To appear in IEEE Infocom 2000.

A Measurement-Based Admission-Controlled Web Server

Kelvin Li Sugih Jamin[†]
Electrical Engineering and Computer Science Department
University of Michigan
Ann Arbor, MI 48109-2122
{nivlek, jamin}@eecs.umich.edu

Abstract- **Current HTTP servers process requests using a first come first serve queuing policy. What this implies is that the web server must process each request as it arrives. The result is that the more requests a client makes, the more replies the server will generate in response. Unfortunately, the bandwidth of the network and the processing capabilities of the server are often limited resulting in an aggressive client, or sets of clients, consuming the majority of the server's resources, limiting other clients' ability to use their fair allocation. While the traditional behavior of a web server works efficiently for a web site that is non-discriminating towards all clients, guaranteeing service for preferred clients from the server itself is not yet possible. This paper describes the algorithm we have designed and implemented on the Apache HTTP server, which has been shown to be effective in allocating configurable fixed percentages of bandwidth across numerous simultaneous clients, independent of the aggressiveness of the clients' requests.**

I. INTRODUCTION

Traditional web servers operate using a first come first serve queuing policy. As each client makes a request, the server processes the request, generates a reply, then accepts the next request for processing. The more requests a client produces for the server, the more responses the server will generate to match. If we define the server's resources by its limiting hardware capabilities, specifically server network bandwidth and server processing cycles, then over a fixed interval of time, the client will consume an amount of the server's resources proportional to the number of requests it makes. While this mechanism is efficient for serving clients at low server utilization, when server load increases to saturation, some clients may receive poor service or no service at all. Poor service is defined as being a state when a high response time to required resource ratio exists. While this may still not be of concern to non-discriminating administrators of web service, this problem of indiscriminate service is of great interest to service providers who have guaranteed resources to certain clients of preferred status.

Web farms, collections of web servers maintained by a common administrator hosting one or more web sites, are becoming increasingly popular to establish because they allow individuals and organizations to outsource the administration of their web sites. In order to provide good service to such a wide variety of clients, it is important to the administrators that each client can depend on having bandwidth guarantees, perhaps proportional to an amount they are willing to buy from the web farmer. An admission-controlled web server would be required here to keep an unexpectedly popular web site from suppressing the service from other sites in the farm.

Because of the random nature of web and network traffic, a web server cannot be expected to accurately regulate its own utilization without the ability to determine its own bandwidth consumption. Thus, parameter-based admission-controlled web servers such as [1] cannot adapt at a fine enough granularity to allow the server to be optimally utilized at all times. It is when resources become scarce that the fair service of clients is placed in jeopardy. Our measurement-based admission-control algorithms were specifically designed to regulate the usage of resources at this point.

What this paper describes is the algorithm that we have developed to allow the web administrator to guarantee resources to preferred clients in spite of the degree of competition for those available. These clients that are requesting more resources than they are entitled to are deemed aggressive and regulated until they return to a more passive state.

II. HTTPD BACKGROUND

The difficulty in effectively managing the server's r e-sponse to client requests is a result of the server's archite c-ture. Briefly, HTTP servers are daemon processes that listen for client requests through connections on a particular preferred socket port. When the client makes a connection, the server attaches a child process to it by either spawning one anew or awakening a dormant one. The request is then passed to that child for processing. Although at any particular point in time multiple child processes may be concurrently active, the requests are still funneled through the daemon process which is restricted by the listen queue's first in first out (FIFO) queuing policy. The listen queue is a buffer that places the client on hold, while the server passes its connection to one of its child processes. This keeps a con-

[†] Sugih Jamin's research is supported in part by the NSF CAREER Award ANI-9734145 and the Presidential Early Career Award for Scientists and Engineers (PECASE) 1998. Additional funding is provided by MCI WorldCom, Lucent Bell-Labs, and Fujitsu Laboratories America, and by equipment grants from Sun Microsystems Inc. and Compaq Corp.

nection request from being lost if the server cannot respond quickly enough.

The length of this listen queue can be specified during the `listen` socket API call when the server is started. The length of the listen queue is very important. A short listen queue may result in lost connection attempts. As a result, an aggressive client will have a higher probability of filling the queue as soon as a position becomes available. A long listen queue may contain stale entries, which are requests no longer useful for processing since replies beyond a client specified timeout period are ignored. In addition, when a long listen queue becomes filled with entries, request latencies tend to accrue.

One of the requirements behind a web server that can deal with aggressive clients is that the preferred clients' r e-quests must be capable of being placed on the listen queue. It is important for an algorithm which does admission control to be able to remove requests off the listen queue as quickly as possible, especially if the requests will be ignored or delayed, so that preferred clients can be admitted and processed as soon as possible. While our algorithm does not use a separate process to clean out the listen queue to set aside preferred clients for pre-admission, its behavior adheres to this necessity by deferring non-preferred clients so that CPU time can be used to remove the next request.

III. Measuring Bandwidth

The perception of bandwidth can be understood from different aspects. First, there is the network bandwidth, which is the actual number of bits transferred within a finite amount of time. This is a fixed upper limit of the physical network and interfaces. To measure this quantity of bits/second from the server is a difficult if not an impossible task. To do so requires a timer to be set immediately before the data is transmitted and the measurement to be completed as soon as the transmission has ceased. With the myriad of system calls necessary to send a stream of data out of the server, actual network bandwidth can not be precisely measured because times recorded may be obfuscated by function call overhead and process preemption. Because multiple processes and threads need to share the network interface, measuring actual network bandwidth from the server meets many challenges. As we will discuss, actual network bandwidth needs only to be estimated.

The perception of bandwidth can also be understood from the perspective of the client. This is essentially the response time of the request. When users complain about a slow network, it is actually a misnomer for a busy network. Essentially, the client or server needs to wait for the network to become free before additional transmissions can be made. This perceived bandwidth measure can still be quantitatively measured by the following formula:

$$Perceived\ Bandwidth = \frac{Bytes\,Transferred}{Response\,Time} \qquad (1)$$

The perceived bandwidth can be accurately measured by the clients and server. From the server, a timer can be loosely set sometime before the transmission of data, although necessarily after admission. The response time can then be measured after the data has been sent. (This assumes synchronous/blocking sends.) Overhead for processing the request, such as reading the file from disk, and pre- and post-processing of requests is included in the response time. This overhead is independent of file size, and so it necessarily reduces the perceived bandwidth of the request by a constant amount. Perceive bandwidth is therefore not a linear function of file size. This implies that measuring network bandwidth by itself is not an accurate measure from which to determine server utilization. Clients that request several small files have a lower perceived bandwidth utilization measurement than clients who request a few large files although their total network utilization is equal or less than the latter. The result is that clients with smaller requested files will receive preferential treatment. As long as the time for data transfer dominates the perceived response time, the perceived bandwidth is an accurate enough measure for measurement-based admission-control. Unfortunately, this is not the case given current server technologies. Processing time is a significant portion of response time at common file sizes.

Because the processing time cannot be distinguished from data transfer time, the client application will only be able to present the data to the user at the rate of the perceived bandwidth. What this allows the server to do is defer the transmission of data to the client on a per request basis so that the client will perceive diminished bandwidth. Without completely starving the aggressive client and causing the connection to be timed out, aggressive clients can still be serviced with intermittent delays proportional to their over-activity, while more passive clients are still serviced inside the aggressive client's delay period.

To deal with the natural burstiness of network traffic, the server keeps track of the current average bandwidth by using exponential averaging:

$$avg\,bw = [new\,bw \cdot a] + [old\,bw \cdot (1 - a)] \qquad (2)$$

Here, α is a constant fractional value between 0.0 and 1.0. The larger the value of α, the more quickly the average bandwidth is adjusted to represent the current state of the server's utilization. With values of α too large, instantaneous values of average bandwidth are not representative of true network utilization. Contrariwise, as values of α become too small, average bandwidth measurements adjust to their actual value too slowly resulting in poor utilization when the server cannot react to transient windows of free bandwidth. Values of α that worked well ranged from 0.05 to 0.30.

Our initial trials to determine how similar perceived bandwidth from the server would be to the client led us to develop a simple bandwidth measurement tool named `bw_tester`. This program emulated client requests by repeatedly placing HTTP requests in series. To simulate *N* concurrent clients, an instance of `bw_tester` was simultaneously started on *N* separate machines. Bandwidth measurements commenced a few seconds after requests were initiated, and measurements ceased a few moments before the last request. This ensured that bandwidth measurements were only made when all *N* clients were concurrently active. Each `bw_tester` instance made requests sequentially. As soon as a request was completed, the next connection was immediately made for the new request. The results of the bandwidth measurements agreed with those derived from the server. As we continued experimenting by considering trials with various file lengths, we found that the perceived bandwidth was dependent on the file size. The observations suggested that as the file size decreased, the measured perceived bandwidth would also diminish. This expected phenomenon was a result of the increased proportion of time spent on a request as a result of constant processing overhead. Fig. 1 illustrates the outcome from this study.

Determining the utilization of a web server based solely on the bandwidth measurement may not be entirely accurate. A web server can have very low bandwidth utilization and still be fully saturated or overloaded if the requested file sizes are small. Fortunately, most files that are requested on the web, have a file size between 100 and 100,000 bytes [2].

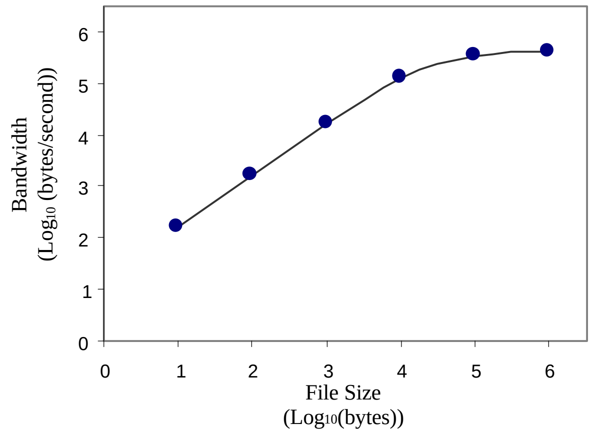

Fig. 1. File Size versus Maximum Measured Bandwidth

IV. Admission Perception

Since the web server must examine all client requests in order to establish their source, the concept of admission is really a decision as to whether the requests will be serviced by one of the child processes or not. The sooner this decision is made the more efficiently the server will operate.

A properly working algorithm will reject requests from a client when it has both received more than its allocation of bandwidth and when the server is fully utilized. This means that if the server is not saturated with requests, any client may take advantage of the unused resources.

When the network or server is busy, users can perceive poor service a number of ways.

1. The user receives a message from the client application explaining that the server is not responding.
2. The server has stalled in the middle of a transmission.
3. The requests' data arrives back to the client at a very slow pace.

The least disquieting of the above three scenarios is the third. If the information from the server is critical, then it is better to receive data slowly, then not at all. The first and second cases waste bandwidth because the user may try to reload by placing the request again. When another request is waiting in the listen queue, the server will consume processing time trying to examine the request again for admission determination. If the request is made by an automated information retrieval service, then a user will not be present to observe the first two most disquieting scenarios, which tend to reduce server utilization by influencing user morale. Any automated system that uses brute force repetition to attain services will cause the servers to react wastefully. This is why scenario 3 is the most favorable method of achieving control over server utilization. Here, the automated system will simply have to wait for a reply like the other clients.

From our initial bandwidth measurement trials we discovered that the amount of overhead required to process a request is a very significant portion of the response time. If processing time is to be spent, it might as well not be in vain if the client can be satiated with the delivery of data. By this principle, the user will perceive the admission control of our algorithm by the third scenario. If the user appears to the server to be an overly aggressive client, then the second scenario will be observed. In the worst case, as with all web servers, the first scenario will occur when the listen queue is full.

The configuration for admission control is specified at the time of server startup. The first configuration parameter is the IP address or subnet of the machine or set of machines, along with the allocation percentage. Rejecting admission for all groups or for a particular machine is also possible; the allocation percentage would be set to zero. To specify admission constraints is to specify the percentage of perceived bandwidth that each client should receive. Configuration for IP address or subnet can be done with any pattern matching language since the IP address of the sending request is merely a string.

The total bandwidth also needs to be specified. This is a fixed value that represents the maximum bandwidth that should be available to all clients. Each client can be guaranteed a percentage of this maximum bandwidth when specified. There are many benefits of specifying a maximum bandwidth rather than allowing the server to serve to capacity. Because a fully-loaded server may be less efficient and reliable, setting the total available bandwidth to a value less than the actual maximum bandwidth allows the server to shed load before it loses efficiency. In addition, multiple web servers may be placed on the same local area network where their network bandwidth will need to be shared.

V. Algorithms

The measurement-based admission-control algorithm is easy to implement. Only a few blocks of new code need to be added to the existing server code. Working with web server code is analogous to working with any concurrent-processing paradigm. The primary daemon process is responsible for preparing shared-memory and other one-time initialization routines. Each child process working concurrently has a distributed algorithm that contributes to the overall behavior of the server. A managing process ensures that each of the parallel child processes conform to their correct behavior by overseeing the accuracy of variables necessary for inter-child communication.

A. Primary Daemon Process

In addition to the original duties, the primary daemon process allocates and initializes the shared memory and prepares the required semaphores. The configuration files, which specify client bandwidth allocation and the total bandwidth available to all clients, are read into shared memory. The Bandwidth Manager process is also spawned at this time.

B. Bandwidth Manager Process

The Bandwidth Manager process is a new process that is responsible for maintaining the timeliness of the information stored in the shared memory. As will be discussed, the bandwidth information is updated each time a request is made. If no requests have been made, the current bandwidth utilized by the client should be properly updated. The Bandwidth Manager periodically reduces the average bandwidth of each client if the client has not triggered an update after a specified amount of time. This is also done by exponentially averaging zero into the current average bandwidth of the client. The α value would not necessarily be the same as that used to update the average bandwidth during request activity. In this case, α would be inversely proportional to the frequency that the Bandwidth Manager is active.

By tallying up the individual average bandwidths of each client, the Bandwidth Manager determines the maximum measured bandwidth of the server at each processing interval. The importance of this value in predicting admission control will be discussed in the child process algorithm description.

Since the Bandwidth Manager is constantly active, allowing it to update the shared memory too frequently will cause the server to waste processing time that could be spent servicing requests. The server may also not be optimal if the Bandwidth Manager is not activated often enough to ensure the bandwidth measurements accurately reflect the amount actually available. Good results were yielded with an activation frequency of once every half to one second.

C. Child Processes

Since admission control is done on a per request basis, each child process is equipped with the algorithms necessary to determine whether to service the request or not.

A very convenient granularity at which to control overall utilization would be per request because the expected processing time of the child processes next action can be established based on the requested file's size alone. Since a file must be transmitted in its entirety to be of value to the client, finer grain admission control would result in fragmented and incompletely sent files as a result of transmission timeouts. At a higher granularity and without the foreknowledge of the extent of processing required by the server's next action, regulating bandwidth would be based on past bandwidth measurements. Because the interval length between successive requests may not necessarily follow a well-understood probability distribution, future bandwidth utilization cannot be accurately conditioned based on utilization history alone. As a result, per request admission control appears to be the best strategy to ensure both accurate regulation of bandwidth and the least likelihood of wasteful usage of critical resources.

A request can be found in one of three stages in the child process:

Processing. The processing stage occurs when the request is being processed. Processing is required for a request to be serviced to completion. This includes reading the file requested from disk, and sending the data over the network to the client. Any number of child processes can be in the processing stage depending on the capabilities of the server. Since it is likely that several child processes are in the process stage simultaneously and are therefore updating the bandwidth measurement variables, a semaphore is used to protect the shared memory. Child processes wait by sleeping a random amount of time until the semaphore is released. Any request that is in this stage is guaranteed to complete barring any processing errors.

TABLE I
LEGEND OF NOTATIONS USED

Symbol	Definition
b_i	Allocated Bandwidth of Client *i*
$\hat{b}_i$	Average Bandwidth of Client *i*
d_i	Available Bandwidth to Client *i*
$\hat{A}$	Total Bandwidth Allocated to All Clients
$\hat{\hat{A}}$	Maximum Measured Bandwidth

Delayed. A request is in the delayed stage when a client is requesting more data than it has been allocated and when the server is fully utilized. A child process is delayed by using the `sleep` function call. If a request is admitted for processing, a delay must be calculated so that the flow of requests can be controlled. Calculated per request, the delay is based on the following formula, where *i* refers to the client's identification from which the request originated:

$$delay_i = \left(\frac{1}{d_i} - \frac{1}{\hat{\hat{A}}} \right) \times \textit{file size} \tag{3}$$

Each child process calculates the delay to be injected $delay_i$ accordingly given the bandwidth available to the client *i*, d_i, the server's maximum measured bandwidth $\hat{\hat{A}}$, and the *file size* of the request. Notice that the delay is proportional to the file size of the request, and inversely proportional to the bandwidth available d_i to the client. If the delay is negative, there is excess available bandwidth to this client for the request, and so the delay is set to zero because the request can be processed immediately. The purpose of injecting such a precise amount of delay into the response time is to force the bandwidth measurement of the reply to be that of the target bandwidth. The target bandwidth is all the bandwidth that is available when the server is not fully utilized, and only the client's allocated bandwidth otherwise.

The delay formula is quite simple to derive. If one expands the equation by multiplying the inverse bandwidth terms by the file size, then the delay becomes the difference between the time allowed to process the file and how quickly the file is expected to be transferred if there is no delay.

The requested file size can be found using the `fstat` function call on the file that was specified in the client's request. If a web server has a small subset of commonly used files, then the client can cache these file sizes outside of shared memory. Files such as `index.html` and frequently downloaded graphic files would be prime examples of frequently accessed files whose statistics could be cached for efficiency.

The maximum measured bandwidth $\hat{\hat{A}}$ is maintained by the Bandwidth Manager process. Since the instantaneous utilized bandwidth is calculated periodically by summing up the average bandwidths $\hat{b}_i$ across each client's share, every time the instantaneous utilized bandwidth exceeds the previous value of the maximum bandwidth $\hat{\hat{A}}$, this variable is updated. Since the $\hat{\hat{A}}$ cannot be accurately determined until the server has been active for sometime, the initial value is set to the total bandwidth $\hat{A}$ that was specified at startup in the configuration file. While the maximum measured available bandwidth $\hat{\hat{A}}$ is approaching its actual value, the delay calculations will be inaccurate. This is actually a self-correcting problem because the server adapts so well. If the $\hat{\hat{A}}$ is too low, since it is set to $\hat{A}$, then the delay injected into the processing time will be too low. This causes each client's utilized bandwidth $\hat{b}_i$ to be greater then its allocation, and thus causes the server's instantaneously measured bandwidth to exceed $\hat{\hat{A}}$. When the Bandwidth Manager process becomes aware of this, $\hat{\hat{A}}$ is updated with a more accurate value, causing the next series of delay calculations to be increased.

The calculation of d_i is a delicate matter to ensure that clients are receiving their fair share of allocated bandwidth while still maximizing the server's utilization. The goal is to achieve the kind of bandwidth distribution accuracy that is similar to the results of a weighted fair queuing (WFQ) algorithm [3]. Unfortunately, WFQ is not the best queuing model for a web server. A round robin approach can be too inefficient to implement because segmenting a request into packets involves a fragmentation cost. Furthermore, a web server needs to be able to discard requests even though the queue of requests may not be full.

To control the flow of bandwidth, the server must establish what state it is currently in to decide which client to limit and how much. The server can be in two different states, which directly affects how much available bandwidth should be given to each client. The server can either be fully saturated, or partially saturated. If the server is idle, it is considered to be in a partially saturated state. If a server is partially saturated, each client *must* be receiving its required bandwidth. Moreover, if a client needs more bandwidth than its allocation, it will receive it if the server is partially saturated. When a server is fully saturated, the scenario is more difficult and it becomes very important to determine which client or clients are receiving a proportion of the bandwidth greater than their allocation. This misdirected bandwidth should be delivered to the client or clients that are not receiving their minimal share.

As mentioned before, if the server is partially saturated, then all clients must be receiving the amount of bandwidth they can utilize. Nothing can be said about the proper allocation of bandwidth if the server is fully saturated. A server can be fully saturated, have disproportionate allocation, and still be in a correct state if one of the clients does not need all of the bandwidth that it has been allocated. In this case, all the clients are still satisfied.

Let the state of unsatisfaction be deemed when there exists any client that could use more bandwidth than it has so far utilized but it is not receiving at least its allocation because of suppression by any one of the other clients. This state of unsatisfaction can be determined when two conditions exist for any client. These two conditions are 1) when the client's average bandwidth is less than its target bandwidth and 2) the client is active.

$$unsatisfaction = \bigcup_{j=1}^{N} \left[\left(\hat{b}_j < b_j \right) \cap active_j \right] \tag{4}$$

The determination of whether a client is currently active is made with a boolean flag. Each time a child process places a client's request in the processing stage, the active flag is updated to `on`. The Bandwidth Manager process periodically flips this active flag to `off`. The reason the child process does not flip the flag to `off` by itself at the end of processing its request is because there is simply not enough time for the other child processes to read the value when determining the unsatisfaction condition. It was observed that the child processes were not being preempted by each other often enough for the activity values to be read at `on` even though the clients were very active. When a client is not fully utilizing its allocated bandwidth, the active flag tends to remain in the `off` state.

When the unsatisfaction condition has been established, the process handling the client's request can determine whether to relinquish bandwidth or try to attain more. Recall, to increase or decrease the amount of bandwidth given to a client is a matter of controlling how much delay to inject into the processing time of a request. The delay calculation depends fundamentally on the available bandwidth, if the file size to be sent is fixed.

The available bandwidth is calculated dependent on the unsatisfaction condition. If the unsatisfaction condition exists, then the available bandwidth is given by:

$$avb\,bw_i = \begin{cases} 2 \cdot b_i - \hat{b}_i & \text{if } \hat{b}_i < b_i \\ b_i & \text{if } \hat{b}_i \geq b_i \end{cases} \tag{5}$$

Notice that if any client is not satisfied, then the child process of the aggressive client will force its available bandwidth down to its allocated bandwidth. What this ensures is that every client will always have at least its allocated bandwidth. If a client's average bandwidth is less than its allocation, then the client needs to be more aggressive at reaching its target. A formula that worked well was twice the target bandwidth less the bandwidth already attained. Since more than one client may be unsatisfied, the amount of available bandwidth each client can be allowed should be relative to the amount of bandwidth that it should be receiving, less the amount that it has already attained.

When the unsatisfaction condition does not exist, it is only important to maintain this favored condition and allow the server to be fully utilized. What this involves is ensuring that no client loses bandwidth to another client while still allowing clients that could use more bandwidth to attain more.

To determine how much bandwidth is still available, the child process must sum up the average bandwidth across all the clients; This is the bandwidth utilized. Let *N* be the number of clients that were allocated separate amounts of bandwidth. The variable *utilized* is defined by the value below:

$$utilized = \sum_{j=1}^{N} \hat{b}_j \tag{6}$$

The amount of unused bandwith can be calculated with the following formula, where the total bandwidth is the amount specified in the configuration file.

$$remaining\,bw = \hat{A} - utilized \tag{7}$$

Once the remaining bandwidth had been determined, the available bandwidth can be calculated based on this value. If any bandwidth remains, the client is allowed the total bandwidth less the amount utilized by other clients. If the remaining bandwidth approaches zero, then that client will receive its allocated share. This guarantees that the client will receive at least its allocated amount.

$$avb\,bw_i = \begin{cases} B - (utilized - \hat{b}_i) & \text{if } remaining\,bw > 0 \\ b_i & \text{if } remaining\,bw \leq 0 \end{cases} \tag{8}$$

If any of these available bandwidth calculations result in values less than zero, the determined available bandwidth is set to zero. This is a safeguard against temporary spikes in average bandwidth that may result in inaccurate utilized bandwidth calculations.

$$avb\,bw_i = \max(avb\,bw_i, 0) \tag{9}$$

A user will perceive a slow connection if the request is delayed in the processing stage.

Standby/Rejection. A process is in standby if the request has not yet been processed and is waiting for the bandwidth semaphore to be released. If a process arrives at this stage and there are an excess number of processes concurrently sleeping (a run-time upper limit constant), then the newly arrived process is immediately killed. This is necessary for several reasons. As server load increases, the number of sleeping child processes associated with a client increases due to the delay stage. Removing the processes of aggressive clients frees the server to service other clients more effectively. Furthermore, when a client has too many of its processes in the delayed stage, the calculated delay may exceed the time out period of the client, and the response will be ignored anyway. A request that is rejected in this stage will be perceived by the user as a stalled request.

VI. Experimental Evaluation

Testing was performed within the confines of the University of Michigan's eecs.umich.edu network. The purpose of minimizing the distance between clients and server was to reduce the unpredictability of request and reply times due to external traffic and congestion. A variety of tests were performed with various maximum bandwidth specifications and client bandwidth allocations. Most tests resulted in the accurate dissemination of bandwidth according to allocation.

The following configuration and result exemplifies the accuracy of the admission control and bandwidth allocation algorithms:

TABLE II
Web Server Bandwidth Configuration Parameters

Client	Allocation (%)
A	10
B	20
C	30
D	40

Total Bandwidth	100 kilobytes

Here in Table II we have specified four clients ($N = 4$), labeled A through D, with allocations of total bandwidth 10 through 40 percent, respectively. The total bandwidth was set to 100 kilobytes per second, a value less than the maximum bandwidth.

To test the web server for the key characteristics in our measurement-based admission-controlled web server, it was necessary to prove that 1) bandwidth was optimally utilized, 2) when a client makes requests, it is given at least its minimum allocation, and 3) the server restricts total bandwidth utilization across all clients to that specified in the configuration file. There is nothing implied about distributing excess bandwidth proportionally or evenly to requesting clients.

Client requests were generated using `httperf`, a web server performance measurement tool [4]. One of `httperf`'s important characteristics is its ability to gene rate and sustain server load. To use `httperf`, it is necessary to specify a small number of request parameters, such as hostname, port, and URI, along with behavioral parameters such as request rate and the number of connections that should be made in that trial. Instead of making connections sequentially, one after the completion of another, `httperf` forks a requesting process periodically, with an interval length inversely proportional to the request rate. Each time a request is made, another port is selected for the new connection. This is how a high degree of server utilization is sustained.

To enumerate the various combinations of simultaneous client loads, clients A, B, C, and D executed a batch file from the UNIX shell, which executed `httperf` and `sleep`, when activity and idleness were specified, respectively. The client activity was directed by the values in Table III. Each experimental combination epoch lasted approximately two minutes. During periods of activity, 10 KB files were requested at a rate of 6 requests per second per client.

TABLE III
Client Activity Schedule

Epoch	Client A	Client B	Client C	Client D
1-2	*active*	*active*	*active*	*active*
3	*idle*	*active*	*active*	*active*
4	*active*	*idle*	*active*	*active*
5	*idle*	*idle*	*active*	*active*
6	*active*	*active*	*idle*	*active*
7	*idle*	*active*	*idle*	*active*
8	*active*	*idle*	*idle*	*active*
9	*idle*	*idle*	*idle*	*active*
10	*active*	*active*	*active*	*idle*
11	*idle*	*active*	*active*	*idle*
12	*active*	*idle*	*active*	*idle*
13	*idle*	*idle*	*active*	*idle*
14	*active*	*active*	*idle*	*idle*
15	*idle*	*active*	*idle*	*idle*
16	*active*	*idle*	*idle*	*idle*
17-18	*active*	*active*	*active*	*active*

Fig. 2 represents the bandwidth that was allocated to each client over time. For example, client A colored by the lightest gray, who was allocated 10% of the total bandwidth by the schedule in Table II, is shown to receive approximately 10 KB/s between times 0 to 240 seconds. At 240 seconds, client A begins to lose bandwidth because it has entered its third epoch, a moment of idleness as specified in Table III. According to Table III, client A will fluctuate between idleness and activity at alternating epochs, which can be seen in the graph as periods of receiving at least its allocated 10% or no bandwidth at all. All the other three clients follow the same behavior.

As Fig. 2 indicates, each client is capable of receiving the bandwidth allocated when it is requesting it. The server is also successful in distributing bandwidth such that its total bandwidth falls very close to the range as specified in the total bandwidth configuration file as noted in Table II. Notice that in the first two epochs 1-2, client D is actually utilizing a little more than its allocated 40%, but after $\hat{\hat{A}}$, the maximum measured bandwidth, comes closer to its actual value, in the final epochs 17-18, these values are even closer to their allocation.

TABLE IV
ERROR ANALYSIS

Allocation (%)	Actual Allocation (%)	Error (%)
10	9.48	0.52
20	19.05	0.95
30	29.90	0.10
40	41.56	1.56
Mean Squared Error		**0.0190**

The statistics displayed in Table IV were gathered from approximately 150 data points sampled during the time period between epochs 17-18. According to the allocation error table, the algorithm is quite successful in achieving the requirement specified by the configuration files with a Mean Squared Error (MSE) of 0.0190. MSE is a commonly used metric to quantify the difference between a value from an experiment and its target. The MSE calculation in Table IV was performed with the following formula:

$$MSE = \sqrt{\sum_{j=1}^{N} \left(Target\ Allocation_j - Actual\ Allocation_j\right)^2} \quad (10)$$

Fig. 3 displays the total and maximum measured bandwidth utilization over time graph. As can be observed, utilization is always very close to *B*, the allocation specified in the bandwidth configuration file. The utilized bandwidth peaks above 100 KB sporadically, but the server quickly adjusts the delay to inject into the response to accommodate this. Notice that as the server comes to appreciate the actual maximum bandwidth, the bandwidth allocations become more accurate.

Recall from Section III that a connection overhead exists per request. If the size of files that are requested from the server are consistently less than 10 KB, the average bandwidth actually received per client may not be as allocated and MSE values will increase.

VII. FUTURE WORK

While the algorithms presented in this paper regulate bandwidth based on the knowledge of file size, the importance of dynamically generated web pages should not be downplayed. Additional work needs to be done to determine how well these algorithms will perform when processing time consists of a much greater proportion of time than does that required for data transfer over the network. Algorithms can be developed that can better differentiate between transmission time and processing time but additional research needs to be made as to how to weight each of these measurements when allocating resources. What we have developed here is a basic framework with algorithms and formulas that achieve the basic characteristics necessary for server-based control over accurate bandwidth allocation and optimal utilization.

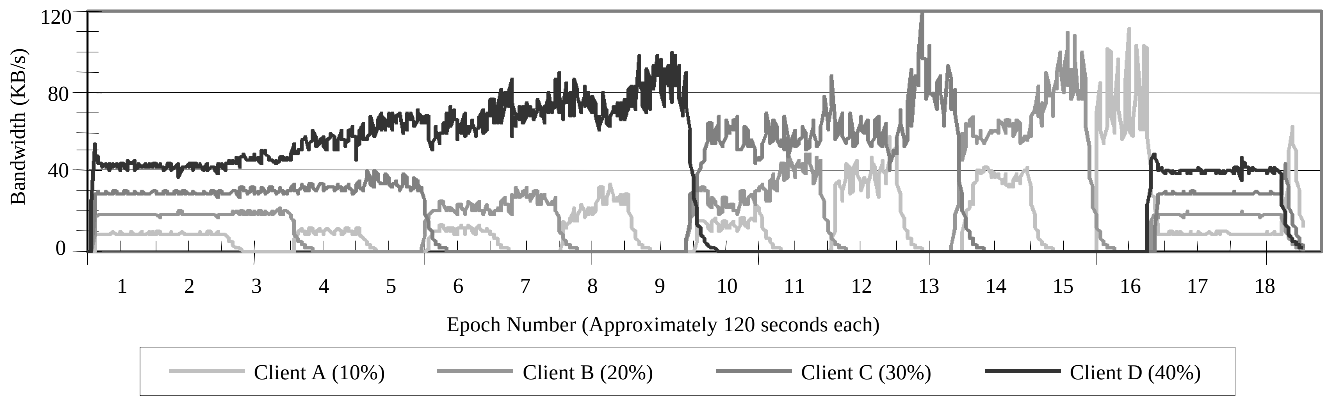

Fig. 2. Bandwidth Allocation Over Time

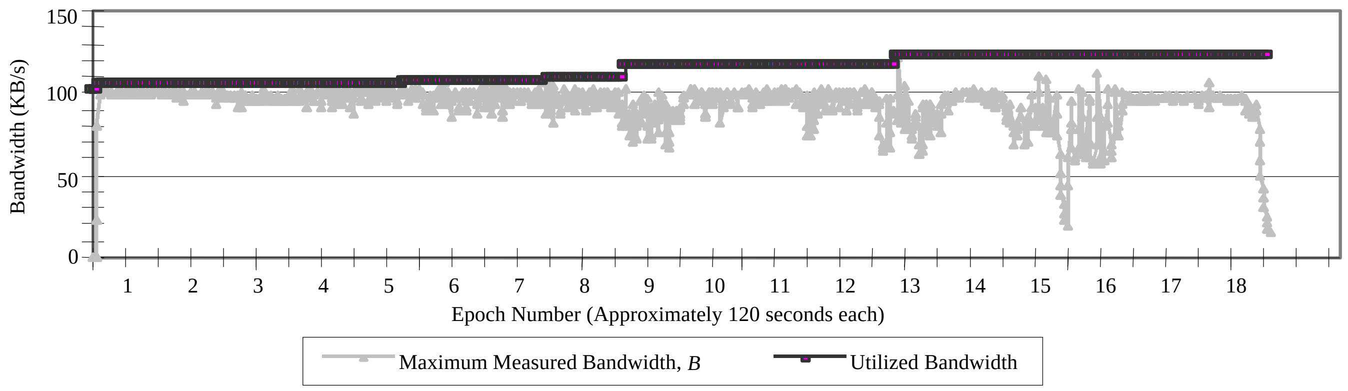

Fig. 3. Measured Total and Maximum Bandwidth

VIII. SUMMARY

With the increasing popularity of the web and the limited capacity of current network and processing technologies, it becomes ever more important to control how these critical resources are utilized. Achieving optimality while not satisfying any guarantees of availability cannot be an answer to these problems. What we have provided in our algorithms for implementing a measurement-based admission-controlled web server is the ability to accurately control how bandwidth is distributed across various clients of unequal requirements. What we have shown is that measurement-based admission-control is possible without extensive trial and error-parameterized configuration.

ACKNOWLEDGMENT

We thank Yilun Shang for the initial implementation of the bandwidth measurement and admission control code.

REFERENCES

[1] L. Cherkasova and P. Phaal, “Session based admission control: a mechanism for improving the performance of an overloaded web server,” Hewlett Packard, Palo Alto, CA.

[2] M.F. Arlitt and C.L. Williamson, “Web server workload characterization: the search for invariants,” *Proceedings of the 1996 ACM SIGMETRICS Conference on the Measurement and Modeling of Computer Systems*, Philadelphia, PA, ACM. 1996.

[3] A. Demers, S. Keshav, and S. Shenker, “Analysis and simulation of a fair queuing algorithm,” *Journal of Internetworking Research and Experimence*, October 1990.

[4] D. Mosberger and T. Jin , “httperf—a tool for measuring web server performance,” Hewlett Packard, Palo Alto, CA.

[5] A. Feldmann, R. C壙eres, F. Douglis, G. Glass, and M. Rabinovich, “Performance of web proxy caching in heter o-geneous bandwidth environments,” *Proc. IEEE Infocom ’99*, March 1999.

[6] B. Laurie, P. Laurie, and R.J. Denn, *Apache: The Definitive Guide*. O'Reilley & Associates, 1997.

[7] W.R. Stevens, *Advanced Programming in the UNIX Environment*. Addison-Wesley, 1993.

[8] W.R. Stevens, *Unix Network Programming*. Prentice Hall, 1997.