Enforcing Resource Sharing Agreements among Distributed Server Clusters

(NYU-CS-Technical Report No. TR2001-818)

Tao Zhao and Vijay Karamcheti
Department of Computer Science
Courant Institute of Mathematical Sciences
New York University
{*taozhao,vijayk*}*@cs.nyu.edu*

Abstract

Future scalable, high throughput, and high performance applications are likely to execute on platforms constructed by clustering multiple autonomous distributed servers, with resource access governed by agreements between the owners and users of these servers. As an example, application service providers (ASPs) can pool their resources together according to pre-specified sharing agreements to provide better services to their customers. Such systems raise several new resource management challenges, chief amongst which is the enforcement of agreements to ensure that, despite the distributed nature of both requests and resources, user requests only receive a predetermined share of the aggregate resource and that the resources of a participant are not misused. Current solutions only enforce such agreements at a coarse granularity and in a centralized fashion, limiting their applicability for general workloads.

This paper presents an architecture for the distributed enforcement of resource sharing agreements. Our approach exploits a uniform application-independent representation of agreements, and combines it with efficient time-window based coordinated queuing algorithms running on multiple nodes. We have successfully implemented this general strategy in two different network layers: a layer-7 HTTP redirector and a layer-4 packet redirector, which redirect connection requests from distributed clients to a cluster of distributed servers. Our measurements of both implementations verify that our approach is general and effective: different client groups receive service commensurate with their agreements.

1 Introduction

Although centralized parallel and clustered servers currently dominate as the platform of choice for running high throughput and high performance applications, there is a growing trend towards executing such applications on platforms constructed by clustering multiple, autonomous distributed servers. Early evidence of this trend can be found in grid systems such as Globus [22] and Legion [24], service provider "computing utilities" [1], and internet-scale networks for content distribution [2], peer-to-peer computing [3, 4], and large-scale clustering [5, 6]. Such distributed server clusters offer scalable resources at a low cost and provide better locality support for distributed clients. However, these advantages are accompanied by new resource management challenges.

A key challenge arises from the fact that such systems may span multiple organizational domains, with access governed by *resource sharing agreements* (also called Service Level Agreements, SLAs) between owners and users of these servers. Even when resources belong to the same domain, agreements may permit more flexible control over resource usage, considering factors such as locality, pricing etc. To consider an example, many organizations use computing and networking resources supplied by service providers (ISPs or ASPs) to reduce maintenance and ownership costs. In this context, SLAs specify, usually in measurable terms, the type and level of services to be provided to a customer's clients. Note that service providers can also have agreements amongst each other to form a bigger resource pool.

This paper focuses on the problem of enforcing such agreements, ensuring that, despite the distributed nature of both requests and resources, user requests only receive a predetermined share of the aggregate resource and that participant resources are not used beyond specified thresholds. Although other researchers have proposed techniques for SLA enforcement in centralized servers [1, 13], such end-point solutions assume either that all client requests are being aggregated into a centralized decision-making location or that extensive feedback about server loads is available. Consequently, these solutions do not scale when both clients and resources are widely distributed.

To understand the problems involved in distributed enforcement, consider the example in Figure 1, where an application service provider S with distributed resources S_1 and S_2, each capable of servicing 50 requests per second, negotiates SLAs with two organizations A and B for 20% and 80% of its resources respectively. Requests from clients of A and B are forwarded to the servers using two redirectors R_1 and R_2. For locality reasons, these redirectors bias their distribution of requests sent to each server: R_1 forwards 75% of its requests to S_1 and 25% to S_2, while R_2 does the reverse.

Now consider what happens in a situation when the load (in requests/sec) at the two redirectors is $(A: 20, B: 20)$ and $(A: 20, B: 60)$ respectively. Because of the locality bias in the redirectors, each server sees the following load: $(A: 20, B: 30)$ for S_1 and $(A: 20, B: 50)$ for S_2. Since S_1 and S_2 are assumed to enforce the SLAs independently, S_1 services $(A: 20, B: 30)$ requests per second, and S_2 services $(A: 10, B: 40)$ resulting in an aggregate processing rate of $(A: 30, B: 70)$. Note that this violates the SLA guaranteeing 80% of S's resources to B.

The above example shows that agreement enforcement applied only at end-points, independent from activities elsewhere in the system, cannot handle agreements involving distributed requests and resources. To solve this problem, this paper describes an architecture for coordinated enforcement of resource sharing agreements at multiple admission points. Our approach, which borrows ideas from *lottery scheduling* [30] and was introduced in its preliminary form in [35], employs an application-independent way of representing agreements using "tickets" (representing transfer of rights) issued by "currencies"; the latter denominate tickets and have dynamically fluctuating value determined by physical system resources. In their general form, agreements consist of a lower bound (guaranteed

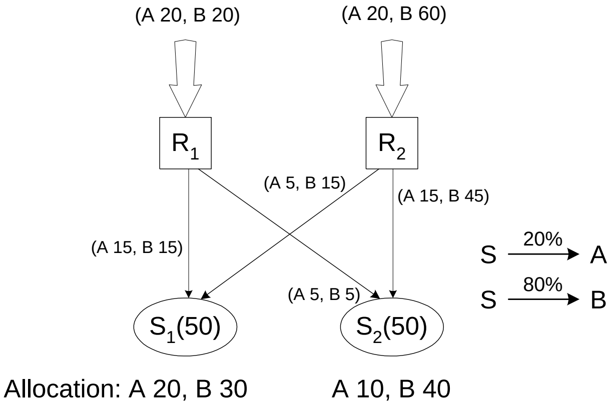


Figure 1: End-point agreement enforcement cannot handle distributed incoming requests.

level of service) and an upper bound (best-effort level of service). Taking advantage of this uniform expression, we shift the responsibility of enforcing agreements from applications at the server side to the network fabric, where we realize a distributed queuing strategy that enforces these agreements and optimizes a global metric (such as system-wide average response time) despite client requests from multiple network entry points. Our scheme additionally accounts for hierarchical and transitive agreement structures using a linear-programming formulation.

We have successfully implemented this strategy in two different network layers: an application layer HTTP redirector (L7 switch) and a transport layer packet redirector (L4 switch), which forward client requests to network-attached servers. Based upon knowledge of the aggregate sharing agreements and incoming request load conditions, each redirector performs appropriate queuing of client requests (coordinated with other redirectors) so as to enforce these agreements. The application layer redirectors send back HTTP redirection headers to route a client's request to desired servers, while the transport layer redirectors use network address translation (NAT) to forward packets to cluster servers. Measurements using a synthetic web request generator program, WebBench [7], demonstrate that both implementations can enforce resource sharing agreements effectively, while gracefully adapting to dynamically changing request loads.

The rest of the paper is organized as follows. In Section 2 we discuss different models of resource sharing agreements and their uniform expression using the ticket/currency-based scheme. Section 3 describes the distributed agreement enforcement strategy and Section 4 presents its two prototype implementations. Section 5 evaluates the performance of these prototypes. Related work is covered in Section 6 and Section 7 summarizes the paper.

2 Expressing Resource Sharing Agreements

We are interested in resource sharing agreements in two contexts: a *community* context and a *service provider* context. In a community, sharing agreements are between cooperative participants who contribute resources for controlled use by other members (e.g., a peer-to-peer file sharing application). In the service provider context, agreements are between a provider that owns resources and customers whose clients access these resources (e.g., an ASP that hosts web-based applications for its clients). Agreement enforcement must typically optimize different criteria in the two cases:

a community-wide metric (e.g., average response time) for the former, and service provider revenue for the latter.

2.1 Different Models of Agreements

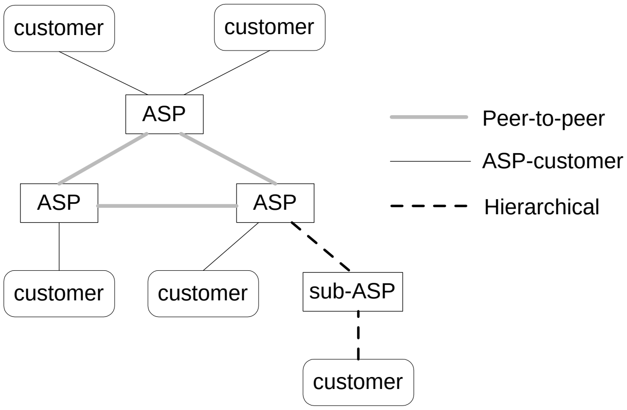


Figure 2: Different models of resource sharing agreements.

Figure 2 shows different models of resource sharing agreements in both community and service provider contexts. Three application service providers (ASPs) in the center form a resource sharing community. Among them there are *peer-to-peer* sharing agreements so that they can access larger resource pools to provider better performance to their customers' clients while amortizing their costs. Each ASP has *ASP-customer* agreements (SLAs) with multiple customers to specify the level of services to be provided to customers' clients. When a sub-ASP resells ASP services to its own customers, *hierarchical* agreement structures emerge. In this paper we mainly focus on the former two agreement models, although our techniques can be naturally extended to the latter.

2.2 Agreement Structure

Agreements are contracts between principals owning resources and those wishing to use them. In this paper, we focus on *rate resources* such as CPU share, network bandwidth, and server transaction rate. Agreements refer to the access a principal j has on principal i's resources over a time window and are modeled as a tuple: $[\mathrm{lb}_{\mathrm{ij}}, \mathrm{ub}_{\mathrm{ij}}]$ representing the lower bound (guaranteed reservation during overload) and an upper bound. The lower bound is different from other reservation systems in which the reserved resources are put aside waiting for requests. In our model, resources reserved for principal j can be used by others if j does not use it. This ensures better resource utilization. In addition, agreements are interpreted dynamically: changes in a principal's resource levels affect the amount available to others via agreements. Extending ownership to include resources that a principal obtains via its own agreements enables transitive flow of resources.

2.3 Agreement Representation using Tickets and Currencies

Enforcement of agreements is complicated in the presence of different resource types and transitive agreement chains. To permit simpler enforcement, we express agreements in a uniform and abstract fashion using the notions of *tickets* (representing transfer of rights) backed by *currencies* (whose values are determined by physical resources).[1]

An agreement between two principals A and B is represented by a flow of tickets from A to B, denominated by A's currency. These tickets contribute value to B's currency, and "fund" requests made by B's clients; the latter are only permitted if there are funds. Informally, agreements that permit a principal to use others' resources have the effect of increasing its currency value, while agreements that let others use this principal's resources have the opposite effect. The reason for choosing this expression model is (1) to decouple agreement structures from the physical resources available at any time to a principal; and (2) to permit uniform treatment of both direct and indirect agreements. Indirect agreements do not need to be explicitly represented; instead, their effect is naturally factored in because of the flow of resource values (tickets) through multiple currencies.

To represent the general $[\mathrm{lb}, \mathrm{ub}]$ form of agreements, there are two types of tickets: *mandatory* and *optional*, and two corresponding values for each currency. A mandatory ticket corresponds to the lower bound ($\mathrm{lb}_{\mathrm{ij}}$) of an agreement, and an optional ticket represents the difference between the upper and lower bound ($\mathrm{ub}_{\mathrm{ij}} - \mathrm{lb}_{\mathrm{ij}}$).

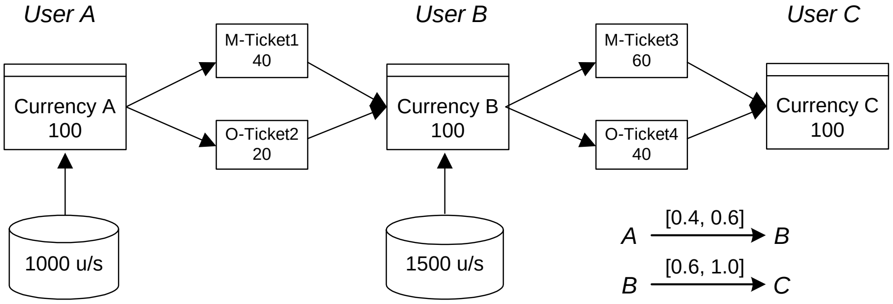


Figure 3: Use of tickets and currencies to express sharing agreements.

Figure 3 shows an example system with three principals to clarify this model: A and B own resources with capacities of 1000 and 1500 units/second respectively, which they share with each other and with C. The physical resources fund currencies A and B. A's agreement [0.4, 0.6] with B, allowing the latter to access between 40% and 60% of its resources is captured by a mandatory (M-Ticket1) and an optional ticket (O-Ticket2), carrying face values 40 and 20 respectively (these values are normalized with respect to the face value of A's currency, 100). B's agreement [0.6, 1.0] with C is represented similarly. A ticket's real value is computed in terms of the real value of its issuing currency. For example, M-Ticket3's real value is 60% of the real value of B's currency, which in turn is $1500 + 1000 \times (40/100) = 1900$, i.e., 1140. O-Ticket4's real value is $1900 \times 40/100 + 200 \times ((60+40)/100) = 960$, with contributions both from the mandatory value of B's currency (as per its face value: 40%) as well as the latter's optional value (as per the upper bound: 60%+40%).

A currency's final remaining value, accounting for resources transferred out, represents the mandatory and (additional) optional amounts of resource available to the associated principal. In this example, these final values are (600,400) for A, (760,1340) for B, and (1140,960) for C. To explain B's values: $760 = (1900-$ M-Ticket3's value), and $1340 = (1140 + 200)$, where 1140 is the value of M-Ticket3 and 200 is the optional part passed from A via O-Ticket2. Section 3.1.1 provides

[1] Note that although we use terminology similar to some economics-based resource allocation schemes [29, 31, 16, 15], unlike them, our approach does not involve any competitive negotiation models.

general expressions for computing real values of currencies, accounting for possible cycles in the agreement graph.

In the above example, all currencies have a face value of 100, so that all face values of issued tickets are just the percentage number of represented agreements. In fact the face value of a currency can be an arbitrary number, which gives flexibility to change agreements by inflating or deflating the value of a currency.

3 Request Scheduling for Agreement Enforcement

Most existing resource management infrastructures [22, 24] provide only limited support for enforcing agreements, typically restricted to the end points. As our example in Section 1 demonstrates, such support is inadequate for enforcing sharing agreements involving distributed requestors and resources. An additional consideration is that the resource scheduler must keep track of resource availability via both direct and transitive agreements; simulation studies in our previous work [35] show that the latter can lead to significantly increased resource availability.

Figure 4(left) shows our agreement enforcement architecture, which coordinates multiple redirector nodes to schedule requests from distributed clients to servers. The ticket/currency-based scheme enables redirector functionality to be oblivious to the specific resource type being accessed. Each redirector node logically maintains a set of queues (see Figure 4(right)), one for each principal, and forwards a subset of these requests to servers every time window to satisfy the mandatory (MC_i) and optional (OC_i) request processing rates the principal is entitled to, computed as described below.

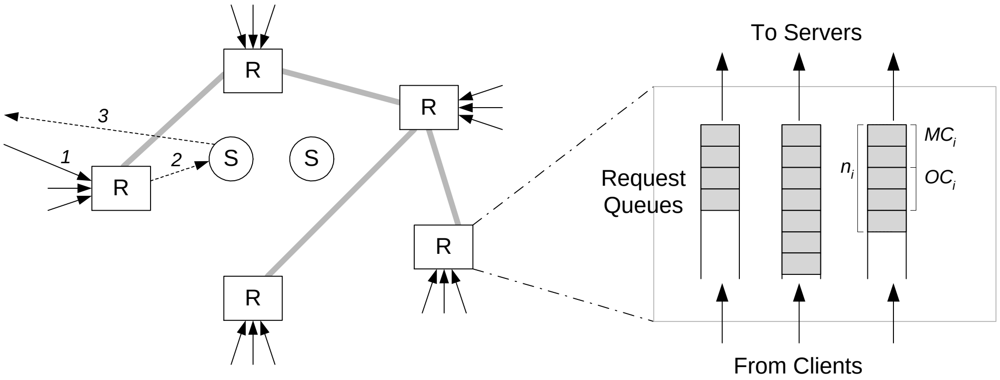


Figure 4: Agreements are enforced by coordinating multiple redirector nodes (R), which queue and forward requests from distributed clients to servers (S) so as to satisfy aggregate resource sharing agreements.

Stated differently, the redirector nodes perform admission control at the system edges to shape incoming request traffic to server clusters so that desired service levels are obtained. We currently assume that requests are short-lived (as compared to the time window) and resource consumption of each request is known a priori, either by specification or by profiling. An implication of this assumption is that the architecture does not need to track request completion. Requests for many types of services fit this model, with examples including web requests, small file transfers, etc. Extending our architecture to support longer lived requests such as continuous media streams would require additional (but orthogonal) support on the server side; such support would provide a sandbox or a resource container environment [18, 13] to ensure that the request complies with its allocated server share.

We first describe the queuing algorithms assuming a single redirector, and then extend these to multiple redirectors distributed across a wide-area network.

3.1 Scheduling using a Single Redirector

Given an agreement graph as input, the scheduler must (1) determine per-principal mandatory and optional request processing rates implied by both direct and indirect agreements; and (2) based upon request patterns observed at runtime, decide how to schedule each principal's requests to optimize a global system-wide criterion.

3.1.1 Per-Principal Mandatory and Optional Access Levels

For quasi-static agreement structures, mandatory and optional request processing rates can be pre-computed based on the agreement graph. For simplicity, our description assumes a single resource type: the agreement matrix consists of elements $[\mathrm{lb_{ij}}, \mathrm{ub_{ij}}]$ representing direct agreements. Principal i's physical resources are represented by their aggregate capacity V_i, scaled in terms of the average requirements of a request.

To compute the mandatory (MC_i) and optional (OC_i) processing rates for requests of a particular principal i, we first calculate the *flow* of mandatory and optional resources from each principal j to i (see Figure 5(left)). The flow of resources can themselves be expressed in terms of a recurrence relation involving the number of direct agreements along the path. Formulae 1 and 2 in Figure 5 give the expressions for these flows, $MI_{ji}^{(m)}$ and $OI_{ji}^{(m)}$, for paths involving at most m tickets. Formula 1 says that mandatory resources flow along mandatory tickets (lower bound) from one currency to another: the $(1 - \sum_k \mathrm{lb_{ik}})$ factor excludes the mandatory value i passes along to others (i.e. which leaks out of i) as shown in the figure. Formula 2 is more complex, capturing the fact that mandatory currency values contribute to mandatory resources (via mandatory tickets) up to a particular point in the path, and thereafter to optional resources (via an optional ticket at the specific point, and via agreement upper bounds beyond that point). The summation constraints ensure that there is no cycle along the transitive agreement path.

Although these formulae are complicated, note that they can be rewritten as:

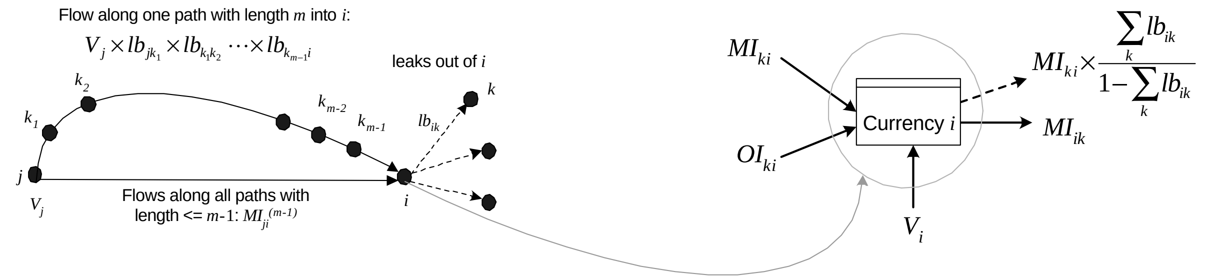
$$MI_{ji}^{(m)} = V_j \times MT_{ji}^{(m)} \quad \text{and} \quad OI_{ji}^{(m)} = V_j \times OT_{ji}^{(m)}$$

where $MT_{ji}^{(m)}$ and $OT_{ji}^{(m)}$ can be pre-computed from the agreement matrix.

Once the flows between each pair of principals has been computed, principal i's mandatory and optional access levels, MC_i and OC_i, are simply defined in terms of the transitive closure, MI_{ji} and OI_{ji} of the above expressions (see Formulae 3 and 4 in Figure 5). In essence, the computations described here allow us to reduce any arbitrary agreement graph to a series of simple expressions that relate the real currency values for each principal to the physical resources available in the system.

3.1.2 Queuing Algorithms

Knowing MC_i and OC_i for each principal i, the queuing algorithm must determine in each time window, what subset of the n_i requests in i's queue (see Figure 4(right)) must be forwarded to


Flow along one path with length m into i:
$V_j \times lb_{jk_1} \times lb_{k_1k_2} \cdots \times lb_{k_{m-1}i}$
leaks out of i
k_1
k_2
k_{m-2}
k_{m-1}
lb_{ik}
k
j
V_j
i
Flows along all paths with length <= m-1: $MI_{ji}^{(m-1)}$
MI_{ki}
OI_{ki}
Currency i
V_i
$MI_{ki} \times \frac{\sum_k lb_{ik}}{1-\sum_k lb_{ik}}$
MI_{ik}

$$MI_{ji}^{(m)} = MI_{ji}^{(m-1)} + V_j \times \sum_{\substack{1 \le k_1, \cdots, k_{m-1} \le n \\ k_p \ne i, j, k_q (\forall p \ne q)}} lb_{jk_1} lb_{k_1k_2} \cdots lb_{k_{m-1}i} (1 - \sum_k lb_{ik}) \tag{1}$$

$$OI_{ji}^{(m)} = OI_{ji}^{(m-1)} + V_j \times \sum_{\substack{1 \le k_1, \cdots, k_{m-1} \le n \\ k_p \ne i, j, k_q (\forall p \ne q)}} \sum_{r=1}^{m-1} lb_{jk_1} lb_{k_1k_2} \cdots lb_{k_{r-1}k_r} (ub_{k_rk_{r+1}} - lb_{k_rk_{r+1}}) ub_{k_{r+1}k_{r+2}} \cdots ub_{k_{m-1}i} \tag{2}$$

$$MC_i = V_i + \sum_{k \ne i} MI_{ki} - \sum_{k \ne i} MI_{ik} \tag{3}$$

$$OC_i = \sum_{k \ne i} OI_{ki} + \sum_{k \ne i} MI_{ik} + \sum_{k \ne i} MI_{ki} \times \frac{\sum_k lb_{ik}}{1 - \sum_k lb_{ik}} \tag{4}$$

Figure 5: (left) Mandatory resource flow from j to i via a transitive chain of agreements involving intermediate nodes $k_1, k_2, \ldots, k_{m-1}$. (right) The mandatory value of a currency i includes V_i, resources flowing into i and excludes resource flowing out from i. The optional value includes all optional resource flowing into i, plus all resource flowing out of i.

the servers. This decision should respect sharing agreements and at the same time optimize a global metric. The representation of agreements described in Section 2 is general enough to capture various kinds of metrics. In this section, we discuss two of them in different contexts. The first metric minimizes the maximum global response time seen by requests in a community context, while the second maximizes the income of the service provider in a service provider context.

Global Response Time Participants in a community contribute server resources to the resource pool and submit requests to the pool. The goal of our admission control is to minimize the maximum response time of all participants. This optimization is formulated as a linear-programming problem. Let x_{ij} be the number of requests from i's queue that will be scheduled to j's server. Our objective is equivalent to maximizing the minimum fraction of all queues to be processed, i.e., maximizing $\theta = \min_{1 \le i \le n} \sum_k x_{ik}/n_i$. Adding additional constraints to honor agreements, we come up with the following model:

$$\begin{array}{lll} \text{maximize} & \theta & \\ \text{subject to} & \sum_k x_{ik} \ge \theta n_i & \forall i \\ & \sum_k x_{ki} \le V_i & \forall i \\ & MI_{ki} \le x_{ik} \le MI_{ki} + OI_{ki} & \forall i, k \\ & \sum_k x_{ik} \le n_i & \forall i \end{array}$$

The first constraint just restates our requirements on θ. The second prevents the scheduler from assigning requests to use i's server that might exceed its capacity V_i. The third constraint, together with the access level formulae 3 and 4 computed earlier, forces compliance with the agreement lower and upper bounds[2]. Solving this linear programming model, the redirector obtains the queuing schedule that maximizes θ. Although each redirector node supports a large population of clients, the complexity of this strategy only depends on the number of principals involved in the agreements; this latter number is expected to be small.

Note that this model can be easily extended to take locality costs into consideration. We model locality as limiting the number of requests (c_i) that can be pushed to principal i's servers from the redirector in a time window, resulting in the following additional constraint:

$$\sum_k x_{ki} \leq c_i \quad \forall i$$

Total Income of Provider In a service provider context, we consider a simple price model, where a single service provider s negotiates, as part of the SLA, a price p_i with each customer i for each additional request processed by s beyond the mandatory service level. The goal of our admission control now becomes maximizing the provider's income while respecting agreements with all of its customers. This optimization can also be modeled as a simple linear programming model[3], involving x_i, the number of requests that should be processed for customer i in a given time window.

$$\begin{array}{lll} \text{maximize} & \sum_i p_i(x_i - MC_i) & \\ \text{subject to} & \sum_i x_i \leq V_s & \\ & MC_i \leq x_i \leq MC_i + OC_i & \forall i \\ & x_i \leq n_i & \forall i \end{array}$$

Note that although we have chosen to represent the underlying optimization problems as linear programming formulations, the architecture itself is general and flexible enough to host other optimization criteria and solving methods.

3.2 Coordinated Scheduling Across Multiple Redirectors

The queuing strategy as discussed above is not very scalable, because it assumes a single redirector that sees all requests. Our general solution extends the queuing algorithm to a distributed setting by observing that the single-node solutions can work on local redirector queues just the same as before as long as decisions about the fraction of the local queue to transmit to servers, $\frac{\text{xlocal}_{\text{ij}}}{\text{nlocal}_{\text{i}}}$, are based upon global values of the total queue lengths for each principal, i.e., $\frac{\text{xlocal}_{\text{ij}}}{\text{nlocal}_{\text{i}}} = \frac{x_{ij}}{n_i}$. Note that the queue length is an aggregate quantity of the system and can be computed much more efficiently than a neighbor-wise exchange of queue statistics.

In particular, we organize the multiple redirector nodes into a *dynamic combining tree* network (see Figure 4). Several algorithms exist for dynamically overlaying trees on a set of nodes in a wide area network, so we will not discuss this further. Redirectors at the leaves of this tree periodically send up queue length information to their parents. An intermediate node in the tree waits for information from its children, adds its local queue information to this, and passes on the information to its parent. When the queue length information reaches the root, this node sends the final aggregate information down the tree, effectively using the combining tree as a broadcast tree.

[2] The lower bound of this constraint needs to be dropped if the queue is not large enough, i.e. $n_i < MC_i$.

[3] As before, this model needs slight modifications if the queue is not large enough.

This scheme is scalable, requiring a total of $2(n-1)$ message transmissions (as opposed to $O(n^2)$ messages required for pair-wise exchange), however, it has the potential drawback that the computed queue length information is only an estimate that can lag actual conditions. We do not expect the latter to pose a practical problem as long as client request patterns do not vary dramatically over small time scales.

In addition to total queue length, other aggregate queue metrics such as the maximum, minimum, average queue length, and variation in queue lengths, can also be collected in the same fashion if schedulers need such information for other optimization metrics.

4 Prototype Implementations

We have implemented our redirection architecture in two different network layers: an application layer HTTP redirector (L7 switch) and a transport layer packet redirector (L4 switch), both sitting between clients and a clustered server system accessed using HTTP. The request URL signifies the service being requested. Both prototypes follow the structure shown in Figure 4: HTTP requests sent out by clients are routed to a redirector, which maintains a per-organization queue and makes scheduling decisions as discussed in Section 3. Large requests are treated as multiple small ones for the purpose of scheduling.

4.1 Layer-7 Redirection

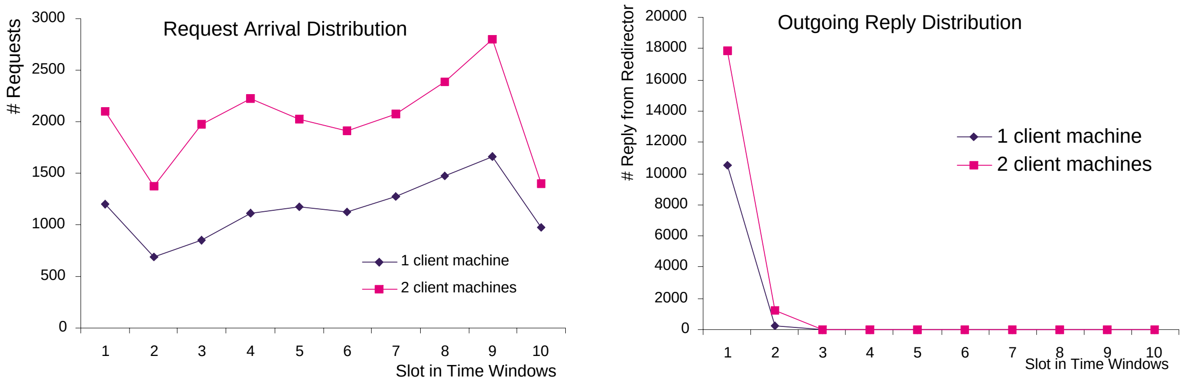
For requests that can be dequeued and sent to an assigned server according to the scheduling decision described earlier, our application level redirector sends an HTTP redirection message (with server information in its “Location” header) to the client, causing it to direct the request to the assigned server. If the request cannot be currently processed according to the service level limitation implied by the agreements, the redirector will send back a redirection header with its own address, which causes the client to resend the request. The redirector calculates scheduling decisions at the beginning of each time window by solving the linear programming models and multiple redirectors are connected to form a dynamic combining tree to periodically propagate queue length information, both as described in the previous section.

4.1.1 Implementation 1: Explicit Queuing

Our first application redirector implementation used explicit per-principal queuing: incoming requests were enqueued and scheduled at the start of the next time window. However, we found during our measurements by using a synthetic web server workload generator, WebBench [7], that server processing rates were not linearly increasing with increased client activity as we would have expected. With a single client, the server saw 115 requests/second, but this number only increased to 190 with two clients, and 230 with three.[4] This behavior could not be simply explained because neither the server nor the redirector became bottlenecks (CPU utilization stayed below 70%), and the network was underutilized (peak bandwidth into the server was 1.4MBps).

To investigate this problem, we measured request/reply activity at the redirector by breaking down each time window into 10 slots. Figure 6 shows these results, which explain our observed

[4] Because the WebBench 4.01 client cannot handle a redirection reply on its own, we added a modified Apache proxy on each client, which affects the base request rate more than any overhead in the redirector itself.



(a) (b)

Figure 6: (a) Request distribution and (b) Reply distribution seen at the redirector in the explicit queuing implementation.

behavior. Requests are spread over the entire time window as expected (the drop in the last slot is because the redirector is running the linear-programming algorithm and hence unable to receive new requests); however, almost all redirection replies are sent out in the first slot (right after the redirector makes the allocation decision). This bursty behavior is propagated via the client-side proxies to the server end and does end up overloading the server, degrading performance when there are multiple clients. Note that this behavior is not visible in the time-averaged CPU utilization at the server.

Thus, explicit queuing ends up bunching together requests that would otherwise have been spread out over time. This could be fixed by having the redirector spread its replies over the time window; however, this would increase queue management costs. Instead, we chose to use the implementation described below, which avoids explicit queuing altogether.

4.1.2 Implementation 2: Implicit Queuing

The main idea here is for the redirector to decide, for each time window, how many requests it can allow for each principal. The redirector still solves the linear-programming formulation, but with estimated queue lengths as opposed to actual ones. This allows requests that fall within a principal's quota to be immediately forwarded, and those that fall outside to be implicitly queued by simply sending a self-redirection message causing the client to retry. This scheme eliminates the anomalous behavior described earlier as can be seen in Figure 7, which shows the server processing rates for different numbers of clients. The single client number is 135, which doubles to 270 as expected with two clients. With three clients, the server saturates at 320 requests per second.

Although our application level redirection doubles the number of network round trips, this is more an artifact of HTTP as opposed to something fundamental. For instance, implementing the strategy within SOAP redirectors [8] would not encounter this problem. Moreover, the scheduling strategy itself can be deployed with other more efficient approaches such as those described in [17]. Below, we present one such implementation that operates as a transport layer (layer-4) switch.

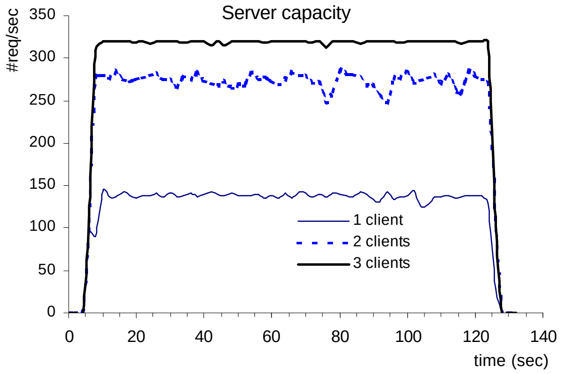


Figure 7: Three client machines saturate the web server with 320 requests per second.

4.2 Layer-4 Redirection

Our layer-4 redirector implementation is on top of Linux Virtual Server [33, 9], a basic framework to build highly scalable and available network services using a cluster of servers. Our redirector has two components: a Linux kernel module plugged into this framework and a user space daemon which communicates with the module.

The Linux Virtual Server framework provides three IP switch techniques: network address translation (NAT), IP tunneling (TUN) and direct routing (DR). While NAT and DR work best for a LAN environment, TUN can be used in wide area network situations where multiple redirectors are geographically distributed. Our implementation can use all three mechanisms without any modification, but we describe only NAT in this paper for simplicity.

Upon seeing a TCP connection establishment (SYN) packet from a client, the redirector chooses a server according to the scheduling decision passed from the user space daemon (see below), rewrites the destination address and the port of the packet to those of the selected server, forwards the packet to the assigned server, and records current connection information so that subsequent packets from the client are sent to the same server. All the work is performed inside the kernel, so that the overhead is low. When the response packets come back from the server, the redirector rewrites the source address and the port back to those of the redirector and forwards it to the client. If the module cannot admit the request according to scheduling decisions, it will put the packet into a kernel-level queue associated with the principal owning the target URL. Another kernel thread in the module periodically checks these queues, reinjecting packets back into the system in subsequent time windows as allowed by the agreements. Note that our implementation maintains connection affinity between client machines and servers to the extent allowed by the sharing agreements; this allows us to efficiently support services that rely on pairwise-negotiated security keys such as those based on the SSL protocol.

The user space daemon periodically collects queue length information from the kernel module, calculates scheduling decisions by solving the linear programming models discussed in Section 3, and feeds allocation information for the next time window into the kernel module.

5 Experimental Results

We use a synthetic server workload generator, WebBench [7] to evaluate the performance of our two prototype redirector implementations. All experiments use Apache web servers, 1 GHz PCs running Windows 2000 on a 100 Mbps switched network, and a WebBench configuration that produces static and dynamic web page requests with an average reply size of 6 KB (individual responses range from 200 bytes to 500 KB). Scheduling decisions are made over 100 ms time windows.

5.1 Application Layer Redirector

In this section, we present experimental results obtained from our layer-7 redirector prototype. We first verify that our system enforces sharing agreements in a service provider context and achieves minimum global response time in a community context. Then we show that our distributed coordination scheme can handle network delays gracefully.

5.1.1 Sharing Agreements in a Service Provider Context

To verify that the queuing scheme described in Section 3 does in fact respect sharing agreements, we set up a system (see Figure 8) involving three principals: the server owner and two organizations *A* and *B*. *A*'s requests are modeled by two WebBench client machines, while a third models *B*'s requests. *A*'s agreement with the server is [0.2,1], while *B*'s is [0.8,1]. Thus, in this system, *B* has more mandatory resources than *A*, but *A* has a higher request rate. The experiment itself runs in three phases. In the first and third phases, both *A*'s and *B*'s clients are active, while in the second phase only *A*'s clients are active.

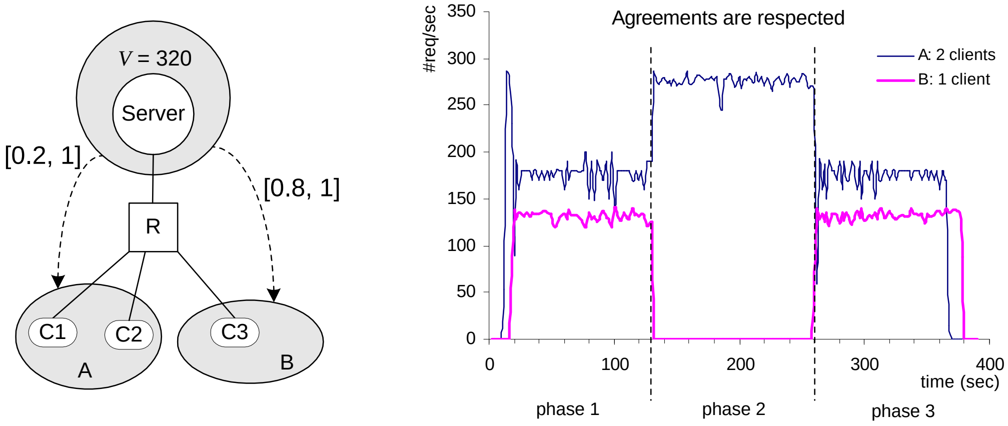


Figure 8: Sharing agreements are respected. *B*'s requests, when present, amount to less than its mandatory limit and are all responded to, while *A*'s requests use up the remaining resources.

Figure 8 shows the expected behavior: in the first phase, *B*'s requests from a single client (130 per second) are all satisfied because this number is below *B*'s mandatory limit of 80% of server capacity. *A*'s requests take up the remainder, around 190 requests per second. In the second phase, *A*'s requests can use up all of the server but are limited to 270 requests/second because of two client machines. The third phase shows that the system is able to respond to changing request patterns: *B*'s requests again get serviced at the desired rate.

5.1.2 Optimization of a Global Metric

In a community context, to minimize the maximum response time for requests across all organizations, the redirector should assign more resources to busier sites (after satisfying the mandatory limits). To verify that the optional tickets are indeed allocated according to incoming request rates, we focused on the first phase of the previous experiment, modifying it so that both *A* and *B* had agreements of [0.2,1] with the server owner. Server capacity was restricted to 250 requests per second to ensure that the results are not affected by a single client's ability to only generate 135 requests per second. Figure 9 shows the expected result: *A*'s requests are processed at twice the rate of *B*'s requests, optimizing community-wide average response time.

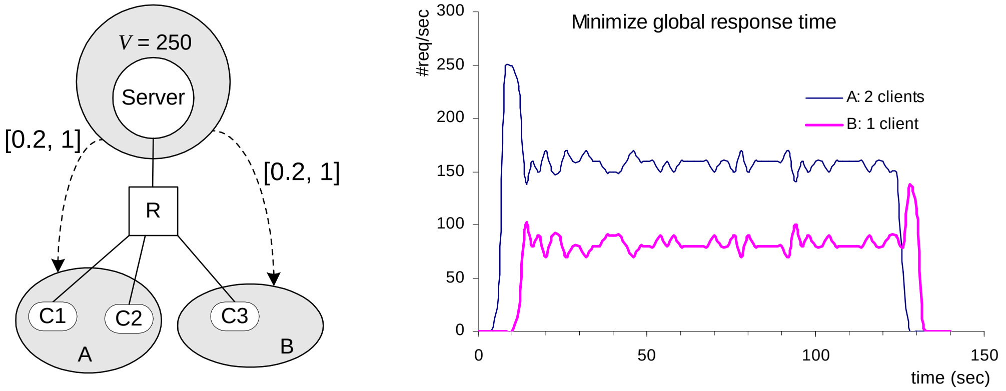


Figure 9: The scheduler optimizes a global response time metric by assigning more resources to community organizations with higher request rates.

5.1.3 Evaluation of Multiple Redirector Scheduling

The coordinated scheduling scheme follows the structure described in Section 3. The system configuration was changed so that it had two redirector nodes instead of our original one. Each client machine sent its requests to a different redirector. As described earlier, each of the redirectors make their own scheduling decisions based only upon aggregate load information obtained via the combining tree. Figure 10 and 11 shows the request processing rate obtained with multiple redirectors for the two experiments described earlier. As can be seen, the coordinated scheme obtains very similar results.

5.1.4 Impact of Network Delay

Since our experiments have been conducted in a LAN setting, it is possible that our results may not extend to WAN settings where communication delays are likely to be longer. To model the latter, we deliberately add delays in the combining tree, so that each redirector receives queue length information that has a lag of 10 seconds from actual values. This relatively large delay also accounts for the larger redirector trees one is likely to use in practice.

As before, two of the three client machines send requests for *A* and the other one for *B*. *A* has an agreement [0.8, 1] with the server and *B* [0.2, 1]. The system has two redirectors, one that handles

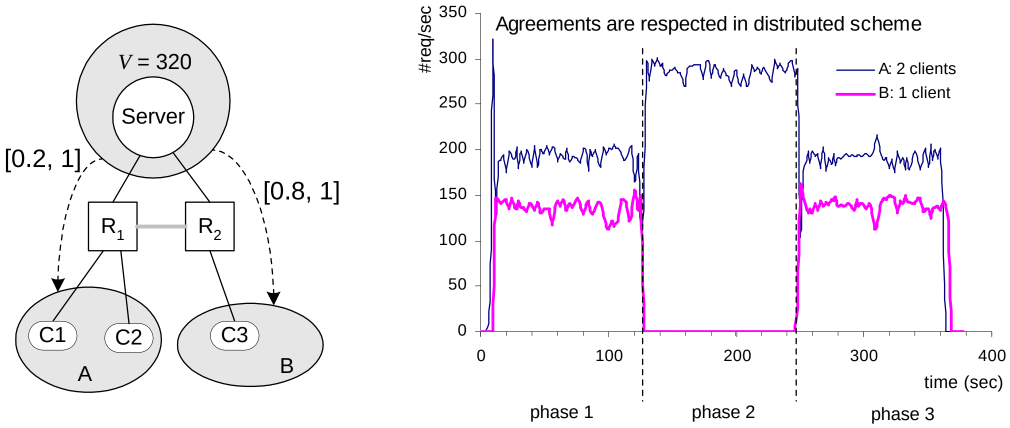


Figure 10: Scheduling with multiple redirectors showing sharing agreements are respected.

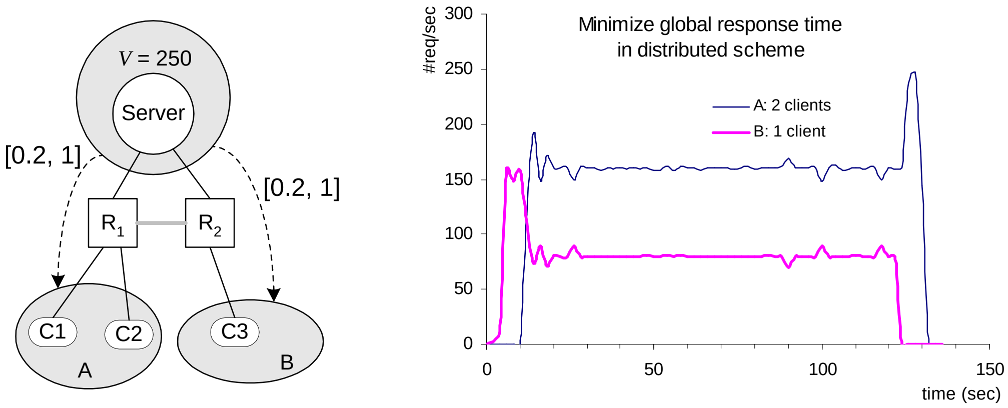


Figure 11: Scheduling with multiple redirectors showing global metrics are optimized.

A's requests and the other *B*'s (see Figure 12). The experiment proceeds in three phases: in the first and third phases, only *B* has requests, while in the second phase both *A* and *B* have requests.

Figure 12 shows how network delay affects system behavior. At the start (phase 1), *B*'s redirector does not know the status of the rest of the system: it therefore conservatively uses only of its mandatory tickets (i.e., half of 20% of 320) achieving a request processing rate of 30 requests/second. After about 10 seconds, when it receives global queue length information and learns that there are no other requests, it can use all of the server resources but is limited by the single client rate (phase 2). When *A*'s requests start arriving, there is a period of 10 seconds where *A*'s and *B*'s requests compete for the server (phase 3). However, once the information becomes available, each redirector changes its scheduling policy so that agreements are respected: *B*'s requests see a processing rate of 65 per second (20% of 320), while *A*'s requests are limited to 255 per second (80% of 320) (phase 4). Similar behavior is observed when *A*'s requests stop arriving (phases 5 and 6), verifying that as long as request patters are stable for time scales longer than network delays, our coordination scheme can gracefully cope with network delay.

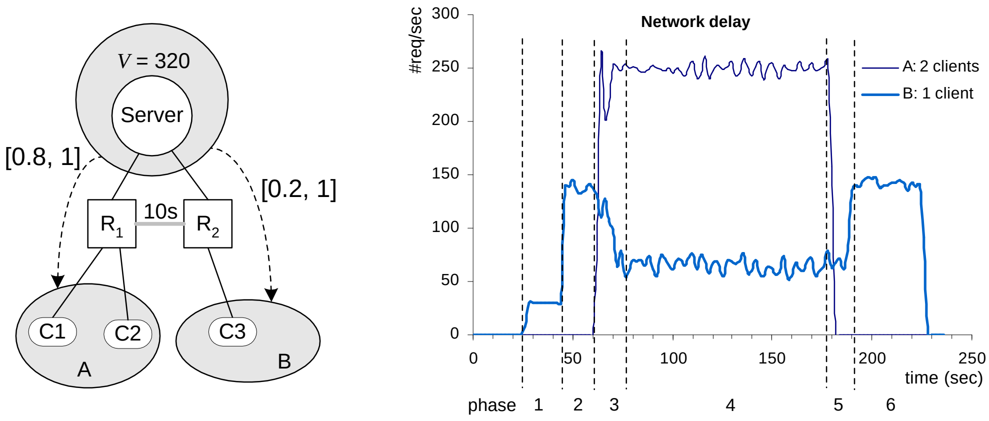


Figure 12: The multiple redirector scheme with a 10 second network propagation delay.

5.2 Layer-4 Redirector

In this section we present results of two experiments obtained using our L4 redirector prototype: one in a community context and another in a service provider context. Needless to say, the L4 redirector outperforms the application-level redirector in terms of its impact on request latency and bandwidth (the experiments reported below require less than 15% CPU usage on the redirector). However, a Layer-7 redirector might still be an appropriate implementation in situations requiring application-specific processing of requests at the redirectors (in addition to enforcement of resource sharing agreements).

5.2.1 Sharing Agreements in a Community Context

In this experiment, we setup a system with 2 principals *A* and *B* in a community. Both of them own a server with capacity 320. *B* shares its server with *A* according to the agreement [0.5, 0.5] (see Figure 13). The experiment runs in four phases. In the first phase, two WebBench client machines send requests for *A* and one for *B*. Since *A* can use *B*'s server to process 160 requests per second, *A*'s requests are processed at the rate of 480 per second, while *B*'s requests only receive service at 160 per second. In the second phase, both of *A*'s clients stop sending requests, therefore *B*'s requests are processed at the rate of 320 using the full capacity of *B*'s server. In the third phase, one of *A*'s client machines starts sending requests again, so processing rate for *A*'s requests can go back to 480 but is actually limited to about 400 because of the limitation of one client machine. At the same time *B*'s requests can be processed at the rate of 240 per second (*B*'s server needs to only process 80 *A*'s requests per second). Note that achieving this behavior is only possible because client requests are scheduled against the aggregate community resource. In the last phase, *A*'s client stops sending requests and the behavior of the system is the same as in the second phase.

5.2.2 Maximization of Service Provider Income

In this experiment, a service provider has two servers (each with capacity 320) and two customers *A* and *B*, with agreements [0.8, 1] and [0.2, 1], respectively. In addition, *A* pays more than *B* for each additional request processed beyond the mandatory processing rate. The provider will try to

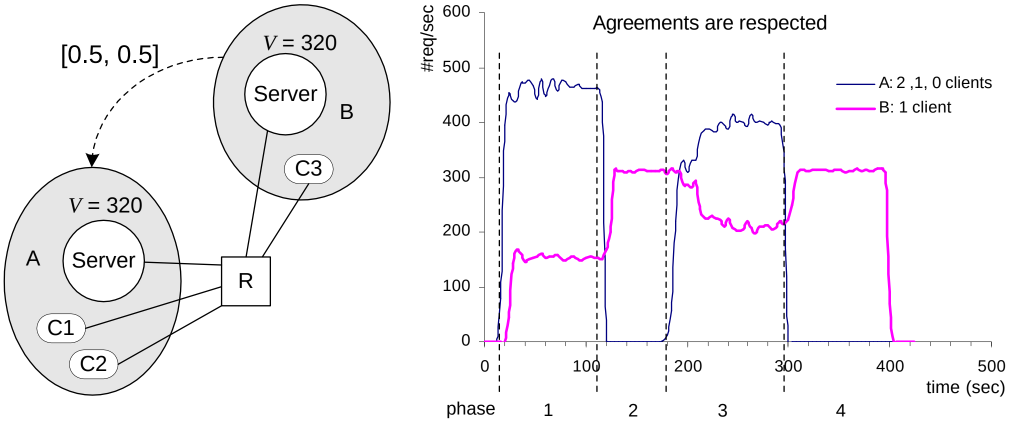


Figure 13: Resource agreements are respected when both *A* and *B* own their own server, and *B* shares 50% of its resources with *A*.

maximize its income while respecting agreements. Figure 14 shows the results. As in the previous example, in phase 1, two client machines send requests for *A* and one for *B*. We can see that *B*'s requests are only processed at the rate of 128 per second (20% of total server capacity, *B*'s mandatory rate), while all remaining server capacity is taken up by *A*'s requests, which generate more income to the provider. In the second phase, there are no requests from *A*'s clients, so all of *B*'s requests are processed, limited to 400 because of the number of requests that can be generated by a single client machine. In the third phase, one of *A*'s client machines again starts generating requests, which are given first preference to the server resource, while *B*'s requests take up the remaining capacity. Finally, the fourth phase, when *A*'s clients again become inactive repeats the behavior of the second phase. In all cases, the provider's revenue is maximized.

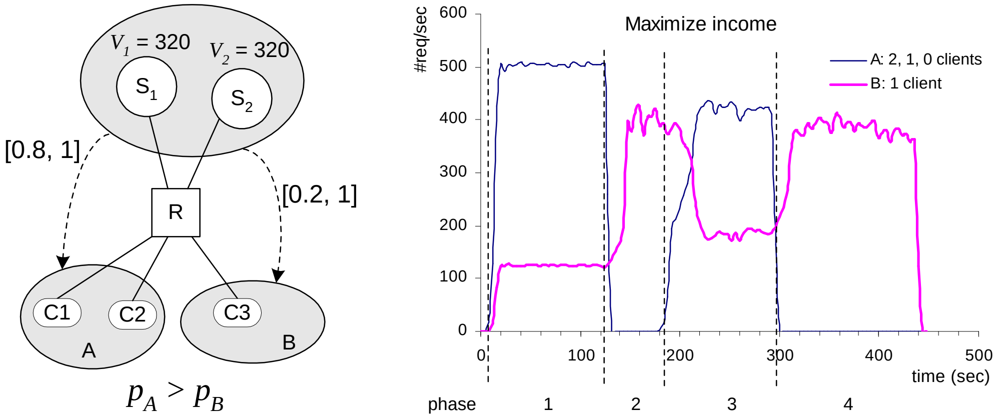


Figure 14: The redirector maximizes the income of the service provider by admitting as many requests for the highest paying customer as possible.

6 Related Work

Our work is closely related to five groups of previous work: large-scale resource management infrastructures [22, 24, 26], SLA enforcement in server farms [1, 13], request schedulers for cluster-based network services [10, 11, 12, 14, 20, 33, 28, 34], economy-based models for resource allocation [29, 31, 16, 15], and queue-based network QoS algorithms [19, 32, 27, 21].

Grid infrastructures such as Globus [22, 23], Legion [24, 25], and Condor [26] need to respect sharing agreements when pooling together resources belonging to multiple domains. However, existing solutions in these infrastructures have either left agreement enforcement to the end-points [26], or have limited themselves to providing an extensible architecture where resource management algorithms can be plugged in [22, 24] without explicitly providing a solution to the problem we describe.

Current-day techniques for SLA enforcement are perhaps best exemplified in the Océano [1] system, a "computing utility" infrastructure for multi-customer hosting on a server farm. It smoothes out peaks in customer requests by sharing the same resources (sequentially in time) among multiple customers. However, in large part due to non-uniform, application-specific SLA structures, Océano's resource allocation scheme operates at a relatively coarse granularity, requiring several minutes to carry out a new allocation decision. An advantage of our scheme is its finer-grained enforcement, which has the potential for impacting the granularity at which SLAs in Océano-like systems are specified.

Cluster-based network servers are now widely used. Commercial products [10, 11] are available to be used as front-ends for cluster servers. Most request distribution strategies fall into two categories: weighted round-robin and its variations for load balancing [12, 14, 20, 33], and content-aware load distribution for locality [28, 34]. Our work focuses on an orthogonal problem, of ensuring that requests from different customers receive service commensurate with pre-specified agreements. More related to our work is the Cluster Reserves technique [13], a mechanism to maintain desired cluster-wide resource allocations for different "service classes". The technique focuses on server resource partitioning (using operating system support for "resource containers") by relying upon availability of server load information at a centralized resource manager. Our technique is in some sense complementary, focusing on distributed admission control on the edges to ensure the right mix of requests arrive at the servers. As we note earlier, extending our technique to support longer-lived requests would require support similar to Cluster Reserves.

Our use of tickets and currencies overlaps in its terminology with economic models for sharing distributed resources; however, our approach is more traditional in its cooperative algorithmic focus. Although economic systems such as Spawn [29], MarketNet [31], Nimrod/G [16], and GRACE [15], can work well in slow-changing systems, they may fail to achieve competitive equilibrium because of dynamic perturbations of supply and demand in environments where resource availability and request patterns can change frequently. It is these latter environments that our techniques focus on.

Finally, our queuing strategy builds upon the same *virtual time* notion for proportional resource sharing that has been used in the context of network queuing algorithms [19, 32] and real-time multimedia CPU scheduling [27, 21]. However, unlike the explicit queue structures in these systems, we have found an alternative credit-based implementation more suitable to our distributed context. Another important difference is the fact that our decisions involve a more complex agreement structure and need to be coordinated across multiple nodes.

7 Summary

We have described an architecture for distributed, coordinated enforcement of resource sharing agreements based on an application-independent way to represent resources and agreements. We have successfully implemented this general strategy in two different network layers: a layer-7 HTTP redirector and a layer-4 packet redirector. Evaluation of both prototype implementations in the context of HTTP-based clustered servers shows that our approach can enforce predetermined resource sharing agreements with low overhead and dynamically adapt to load changes in a responsive fashion.

Acknowledgments

This research was sponsored by DARPA agreements F30602-99-1-0157, N66001-00-1-8920, and N66001-01-1-8929; by NSF grants CAREER:CCR-9876128 and CCR-9988176; and Microsoft. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as representing the official policies or endorsements, either expressed or implied, of DARPA, Rome Labs, SPAWAR SYSCEN, or the U.S. Government.

References

[1] Océano Project, IBM T. J. Watson Research Center - http://www.research.ibm.com/oceanoproject/.

[2] Akamai - http://www.akamai.com/.

[3] Napster - http://www.napster.com/.

[4] Gnutella - http://gnutella.wego.com/.

[5] SETI@Home - http://setiathome.ssl.berkeley.edu/.

[6] Entropia - http://www.entropia.com/.

[7] Webbench - http://www.zdnet.com/zdbop/webbench/.

[8] Simple Object Access Protocol (SOAP) - http://www.w3.org/TR/SOAP/.

[9] Linux Virtual Server Project - http://www.linuxvirtualserver.org.

[10] Cisco LocalDirector - http://www.cisco.com.

[11] IBM interactive network dispatcher - http://www.ics.raleigh.ibm.com/ics/isslearn.htm.

[12] D. Andresen, T. Yang, V. Holmedahl, and O. Ibarra. SWEB: Towards a scalable WWW server on multicomputers. In *Proceedings of the 10th IPPS*, 1996.

[13] M. Aron, P. Druschel, and W. Zwaenepoel. Cluster reserves: A mechanism for resource management in cluster-based network servers. In *Proceedings of the ACM Sigmetrics*, 2000.

[14] A. Bestavros, M. Crovella, J. Liu, and D. Martin. Distributed packet rewriting and its application to scalable server architectures. In *Proceedings of the 6th International Conference on Network Protocols*, 1998.

[15] R. Buyya, D. Abramson, and J. Giddy. Economy driven resource management architecture for computational power grids. In *International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, 2000.

[16] R. Buyya, D. Abramson, and J. Giddy. Nimrod/G: An architecture for a resource management and scheduling system in a global computational grid. In *Proceedings of HPC ASIA*, 2000.

[17] V. Cardellini, M. Colajanni, and P. S. Yu. Dynamic load balancing on web systems. In *IEEE Internet Computing*, June 1999.

[18] F. Chang, A. Itzkovitz, and V. Karamcheti. User-level resource-constrained sandboxing. In *Proceedings of 4th USENIX Windows Systems Symposium*, 2000.

[19] A. Demers, S. Keshav, and S. Shenker. Analysis and simulation of a fair queueing algorithm. In *Proceedings of ACM SigComm*, 1989.

[20] D. M. Dias, W. Kish, R. Mukherjee, and R. Tewari. A scalable highly available web server. In *Proceedings of the IEEE International Computer Conference*, 1996.

[21] K. Duda and D. Cheriton. Borrowed-virtual-time (BVT) scheduling: supporting latency-sensitive threads in a general-purpose scheduler. In *Proceedings of 17th ACM Symposium on Operating Systems Principles*, 1999.

[22] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(2):115–128, Summer 1997.

[23] I. Foster and C. Kesselman. The Globus project: A status report. In *Proceedings of the Heterogeneous Computing Workshop*, pages 4–18. IEEE Computer Society Press, 1998.

[24] A. S. Grimshaw, W. A. Wulf, J. C. French, A. C. Weaver, and P. F. Reynolds, Jr. Legion: The next logical step toward a nationwide virtual computer. Technical Report CS-94-21, Department of Computer Science, University of Virginia, June 08 1994.

[25] A. S. Grimshaw, W. A. Wulf, J. C. French, A. C. Weaver, and P. F. Reynolds, Jr. A synopsis of the Legion project. Technical Report CS-94-20, Department of Computer Science, University of Virginia, June 08 1994.

[26] M. J. Litzkow, M. Livny, and M. W. Mutka. Condor : A hunter of idle workstations. In *8th International Conference on Distributed Computing Systems*, pages 104–111, Washington, D.C., June 1988. IEEE Computer Society Press.

[27] J. Nieh and M. Lam. The design, implementation, and evaluation of SMART: A scheduler for multimedia applications. In *Proceedings of 16th ACM Symposium on Operating System Principles*, 1997.

[28] V. S. Pai, M. Aron, G. Banga, M. Svendsen, P. Druschel, W. Zwaenepoel, and E. Nahum. Locality-aware request distribution in cluster-based network servers. In *Proceedings of the 8th conference on architectural support for programming languages and operating systems*, 1998.

[29] C. A. Waldspurger, T. Hogg, B. A. Huberman, J. O. Kephart, and W. S. Stornetta. Spawn: A distributed computational economy. In *IEEE Transactions on Software Engineering*, 1992.

[30] C. A. Waldspurger and W. E. Weihl. Lottery scheduling: flexible proportional-share resource management. In *Proceedings of the First Symposium on Operating System Design and Implementation*, 1994.

[31] Y. Yemini, A. Dailianas, and D. Florissi. Marketnet: A market-based architecture for survivable large-scale information systems. In *Proceedings of Fourth ISSAT International Conference on Reliability and Quality in Design*, 1998.

[32] L. Zhang. VirtualClock: A new traffic control algorithm for packet-switched networks. *ACM Transactions on Computer Systems*, 9(2):101–124, 1991.

[33] W. Zhang. Linux virtual server for scalable network services. In *Ottawa Linux Symposium*, 2000.

[34] X. Zhang, M. Barrientos, J. B. Chen, and M. Seltzer. HACC: An architecture for cluster-based web servers. In *Proceedings of the 3rd USENIX windows NT symposium*, 1999.

[35] T. Zhao and V. Karamcheti. Expressing and enforcing distributed resource sharing agreements. In *Proceedings of Supercomputing*, Nov. 2000.