Service Level Agreement Based Distributed Resource Allocation for Streaming Hosting Systems *

Yun Fu, Amin Vahdat
Department of Computer Science, Box 90129
Duke University, Durham, NC 27708
{fu, vahdat}@cs.duke.edu

Abstract

The trend to outsourcing network services to third parties in a utility model has resulted in a new distributed application model where the hosting service, service providers and end clients constitute a co-dependent profit ecosystem. SLAs provide a means for the service providers to specify their target levels of performance and reliability and for the hosting service to arbitrate among competing services under resource constraint. In this paper, we propose a number of design principles for specifying and enforcing SLAs that allow service providers to obtain expected throughput from hosting systems. We propose and evaluate an algorithm, Squeeze*, based on pricing and penalties that allows the hosting service to maximize its profits while flexibly allocating available system resources among competing services based on prespecified SLAs and dynamically changing client access characteristics.*

1 Introduction

Burgeoning distributed hosting services introduce a new model for distributed applications. A *hosting service* establishes a distributed network where many edge servers are deployed on the edges of the Internet to provide low-latency and high-bandwidth network services to end clients. *Service providers* deliver the content to the hosting service, who publishes the data to a subset of available edge servers. The revenue of the hosting service comes from the service providers, who in turn derive their revenue from end clients. However, the popularity of individual services and the quality of services delivered by the hosting service determine how much each service provider is willing to pay. Thus, the hosting service, service providers and end clients constitute a profit ecosystem. In this ecosystem, efficiently allocating limited resources to benefit all three components is a challenging problem. Since the service providers cannot directly control resource allocation on the hosting service, Service Level Agreements (SLAs) contracted between the service providers and the hosting service are required to control the hosting service's resource allocation and client request processing. Given a set of SLAs, the hosting service requires a dynamic and adaptive resource allocation algorithm to maximize its own revenue from all service providers. Further, in the case where service resources are spread across the wide area, the system requires a distributed resource allocation algorithm that can provide global maximal revenue based on local resource usage information at each site.

Distributed hosting systems have many practical applications, e.g., Content Delivery Networks (CDNs) [1] and utility computing systems [5]. We propose two resource allocation models for hosting systems. One model is that each service provider not only delivers data to the hosting service, but also provides service applications, e.g. web servers or streaming servers, to be executed on the edge servers of the hosting service. In this case, the service applications determine the amount of resources re-

*The research is supported in part by the National Science Foundation (EIA-9972879), Hewlett-Packard, IBM, and Microsoft. Vahdat is also supported by an NSF CAREER award (CCR-9984328).

quired for current workload and apply for the resources from the hosting service. So the resource allocation mechanism of the hosting service is relatively simple in this model. It can simply allocate resources to the service provider with the highest bid while considering the SLAs contracted with other service providers. This model is more suitable for a general utility data center. The other model is that the service providers only deliver data to the hosting service and utilize service applications supplied by the hosting service. So the hosting service can fully control the resource usage for each service provider within the internals of the service applications and thus use resources more efficiently. This model can be adopted by CDNs. In this model, the hosting service needs to decide not only how to allocate resources but also how to utilize the resources more efficiently. Without direct resource control from service applications, the service providers can only specify their resource requirements for target levels of performance and reliability by carefully designing SLAs.

In this paper, we investigate resource allocation for a distributed hosting system for streaming multimedia content based on the second resource allocation model. We consider multimedia services because they are more challenging than standard web services. A streaming request can occupy system resources for an indefinite period of time as opposed to a web object request that typically lasts for milliseconds. Thus, carefully reserving and allocating limited resources for streaming requests are critical for a streaming hosting system. We propose a *Service Level Agreement based Streaming Hosting* (SLASH) system as a solution to resource allocation in a distributed streaming service. SLASH can effectively adjust resource allocation for all streaming content it serves based on predefined SLAs.

Before discussing how to define SLAs, we must first determine what resources should be controlled and specified in SLAs. We assume one major reason that service providers would outsource SLASH is because they cannot economically satisfy their customer's bandwidth requirements. Thus, intuitively, SLAs can be specified for how much bandwidth or how many concurrent connections SLASH should provide to the service providers. However, if an SLA directly specifies how much bandwidth should be reserved for a service provider, it is difficult to accurately estimate the revenue loss due to the shortage of the reserved bandwidth since it depends on the request load and the scheduling algorithm. On the other hand, streams with different lengths occupy system bandwidth for different periods of time. Therefore, the resources specified in SLAs should reflect not only space issues (bandwidth), but also time issues (how long the bandwidth is occupied). Thus, we use *shares*, a bandwidth unit used during a system scheduling epoch, as the target resource in SLAs.

Given a set of SLAs specified from all service providers, a hosting center must determine how many resources should be deployed to satisfy each customer's request load, and how to allocate and control resource usage among all hosted services. Obviously, it is not wise to overprovision for the maximum amount of resources required by the peak request load from all hosted services. Sharing resources among hosted services allows for better resource utilization and in turn results in larger profits. Further, since streaming requests occupy system resources for an indefinite period of time, only sharing without resource partitioning may cause the hosting service to work as a work-conserving scheduler, where low-priority services may consume more resources than high-priority ones. For example, in a First Come First Serve (FCFS) system, the resources may be mostly occupied by the service with the highest request load, which may not correspond to the service who would pay the most. Meanwhile, due to the specified SLAs, a penalty may be charged to the hosting service when a minimum level of resources is not delivered to a given service. In some cases, the corresponding profit may outweigh this penalty, leading the hosting service to temporarily accept the penalty.

In this paper, we propose a resource allocation algorithm, *Squeeze*, to efficiently and dynamically allocate resources for hosted services according to SLAs. Since the revenue that a hosting service can obtain is the major concern of the hosting service, we use revenue as the pri-

mary criterion for evaluating different resource allocation mechanisms. We will show that the *Squeeze* algorithm can effectively maximize the revenue of a hosting service by flexibly allocating resources among competing services.

Section 2 describes related work. Section 3 introduces the architecture of SLASH. Section 4 describes our considerations on the design of SLAs and the *Squeeze* resource allocation algorithm. Finally, Section 5 presents the experimental results to show the correctness and efficiency of SLASH implementation on resource control.

2 Related Work

James Kurose and Rahul Simha [10] proposed a microeconomic approach to allocating distributed resources for file allocation problems (FAP). In their system, a number of computers are fully connected into a communication network. Each node stores a part of the entire file system and can generate file access requests, which can be satisfied on either the local node or a remote node. For a given communication cost and request load assignment on each node, the system determines an optimal allocation of the portion of the file system that should be placed on each node. They define a utility function, which only considers the communication cost of the system, as the target optimization goal. They propose two gradient-based algorithms, which can converge to an optimal solution for file allocation. They also present a pairwise interaction algorithm to implement the resource allocation in a distributed manner.

Muse [6] is a recent work considering resource allocation in hosting centers. Muse can dynamically allocate an active server set for a service based on negotiated SLAs and cost, specifically the cost of power. The MSRP resource allocation algorithm proposed in Muse is also a gradient-based algorithm, which depends on a concave utility function. Staring from an initial resource allocation, Muse reassigns one unit of resource at each iteration in a greedy manner. Eventually, the algorithm can stop at an optimal solution. A similar incremental algorithm for optimally allocating discrete resources was previously discussed by Toshihide Ibaraki and Naoki Katoh [7]. In our work, we show how to define SLAs that can cause the revenue function of each service to be a concave function, which is then utilized by the *Squeeze* algorithm to compare and select the best candidate among all services to allocate resources.

In this paper, we focus on bandwidth as the target resource for allocating. For other hosting services with more complicated resources, such as application hosting services, some existing techniques can be utilized to isolate the resource usages of cohosted services on a single node [4] or within a cluster [3]. The hosted services can even run on a virtual host [11] where resource partitioning is completely transparent.

3 System Architecture

Figure 1 depicts the architecture of SLASH, consisting of a set of distributed edge servers and switches dispersed across the edges of the Internet. Each edge server is intended to serve a group of nearby clients. A number of edge servers are grouped together and managed by a nearby switch. Switches are interconnected as a front-end interface of the system to process client requests.

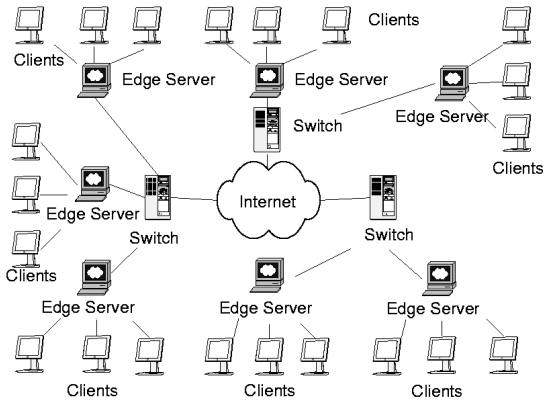

Figure 1: *SLASH architecture*

Using DNS server selection techniques or client customized preferences, client requests can be routed to a nearby switch, which redirects the clients to access appropriate edge servers. The switch identifies the appropriate

edge servers by, for example, utilizing existing client clustering technologies [9] or simply grouping clients by their ASes. To select edge servers for clients, the switch must collect the load status of the edge servers. In SLASH, edge servers regularly report their load status to their local switch. Furthermore, a switch must maintain the content of all the managed edge servers and inform the edge servers to retrieve stream files from original sources. To further utilize system resources, switches can also exchange information to shift request load among one another and to cooperate to manage distributed resource allocation to obtain globally optimal performance.

In SLASH, switches implement server selection by utilizing RTSP messages [13] to redirect client requests to appropriate servers. The streaming data transport is implemented by RTP [12]. When a client initiates a stream connection, it sends an RTSP DESCRIBE request to the closest switch. The switch selects an edge server for the client based on the resource control algorithm and sends back an RTSP response with status code 302, where the selected server is set in the Location field. The client stream viewer then establishes a new connection to the specified server to retrieve the data. To prevent clients from arbitrarily selecting an edge server, which can cause the switch to lose control over resource allocation, SLASH also encrypts the issued time and the client IP address at the end of the Location field. When the Location field is forwarded to the edge server, the server can verify the validity of the redirection. SLASH also adopts a communication mechanism for the switches to maintain a correct view of availability, resource usage, and supplied content of each edge server. For example, the edge server should inform the switch about newly established or closed connections. So the switch can estimate the load on each server.

4 Resource Allocation

To efficiently utilize limited resources in SLASH, we must consider all factors that can affect system performance, such as resource allocation, data placement and request routing. SLASH can simultaneously serve many service providers. Each service provider has many stream files with different content or different qualities, which in turn have different popularities. Through resource allocation, SLASH can reserve some amount of resources for each service provider or even for different quality streams of one service provider to maximize the global performance or the revenue of the entire system. We will show that by allocating resources according to content popularities and SLA specifications, the system can obtain higher overall revenue from customers.

4.1 Price and Penalty Design

Currently, the only distributed resource we consider is bandwidth. We assume each server has limited bandwidth to serve all service providers. We also assume streaming connections are rigid connections, which cannot be preempted by other new connections after they are established. In this paper, we do not consider variable bitrate streams. So if a connection is established, it can neither be disconnected by the server nor dynamically adjust its bitrate.

Based on SLAs contracted with service providers, SLASH should provide quality-guaranteed services and obtain revenue from the service providers. Since SLASH simultaneously hosts multiple services, it always attempts to utilize available resources to serve as many requests as possible. Thus, the resources reserved for a service provider according to SLAs may be utilized by SLASH to serve other service providers if the system can obtain more profit. However, if the quality of the service is affected or the throughput is reduced because of a shortage of deserved resources, a penalty must be charged to the hosting service. Relative to existing hosting techniques where resource allocation is implemented by reserving a set of machines for a given service, SLASH can efficiently multiplex competing hosted services and allocate resources for them at the granularity of shares.

SLAs are specified from many perspectives. One major consideration is that the revenue and penalties described by SLAs can be accurately calculated and are fair to both service providers and hosting services. The service providers can calculate the revenue and

penalties off-line based on system logs and verify the computation based on statistical results provided by third-party monitors. Meanwhile, revenue and penalties must also be easy to estimate online since SLASH needs to estimate the possible tradeoff between the revenue to benefit a service and the possible penalty from another service when adjusting resource allocation between the two services.

We define SLAs to direct SLASH to provide necessary resources for all services, and to stimulate SLASH to allocate spare resources for the services with high request load or high priorities. Two major metrics in an SLA are: *revenue prices* and *penalty prices*.

The revenue price of a stream file can be defined as the profit earned per time unit for a connection of the stream file. For the same steam file, if the system can serve it longer, the service provider should pay more for it. Streams with different qualities can be of different prices. For example, higher quality streams deserve higher prices. Thus, the revenue prices can stimulate SLASH to serve more high-quality streams for customers if there are sufficient resources.

Revenue prices can be fixed (flat) prices or a function of a set of input arguments [6]. One possible argument of this function is the throughput for a service, for example, if a service provider desires to sustain 1000 requests per minute. If the actual throughput is larger than this, the service provider can reduce the price to avoid overcharge for unexpected, high request load. On the other hand it may be willing to pay a lot more for load beyond a given level because it signifies a flash crowd or some unexpected events (major breaking news story). In this paper, we use fixed prices to simplify our discussion.

Compared to revenue prices, penalty prices are more difficult to define since they are based on an estimate for the possible profit loss due to the shortage of deserved resources at any time. Generally, it is not easy to accurately and fairly calculate the loss. For a given request load and a certain amount of resources, the throughput that it can generate highly depends on the scheduling algorithm. For example, assume that in Figure 2 the system should allocate 300Kbps for a service. There are four requests Req_1, Req_2, Req_3 and Req_4 for a stream of the service. Req_1 and Req_3 are for the low-quality (100Kbps) version of the stream. Req_2 and Req_4 are for the high-quality (300Kbps) version of the stream. The profit obtained by providing the high-quality stream is higher than the low-quality stream. In Figure 2, the system rejects Req_1 and Req_3 and accepts Req_2 and Req_4 to maximize overall profit. Obviously, if the system accepts Req_1 and Req_3, it cannot process Req_2 and Req_4 because the bandwidth is occupied at that moment. But, since the system does not serve Req_1 and Req_3, some resources guaranteed by the SLA are not allocated at the moment, causing a penalty to be charged. However, this is "unfair" since the hosting service provides a better service for the service provider and the service provider does not lose any of its deserved profit (assuming that end clients are paying it).

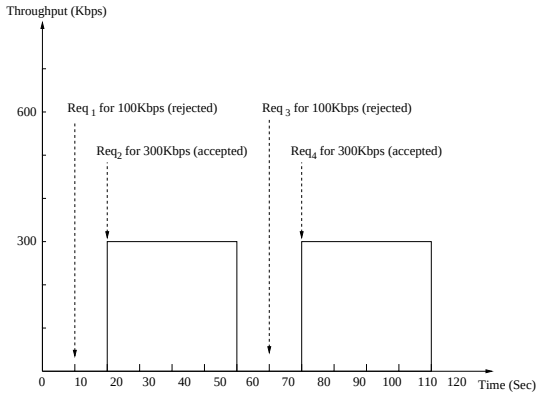

Figure 2: *Req_1 and Req_3 for a low-quality (100Kbps) stream file are rejected. Req_2 and Req_4 for a high-quality (300Kbps) stream file are accepted.*

Even if both the service provider and the hosting service agree to pay a penalty in this case, we still need to consider whether the penalty should cover the entire loss of Req_1 (or Req_3), which can last up to 40 seconds, or only the loss before Req_2 (or Req_4) begins to be served, which is only 10 seconds. While the latter appears to be fairer, it means that we need to simulate the scheduling of rejected requests to estimate how much penalty should be paid for a given request load. Thus, it is not practical to determine if a single rejected request should be charged for a penalty by the resource

allocation at any particular moment. Penalties should be estimated based on the offered load and the profits generated during an interval.

Another consideration for SLAs is the amount of time an individual request occupies while consuming resources at an edge server. Considering that profit is proportional to stream length, the loss from rejecting a short stream is less than the loss from rejecting a long stream if their revenue prices are the same. Thus, the specification of penalties should also include stream lengths.

We assume a limited number of encodings of the same stream, with bitrates all multiples of a bandwidth unit, say 100Kbps, e.g., 100Kbps, 300Kbps, 500Kbps. We estimate the stream length in a specified time interval, an *epoch*. Epochs should be long enough to contain enough data samples to measure the throughput and penalties. When SLASH accepts a stream request whose bitrate is one bandwidth unit and the stream duration is 1 epoch, we say the system sells 1 share at that moment. Thus, the shares of a stream is its bandwidth units multiplied by the stream length in epochs. We define the *trading volume* of an epoch as the number of shares SLASH sells. A trading volume can be directly used to compare with the request load in terms of shares. For example, assume an epoch is 10 seconds and the bandwidth unit is 100Kbps. In Figure 3, the system receives 1 600Kbps request by 1 epoch long in each epoch. In Figure 4, the system receives 1 100Kbps request by 6 epochs long in each epoch. If the SLA specifies that the system should serve 6 shares per epoch, serving either of the request loads in Figure 3 or 4 can satisfy the SLA. Thus, service providers do not contract with SLASH directly by bandwidth, but by shares. Notice that the trading volume of an epoch is different from the throughput. For example, in the first epoch of Figure 4, its throughput is only $100\,Kbps * 1\,epoch = 1\,Mbit$. But its trading volume is 6 shares ($6\,Mbit$).

Shares form the basis for SLAs between service providers and hosting centers. With shares and epochs, we can define the price of a stream as the profit in an epoch per connection ($\$/epc$) or directly as the profit for each share ($\$/share$).

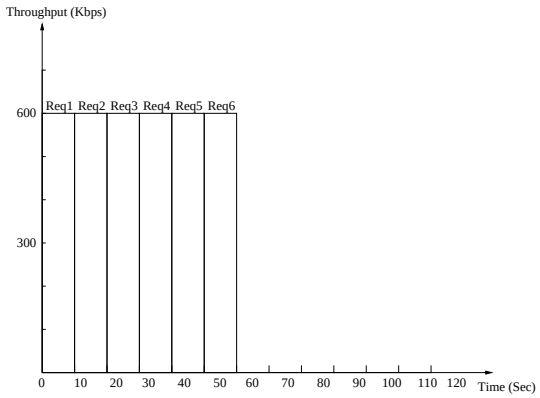

Figure 3: *1 request for a 600Kbps stream by 1 epoch long in each epoch.*

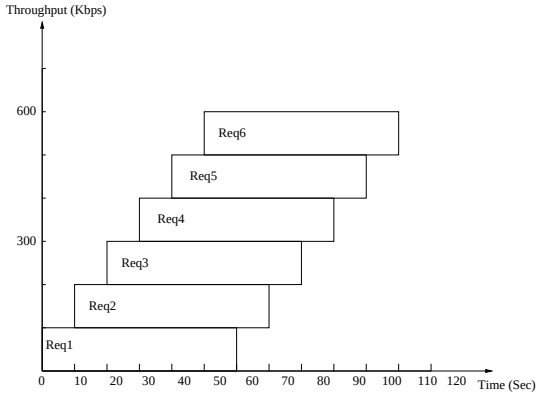

Figure 4: *1 request for a 100Kbps stream by 6 epochs long in each epoch.*

We now formally define penalties by shares. To obtain a certain quality of service, a service provider pays extra money to reserve a number of shares, Σ. Assume the offered load of the service during an epoch is Θ, and the trading volume during that epoch is Λ, where $\Lambda \leq \Theta$. Then, we can define the penalty as:

$$penalty(\Theta, \Lambda) = \begin{cases} 0 & \text{if } \Lambda \geq min(\Sigma, \Theta), \\ P * (min(\Sigma, \Theta) - \Lambda) & \text{if } \Lambda < min(\Sigma, \Theta), \end{cases}$$

where P is the SLA specified *penalty price* for a share. It could be the same as the price that the service provider pays for reserving the Σ amount of shares. Thus, the penalty can be exactly the compensation for the loss of reservation. The penalty price can also be higher than the price used to reserve resources. In this

case, the penalty can refund the money used to reserve resources and compensate the possible loss due to rejection of client requests. Since the penalty calculation does not distinguish which stream files those lost shares are for, the penalty price is the same for all stream files of a service, even for different bitrate versions.

4.2 SLA Based Control

The reason we introduce prices and penalties into SLASH is to provide a means for service providers to control SLASH resource allocation under highly variable client request loads. Higher prices can cause SLASH to allocate more spare resources to a particular service. For a single service, higher prices for high-quality streams can stimulate SLASH to serve more high-quality streams if available bandwidth is sufficient. However, if the bandwidth is limited, the service provider may prefer to serve more connections with low-quality streams using the same amount of resources. Thus, the service provider can define the revenue price of each share for low-quality streams to be higher than for high-quality streams.

We use an example to illustrate some design rules for revenue prices and penalty prices. Assume we have only two services, service A (S_A) and service B (S_B), competing for available resources in a system. The system resources are divided into two parts and reserved by the two services. Both services have a low-quality stream and a high-quality stream. They are the same length, n epochs. The bitrate of the low-quality streams is β_l bandwidth units and β_h bandwidth units for high-quality streams. We define the price for the high-quality stream of S_A as $Pr_{(A,h)}$ in $\$/epc$. Correspondingly, we define $Pr_{(A,l)}$, $Pr_{(B,h)}$ and $Pr_{(B,l)}$. The penalty prices for S_A and S_B are Pe_A and Pe_B respectively. To stimulate SLASH to serve more high-quality streams, we set

$$\begin{aligned} Pr_{(A,l)} &< Pr_{(A,h)} \\ Pr_{(B,l)} &< Pr_{(B,h)} \end{aligned} \tag{1}$$

Our first goal in this example is to allocate resources fairly between the two services. Thus, we set $Pr_{(A,l)} = Pr_{(B,l)}$ and $Pr_{(A,h)} = Pr_{(B,h)}$. Next, we want the system to serve more low-quality streams or downgrade high-quality stream requests to low bitrates when resources are scarce. So we define the following relationship:

$$\begin{aligned} Pr_{(A,h)}/\beta_h &< Pr_{(A,l)}/\beta_l \\ Pr_{(B,h)}/\beta_h &< Pr_{(B,l)}/\beta_l \end{aligned} \tag{2}$$

To maintain fair allocation between the two services, we must ensure that one service does not occupy the other service's reserved resources even when its request load is high enough to force it to begin to downgrade high-quality streams to low-quality streams. It can be easily shown that SLASH will first use the resources reserved for the service itself to serve low-quality streams instead of using the other service's resources only if $Pe_A \neq 0$ and $Pe_B \neq 0$.

However, under circumstances where the request load is high enough such that all resources reserved for one service have been consumed for its low-quality streams, SLASH should utilize the resources reserved from the other service if the other service still reserves some resources for high-quality streams. Thus, we can set the prices as:

$$\begin{aligned} \frac{Pr_{(A,l)}}{\beta_l} - Pe_B &> \frac{Pr_{(B,h)}}{\beta_h} \\ \frac{Pr_{(B,l)}}{\beta_l} - Pe_A &> \frac{Pr_{(A,h)}}{\beta_h} \end{aligned} \tag{3}$$

So the hosting service can earn more profit from each share by serving low-quality streams even if it has to pay a penalty for failure to provide enough resources for high-quality streams of the other service.

With the above constraints, the hosting service always attempts to provide enough resources for both services. If all available resources are allocated to the two services and one service experiences high request load, the hosting service first downgrades requests for high-quality streams to low-quality streams for the service to utilize resources more efficiently without paying a penalty. However, if the request load for this service becomes even higher and all resources for the service are consumed for low-quality streams, the hosting service starts to allocate more resources for this service if the

other service still has some resources for high-quality streams. In this case, although the hosting service has to pay a penalty for the other service, it still can earn more profit by serving more low-quality requests for the busy service.

Equations (1) and (3) define a set of constraints for possible solutions. We use concrete numbers to show possible solutions for the above constraints. For example, β_h is 3 bandwidth units. β_l is 1 unit. Assume $Pr_{(A,h)} = Pr_{(B,h)} = y$ and $Pr_{(A,l)} = Pr_{(B,l)} = x$. We want the penalty price to be at least high enough to compensate the loss of a possible high-bitrate stream. So we set $Pe_A = Pe_B = y/\beta_h = y/3$. It is easy to show that x and y are constrained by $y > x$ and $y < 3x/2$. As illustrated by Figure 5, point P $(9, 12)$ is a possible solution for (x, y). So we can define $Pr_{(A,l)} = Pr_{(B,l)} = 9$, $Pr_{(A,h)} = Pr_{(B,h)} = 12$, and $Pe_A = Pe_B = 3$.

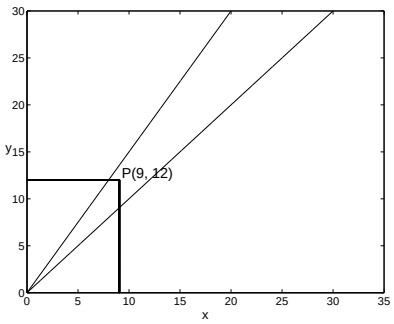

Figure 5: (x, y) *is constrained by two lines:* $y > x$ *and* $y < 3x/2$. $P(9, 12)$ *is a possible solution.*

4.3 Squeeze Resource Allocation Algorithm

Requests to streaming servers can last for many epochs and occupy system resources during this period. Thus, resource allocation must consider both space (bandwidth) and time limitations of the system. As mentioned above, shares are the resources that service providers reserve with SLAs. For a system with β bandwidth units, it can serve β shares continuously in each epoch. Thus, the resources that SLASH can allocate in an epoch is β shares. Shares include both space and time considerations.

The resource allocation mechanism actually contains two steps: *resource allocation* and *admission control*. At the end of each epoch, the system adjusts the amount of resources allocated for each service. Based on this resource allocation, the system admits or rejects client requests to the service during the next epoch.

SLASH keeps track of the request load for each bitrate in every service. We assume the prices of streams with the same bitrate for one service are all the same in this paper. However, in practice, if the prices are different, we can still dynamically estimate the average price for a bitrate in one service. At the end of each epoch, we use an exponentially weighted moving average (EWMA) filter [8] to estimate the request load for every bitrate in each service.

Before discussing our share allocation algorithm, we use an example to illustrate our principles for fully utilizing allocated resources. Figure 6 describes the example and one solution. Figure 7 describes the process for obtaining this solution.

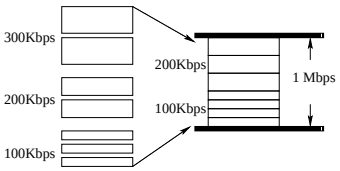

Figure 6: *A bandwidth unit is 100Kbps. The available bandwidth for the service is 1Mbps. To squeeze 3 100Kbps, 2 200Kbps, and 2 300Kbps requests, the system downgrades them to 4 100Kbps and 3 200Kbps streams.*

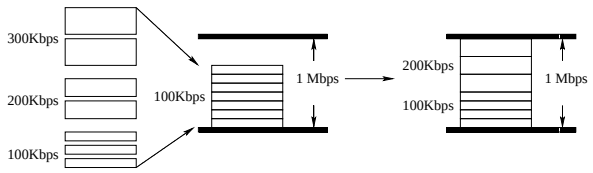

Figure 7: *The system first estimates how much bandwidth is occupied when downgrading all streams to the lowest bitrate. Then the system upgrades those requests whose bitrates are higher than the lowest one to the second lowest bitrate to see how many requests can be upgraded. This process repeats until all bandwidth is consumed.*

Similar to our earlier example, for one service, given a certain request load, Θ_1, Θ_2,..., and Θ_m (shares) for bitrates β_1, β_2, ..., and β_m respectively (to simplify our discussion, assume β_1 is the lowest bitrate the system can downgrade to. Generally it is 1 bandwidth unit), we need to determine how many shares the system should allocate for each bitrate. Assume the revenue price of β_i is P_i ($/epc$), and the penalty price of this service is Pe ($/share$).

First, we need to determine the incremental profit the system can obtain when allocating one more share for a particular service. We can show that there are $m+1$ (or m) critical points on share allocation for each service, which we should calculate before comparing this service with other services. Initially, we ignore the penalty price. The first critical point is the number of shares where all bitrate streams should downgrade to β_1 to fit the allocated shares. It is $C_1 = \Sigma_{i=1}^{i=m}(\Theta_i/\beta_i)$ shares. When the resources allocated for this service is less than or equal to C_1, the price of each share is P_1/β_1, or simply P_1. The second critical point is $C_2 = \Theta_1 + \Sigma_{i=2}^{i=m}(\beta_2 * \Theta_i/\beta_i)$ shares. With C_2 shares, all β_1 streams are served at bitrate β_1. Other streams are served at a bitrate of β_2. Correspondingly, other critical points are $C_i = \Sigma_{j=1}^{j=i-1}\Theta_j + \Sigma_{j=i}^{j=m}(\beta_i * \Theta_j/\beta_j)$ shares. When the system changes resource allocation from C_i to C_{i+1} shares, the profit for each newly allocated share is $P_i' = (P_{i+1} - P_i)/(\beta_{i+1} - \beta_i)$. After obtaining all m critical points, now we consider the penalty. Assume the service reserved resources are Σ shares, it introduces another critical point Σ (if $C_i \neq \Sigma$ for $\forall i$). For all points with $C_i < \Sigma$ ($1 \leq i \leq t$, assuming there are t such points, where C_t is the highest point), if the system changes the allocation from C_i to C_{i+1} (or to Σ for C_t), the profit for each newly allocated share is not P' anymore, but $P_i' + Pe$, which means the hosting service can pay one share less penalty by allocating one more share for the service when the allocated resources for the service is less than the reserved resources Σ.

With the request load in Figure 6, Figure 8 illustrates the revenue increase graph with share allocation. The gradient of each segment is the price for a newly allocated share. It can be proven that the revenue increase curve is concave if $P_1 < P_2 < ... < P_m$ and $(P_1/\beta_1) > (P_2/\beta_2) > ... > (P_m/\beta_m)$ as defined in subsection 4.2.

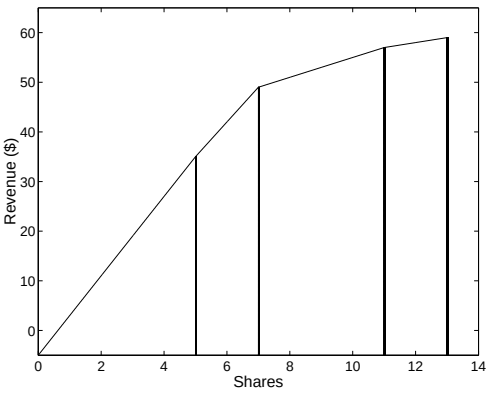

Figure 8: *For the request load in Figure 6, assume each request length is one epoch. Assume the service reserves 5 share resources with a penalty price* $1/share$. *The prices of* $100Kbps$, $200Kbps$ *and* $300Kbps$ *are* 7$, 9$ *and* 10$ *respectively. We can observe that the price increase for each share is* 8$ *from 0 to 5 shares,* 7$ *from 5 to 7,* 2$ *from 7 to 11 and* 1$ *from 11 to 13.*

For each service, we have a graph similar to that of Figure 8, which shows $m+1$ (or m) critical segments. Thus, we know the prices for each share increases in these services, which are the gradients of those segments. With this data, we introduce our resource allocation algorithm, *Squeeze*. Assume we have N services (S_1,...,S_N) and the system available resources are U shares, the *Squeeze* algorithm operates as follows,

- Step 1: Calculate the above revenue increase graph for each service so that we have M_i critical segments for service S_i. We use a counter Pos_i, initiated to 1, to keep track of the current segment for S_i.
- Step 2: Compare all current segments of all services and select those services with the highest price, S_{t_1}, S_{t_2},..., S_{t_k}. Assume the current segments of these services contain D_{t_1}, D_{t_2},...,D_{t_k} shares respectively. If $\Sigma_{j=1}^{j=k} D_{t_j} <= U$, allocate D_{t_1}, D_{t_2},...,D_{t_k} shares for each service respectively. Otherwise, divide U proportionally to D_{t_1}, D_{t_2},...,D_{t_k} and allocate them to the services.

- Step 3: Subtract the allocated shares in this iteration from U. If U is equal to zero, the algorithm stops.
- Step 4: Increase Pos_{t_1}, Pos_{t_2},...,Pos_{t_k} by 1. If not all service's current position Pos_i reaches $M_i + 1$, go to step 2. Otherwise, the algorithm stops.

The *Squeeze* algorithm allocates resources according to the request load. Obviously, if a service does not have any request load, it cannot obtain any shares. It is also obvious that the system can still contain some available resources when the algorithm quits at step 4. We call these available shares as *flexible shares*, which can be used by any service if it experiences sudden request peak load during the next epoch.

From Figure 8, we can also determine how many shares should be reserved for each bitrate of a service, given a certain amount of allocated shares. With the reserved shares for a service, SLASH can perform admission control during the next epoch. The policy of admission control is that a stream with a certain bitrate for one service never uses more than the shares allocated for it plus the current available *flexible shares*. If it uses some amount of the *flexible shares*, the system subtracts the used part from the available amount of the *flexible shares*.

Although the resources allocated in SLASH are shares, our system also controls the bandwidth usages of services. As mentioned before, for a system with β bandwidth units, it can serve β shares continuously in each epoch. Thus, if we allocate D shares for a service, the service should also only use up to D units of bandwidth at any instant of time. However, the bandwidth available when a switch processes requests is also determined by the request load in previous epochs. For example, if a service is initially allocated D shares and there are no requests for that service, it can use that capacity for an arbitrary service. For instance, it can serve D share requests whose lengths are all 2 epochs. In this case, it uses $D/2$ bandwidth units in this epoch. Then, in the next epoch, allocated resources are still D shares. Since available bandwidth is only $D/2$ bandwidth units, the service should only select those stream requests whose length is greater than or equal to 2 epochs to fully utilize the $D/2$ available bandwidth and D allocated shares. Generally, if a service bitrate is provided with D shares in an epoch, the bandwidth used by this bitrate should not exceed D units of bandwidth at any instant of time. If the current available bandwidth is B, SLASH would only admit those requests whose lengths are equal to or larger than D/B epochs. Notice that this D/B threshold is evaluated at any instant of time rather than in the beginning of an epoch since bandwidth is occupied and released continuously.

5 Experiments

We implemented SLASH on Solaris and Linux. Our streaming code is based on the source code publicly provided by Live Networks, Inc. [2]. We added full support for the RTSP protocol and unicast streaming to implement video-on-demand and audio-on-demand (VOD/AOD) services. We implemented the SLASH switch, which can dynamically estimate request load and allocate resources based on the *Squeeze* algorithm. The switch redirects client requests to available servers based on determined resource allocation.

To simplify the model, we only consider a system where each edge server contains a fully replicated copy of all stream files that the system serves. Thus, there are not data placement problems in our study. In our experiment, we use a system that contains 1 switch and 4 servers. Each server's available bandwidth is 90 bandwidth units (each unit is 128Kbps). Thus, the entire system can handle 360 bandwidth units. All system resources are allocated to two services, service 1 and service 2. Each service reserves 180 shares. Each service has only two versions of a single stream, a low-bitrate version and a high-bitrate version, which are the same length, 4 epochs. The low-bitrate version is 1 bandwidth unit. The high bitrate is 3 bandwidth units. The prices for high-bitrate streams are the same for the two services: 12 per epoch. The prices for low-bitrate streams are 9 per epoch. The penalty prices are 4 per share. The system epoch is 10 seconds. In our experiments, the request load is only for high-bitrate versions. Obviously, the system

can only continuously handle 30 requests for high-bitrate streams per epoch, which occupies $30*3*4 = 360$ shares, or 90 low-bitrate streams.

5.1 Resource Reservation

We first illustrate that SLASH can generate smoother throughput and more net profit by reserving a certain amount of resources for each service. In this experiment, we use 4 client machines to emulate 300 clients to access the 4 servers. The request load for service 1 is a constant request load, 15 requests for the high-quality stream per epoch. The request load for service 2 is also constant, 45 requests for the high-quality stream per epoch. If the system does not allocate resources fairly between the two services, a penalty may be charged.

Figure 9 shows the revenue of SLASH versus a First-Come-First-Serve (FCFS) solution. The FCFS solution admits all requests to the cluster and balances the workload among all the servers without any reservation for the next epoch. The FCFS solution cannot downgrade high-quality requests to low-quality requests. However, if the *Squeeze* algorithm downgrades high-quality requests to low-quality requests, it can unfairly obtain more revenue. So, we only provide high-quality streams in this experiment and use the *Squeeze* algorithm only for allocating resources according to the service workload and SLAs without downgrading stream qualities. We can see that the revenue generated by the FCFS solution experiences periodic peaks and valleys because it does not reserve any resources for the next epoch.

Figure 9 does not yet account for penalties. Since the FCFS solution rejects many requests when all resources are consumed, it cannot provide enough shares to the services as reserved by the SLAs. Thus its penalty is higher than SLASH as illustrated in Figure 10. We notice that the revenue generated by SLASH does not decrease significantly when considering the penalty since it reserves resources for each epoch.

Figure 11 demonstrates the resources used by the two services when using the FCFS algorithm to allocate resources. Obviously, the resources used by the two services are proportional to their workloads. Figure 12 demonstrates the resources used by the two services when using the *Squeeze* algorithm. It shows that the resources used by the two services are usually balanced. Although the resources used by service 1 are sometimes lower than its SLA specified resources (180 shares). We find that the reserved resources by SLASH are always strictly 180 shares for each service during the entire experiment, which are calculated by the *Squeeze* algorithm according to the two service workloads and the SLAs. The skewed points on Figure 12 are because the time skew between the SLASH switch and the synthetic workload generator. Although the workload generator generates constant request load of 15 requests per epoch for service 1. Some of them maybe arrive early or late. So the switch accounts them into the previous or the next epoch, which causes that there are not enough requests in some epochs.

As mentioned above, the total request load is 60 requests per epoch. The experiment is conducted for 50 epochs, which introduces 3000 requests in total. The FCFS algorithm accepts 1528 requests, which brings a revenue of 73344. However, it also has a total penalty of 29520. Thus, the net profit is 43824. For the same request load, SLASH only accepts 1480 requests. So the revenue is 71040, which is lower than the FCFS algorithm because it maybe hold resources for future requests without serving requests and the offered load is skewed as mentioned above. But the penalty caused by SLASH is only 960. So the net profit of SLASH is 70080, which is 60% higher than the FCFS algorithm.

5.2 Dynamic Resource Adjustment

In this experiment, we compare the *Squeeze* algorithm with two other solutions. One solution is static reservation, which reserves resources for each service according to SLAs. In this scheme, one service does not use the other service's resources even if it experiences high request load. So this solution avoids paying penalties most of the time, except when the resources used by one service are not released in a timely fashion in following epochs. However, in this experi-

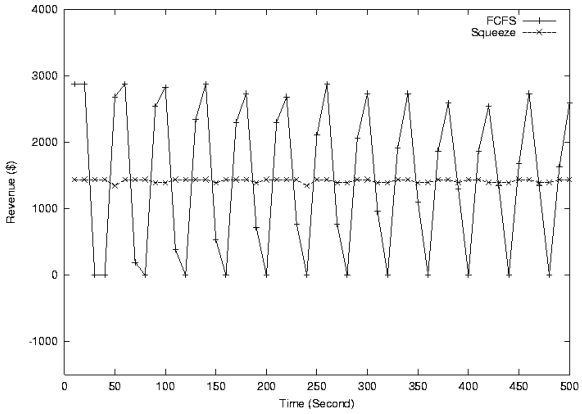

Figure 9: *Without considering the penalty, the* Squeeze *algorithm makes the revenue output smoother. The FCFS solution causes the system throughput periodically to arrive peak and bottom values.*

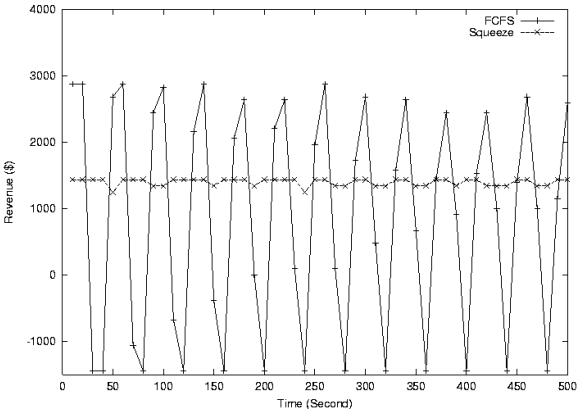

Figure 10: *Considering the penalty, the revenue that the FCFS algorithm can obtain is lower compared to SLASH.*

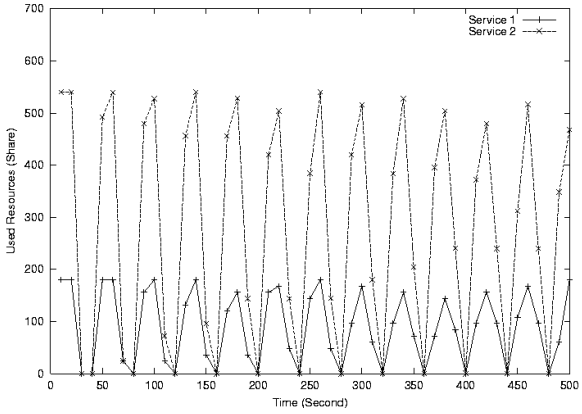

Figure 11: *The number of shares consumed by the two services in each epoch when using the FCFS algorithm.*

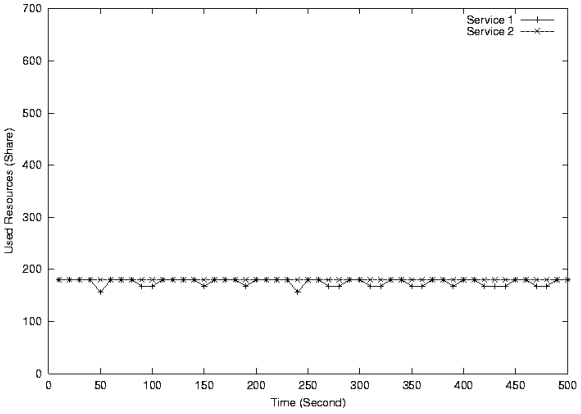

Figure 12: *The number of shares consumed by the two services in each epoch when using the* Squeeze *algorithm.*

ment, the request load is high enough for the two services to fully utilize their resources. The static reservation does not suffer significantly from the inability to adjust resources dynamically. The reason we use this solution is that it does not downgrade stream quality even when the request load is high. Another solution is a greedy algorithm. It always downgrades the quality of streams to the lowest quality to obtain the maximum benefit for each share. Besides this, its resource allocation mechanism is the same as the *Squeeze* algorithm. In other words, it is a special version of the *Squeeze* algorithm, whose revenue increase graph for each service only increases to the first critical point if not considering the penalty point. Thus, this greedy algorithm can also dynamically adjust resource allocation among services according to their revenue prices and request loads.

Figure 13 illustrates the request load between the two services. We intentionally assign a constant request load for service 1, which is 15 requests per epoch. For service 2, the request load is also 15 requests per epoch in the beginning. Then it constantly increases by 10 requests/epoch every 4 epochs. In this experiment, we use 10 machines to emulate 650 clients to access the 4 servers.

As mentioned in Section 1, the major criterion for evaluating different resource allocation schemes is the revenue that hosting services can obtain. Figure 14 shows the revenue generated by each of the 3 solutions without considering any penalty. Static reservation keeps an average 1440 revenue per epoch because it handles 30 high-quality requests per epoch ($12 * 30\,requests * 4\,epochs$). The greedy algorithm downgrades all requests to low quality. However, initially it does not have enough requests to downgrade. So its revenue stays at 1080 ($9*30\,requests*4\,epochs$). As request load increases, it obtains enough requests to downgrade, and the system revenue increases correspondingly until it reaches its peak revenue, 3240 ($9 * 90\,requests * 4\,epochs$). We see that the *Squeeze* algorithm can always maintain the optimal allocation between the other two solutions. Figure 15 shows the revenue with the penalty charge. Obviously, the effect on the static solution is the least since resources are not used by the other services in this solution. The greedy algorithm's revenue decreases greatly in the beginning due to the shortage of deserved resources, which are held by it for possible requests to downgrade.

Figure 16 shows the resource allocation by the *Squeeze* algorithm. We can see that the resources are fully allocated for high-quality streams (180 shares for each service) in the beginning. With the increase of the request load for service 2, more resources of service 2 are allocated for low-quality streams. Correspondingly, the resources for high-quality streams of service 2 decrease. However, the resource allocation for service 1 does not change. When the resources for service 2's low-quality streams reaches 180 shares, meaning that there are no more resources to use for serving service 2's requests. Thus, the system begins to downgrade the requests for service 1 to empty out more space for service 2. This leads to a decrease in the allocation of high-quality streams to service 1. The released resources are used for serving low-quality streams of both service 1 and service 2. Finally, all system resources are allocated for low-quality streams. This figure shows that *Squeeze* is able to allocate resources according to our price and penalty design. So a key point is that the penalty is somehow independent of the quality of the stream.

6 Conclusion and Future Work

In this paper, we discuss principles for the design of SLAs for a distributed streaming hosting system, defining appropriate target resources, revenue prices and penalty prices. We then describe the *Squeeze* algorithm to allocate resources according to specified SLAs. Our price and penalty design enables each service to have a concave revenue increase graph for increased resource allocation. This is a necessary condition for our algorithm to determine the optimal resource allocation for each service. Different from existing gradient-climbing solutions [6], the *Squeeze* algorithm can directly determine a set of critical points to allocate resources instead of increasing and decreasing units of resources among services until equilibrium is achieved. We present some initial experimental results and show that our algorithm operates as expected. The *Squeeze* algorithm can obtain better and smoother revenue than the FCFS solution, showing that resource reservation is critical for streaming services. Meanwhile, the *Squeeze* algorithm can dynamically adjust resource allocation and always maintain optimal revenue compared to static allocation and a greedy allocation algorithm.

Currently, we only consider the case where each edge server has unlimited storage capacity. So there are not data placement problems for this system. However, if the storage capacity of edge servers is limited, when a switch selects a server to serve a request and the server does not contain the requested stream, the switch must consider the cost to retrieve the stream. An intuitive solution is to introduce a cost function for retrieving stream objects. An optimal allocation should consider both the cost and the possible revenue.

Another aspect of future work is coordinating switches to obtain global optimality. Currently, we only define the resource allocation problems on one switch. SLASH is intended to be a scalable system where multiple switches can determine local optimal resource allocation and communicate with each other in an asyn-

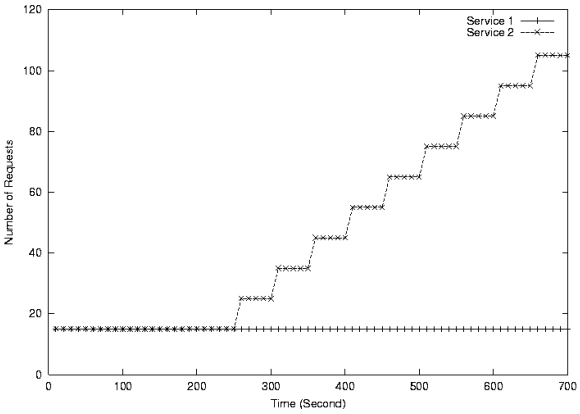

Figure 13: *The request load for the two services. All requests are for high-quality streams (3 bandwidth units).*

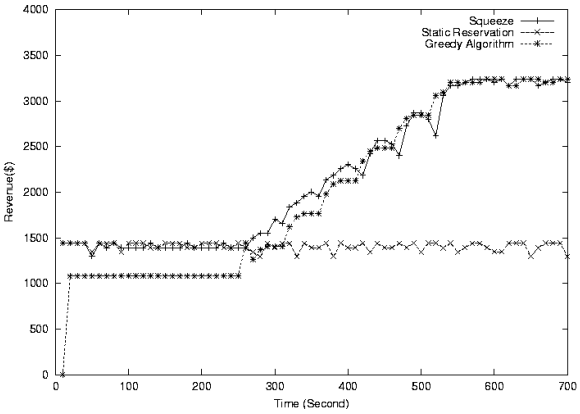

Figure 14: *Without considering the penalty, the revenues generated by the three solutions.*

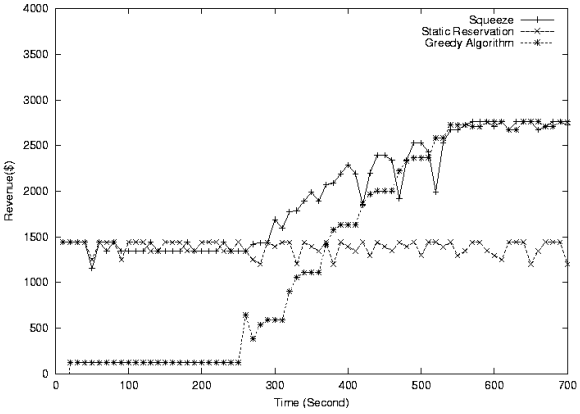

Figure 15: *With the penalty, the revenue generated by the three solutions.*

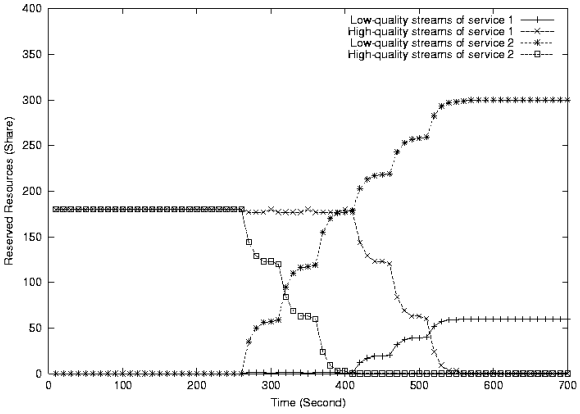

Figure 16: *Resources allocation among different quality streams by the* Squeeze *algorithm.*

chronous manner to maximize global revenue. An extended *Squeeze* algorithm for resource allocation among switches is required. Finally, coordinating data placement among switches is another interesting problem.

Acknowledgments

We thank Rebecca Braynard for her careful review on earlier drafts. The paper also benefited from discussions with Jeffrey Chase, Ludmila Cherkasova, Wenting Tang, Adolfo Rodriguez and Dejan Kostić.

References

[1] Akamai. `http://www.akamai.com`.

[2] Live Networks, Inc. `http://www.live.com`.

[3] Mohit Aron, Peter Druschel, and Willy Zwaenepoel. Cluster Reserves: A Mechanism for Resource Management in Cluster-based Network Servers. In *The Proceedings of the ACM SIGMETRICS Conference*, June 2000.

[4] Gaurav Banga, Peter Druschel, and Jeffrey C. Mogul. Resource container: a new facility for resource management in server systems. In *The Proceedings of the*

3rd USENIX Symposium on Operating Systems Design and Implementation, Feburary 1999.

[5] Rebecca Braynard, Dejan Kostić, Adolfo Rodriguez, Jeffrey Chase, and Amin Vahdat. Opus: an Overlay Peer Utility Service. In *Proceedings of the 5th International Conference on Open Architectures and Network Programming (OPENARCH)*, June 2002.

[6] Jeffrey S. Chase, Darrell Anderson, Prachi Thakar, Amin Vahdat, and Ronald Doyle. Managing Energy and Server Resources in Hosting Centers. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, October 2001.

[7] Toshihide Ibaraki and Naoki Katoh. *Resource Allocation Problems: Algorithm Approaches*. MIT Press, Cambridge, MA, 1988.

[8] Minkyong Kim and Brian Noble. Mobile Network Estimation. In *Proceedings of the 7th ACM Conference on Mobile Computing and Networking*, July 2001.

[9] Balachander Krishnamurthy and Jia Wang. Topology Modeling Via Cluster Graphs. In *Acm Sigcomm Internet Measurement Workshop*, 2001.

[10] James F. Kurose and Rahul Simha. A Microeconomic Approach to Optimal Resource Allocation in Distributed Computer Systems. *IEEE Transaction on Computers*, 38(5):705–717, May 1989.

[11] John Reumann, Ashish Mehra, Kang G. Shin, and Dilip Kandlur. Virtual Services: A New Abstraction for Server Consolidation. In *The Proceedings of the USENIX 2000 Technical Conference*, June 2000.

[12] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson. RTP: A Transport Protocol for Real-Time Applications. `http://www.ietf.org/rfc/rfc1889.txt`, January 1996.

[13] H. Schulzrinne, A. Rao, and R. Lanphier. Real Time Streaming Protocol (RTSP). `http://www.ietf.org/rfc/rfc2326.txt`, April 1998.