnikety, S., Nahum, E., Tracey, J., and Zwaenepoel, W. (2004). A method for transparent admission control and request scheduling in e-commerce web sites. In *Proceedings of the 13th International Conference on World Wide Web (WWW'2004)*, pages 276–286, New York, NY, USA.

[Feldman et al. 2005] Feldman, M., Lai, K., and Zhang, L. (2005). A price-anticipating resource allocation mechanism for distributed shared clusters. In *Proceedings of the 6th ACM Conference on Electronic Commerce (EC'05)*, pages 127–136, New York, NY, USA. ACM Press.

[Friedman et al. 1997] Friedman, N., Geiger, D., and Goldszmidt, M. (1997). Bayesian network classifiers. *Machine Learning*, 29(13):131–163.

[Fu et al. 2003] Fu, Y., Chase, J., Chun, B., Schwab, S., and Vahdat, A. (2003). SHARP: an architecture for secure resource peering. In *Proceedings of the nineteenth ACM Symposium on Operating Systems Principles (SOSP'03)*, pages 133–148. ACM Press.

[Govil et al. 2000] Govil, K., Teodosiu, D., Huang, Y., and Rosenblum, M. (2000). Cellular Disco: Resource Management using Virtual Clusters on Shared-memory Multiprocessors. *ACM Transactions on Computer Systems*, 18(3):229–262.

[Hellerstein 2004] Hellerstein, J. L. (2004). Challenges in Control Engineering of Computer Systems. In *Proceedings of the American Control Conference*, pages 1970–1979, Boston, Mass, USA.

[Hellerstein et al. 2004] Hellerstein, J. L., Diao, Y., Tilbury, D. M., and Parekh, S. (2004). *Feedback Control of Computing Systems*. John Wiley and Sons. 429 pp.

[Irwin et al. 2005] Irwin, D., Chase, J., Grit, L., and Yumerefendi, A. (2005). Self-recharging virtual currency. In *P2PECON '05: Proceedings of the 2005 ACM SIGCOMM Workshop on Economics of Peer-to-Peer Systems*, pages 93–98, New York, NY, USA. ACM Press.

[Kamra et al. 2004] Kamra, A., Misra, V., and Nahum, E. (2004). Yaksha: A Self-Tuning Controller for Managing the Performance of 3-Tiered Web Sites. In *International Workshop on Quality of Service (IWQoS)*.

[Kohler et al. 2000] Kohler, E., Morris, R., Chen, B., Jannotti, J., and Kaashoek, M. F. (2000). The Click modular router. *ACM Transactions on Computer Systems*, 18(3):263–297.

[Lazowska et al. 1984] Lazowska, E. D., Zahorjan, J., Graham, G. S., and Sevcik, K. C. (1984). *Quantitative System Performance : Computer system analysis using queueing network models*. Prentice Hall. 417 pp., available on-line at http://www.cs.washington.edu/homes/lazowska/qsp/.

[Lu et al. 2002] Lu, C., Stankovic, J. A., Tao, G., and Son, S. H. (2002). Feedback Control Real-Time Scheduling: Framework, Modeling and Algorithms. *Real Time Systems Journal*, 23(1/2):85–126.

[Menascé and Almeida 2001] Menascé, D. A. and Almeida, V. A. F. (2001). *Capacity Planning for Web Services: metrics, models, and methods*. Prentice Hall. 608 pp.

[Moore et al. 2002] Moore, J., Irwin, D., Grit, L., Sprenkle, S., , and Chase, J. (2002). Managing mixed-use clusters with cluster-on-demand. Technical report, Department of Computer Science, Duke University. http://issg.cs.duke.edu/cod-arch.pdf.

[Pai et al. 1998] Pai, V. S., Aron, M., Banga, G., Svendsen, M., Druschel, P., Zwaenepoel, W., and Nahum, E. M. (1998). Locality-aware request distribution in cluster-based network servers. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*, pages 205–216.

[Reiser and Lavenberg 1980] Reiser, M. and Lavenberg, S. S. (1980). Mean-value analysis of closed multichain queuing networks. *Journal of the Association for Computing Machinery*, 27(2):313–322.

[Schroeder et al. 2006] Schroeder, B., Wierman, A., and Harchol-Balter, M. (2006). Open versus closed: a cautionary tale. In *Third Symposium on Networked Systems Design & Implementation (NSDI'06)*, pages 239–252. USENIX Association.

[Sullivan et al. 1999] Sullivan, D. G., Haas, R., and Seltzer, M. I. (1999). Tickets and Currencies Revisited: Extensions to Multi-Resource Lottery Scheduling. In *Workshop on Hot Topics in Operating Systems (HotOS VII)*, pages 148–152.

[Sutherland 1968] Sutherland, I. E. (1968). A futures market in computer time. *Commun. ACM*, 11(6):449–451.

[TPC 2006] TPC (2006). TPC Benchmarks. Transaction Processing Performance Council. http://www.tpc.org/.

[Urgaonkar et al. 2007] Urgaonkar, B., Pacifici, G., Shenoy, P., Spreitzer, M., and Tantawi, A. (2007). An Analytical Model for Multi-tier Internet Services and its Applications. *ACM Transactions on the Web*, 1(1).

[Urgaonkar and Shenoy 2004] Urgaonkar, B. and Shenoy, P. (2004). Sharc: Managing CPU and Network Bandwidth in Shared Clusters. *IEEE Transactions on Parallel and Distributed Systems*, 15(1):2–17.

[Urgaonkar et al. 2002] Urgaonkar, B., Shenoy, P., and Roscoe, T. (2002). Resource Overbooking and Application Profiling in Shared Hosting Platforms. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation (OSDI'02)*, pages 239–254, Boston, MA.

[Uttamchandani et al. 2005] Uttamchandani, S., Yin, L., Alvarez, G., Palmer, J., and Agha, G. (2005). Chameleon: a self-evolving, fully-adaptive resource arbitrator for storage systems. In *Proceedings of the 2005 USENIX Technical Conference*, pages 75–88, Anaheim, CA, USA.

[Waldspurger and Weihl 1994] Waldspurger, C. A. and Weihl, W. E. (1994). Lottery scheduling: Flexible proportional-share resource management. In *Proceedings of the First Symposium on Operating Systems Design and Implementation (OSDI'94)*, pages 1–11, Monterey, California.

[Wang 2001] Wang, Z. (2001). *Internet QoS*. Morgan Kaufmann. 240 pp.

[Welsh and Culler 2003] Welsh, M. and Culler, D. (2003). Adaptive Overload Control for Busy Internet Servers. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems (USITS-03)*, Seattle, WA, USA.

[Welsh et al. 2001] Welsh, M., Culler, D., and Brewer, E. (2001). SEDA: An Architecture for Well-Conditioned Scalable Internet Services. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles (SOSP'01)*, pages 230–243, Banff, Alberta, Canada.

[Zhang et al. 2005] Zhang, Q., Sun, W., Riska, A., Smirni, E., and Ciardo, G. (2005). Workload-Aware Load Balancing for Clustered Web Servers. *IEEE Transactions on Parallel and Distributed Systems*, 16(3):219–233.