

# Admission Control and Scheduling for High-Performance WWW Servers

Azer Bestavros, Naomi Katagai, and Jorge M. Londoño

Computer Science Department  
Boston University

*August 1997  
(Revised May 1998)*

---

## Abstract:

In this paper we examine a number of admission control and scheduling protocols for high-performance web servers based on a 2-phase policy for serving HTTP requests. The first "registration" phase involves establishing the TCP connection for the HTTP request and parsing/interpreting its arguments, whereas the second "service" phase involves the service/transmission of data in response to the HTTP request. By introducing a delay between these two phases, we show that the performance of a web server could be potentially improved through the adoption of a number of scheduling policies that optimize the utilization of various system components (e.g. memory cache and I/O). In addition, to its premise for improving the performance of a single web server, the delineation between the registration and service phases of an HTTP request may be useful for load balancing purposes on clusters of web servers. We are investigating the use of such a mechanism as part of the Commonwealth testbed being developed at Boston University.

---

## Introduction

With the wide-spread popularity of the World Wide Web, the issue of network and server load capacity becomes increasingly important, and must be addressed. As the availability of information and services on the Web increases dramatically, so does the number of global online users. WWW traffic already represents the busiest segment of the entire Internet today and continues to expand at a rapid rate. Web servers have the burden of accommodating this sudden growth, prompting the demand for more efficient and adaptable server systems. This, in turn, directs attention to the scalability of a high performance web server in order to continually meet the demands of increased network traffic.

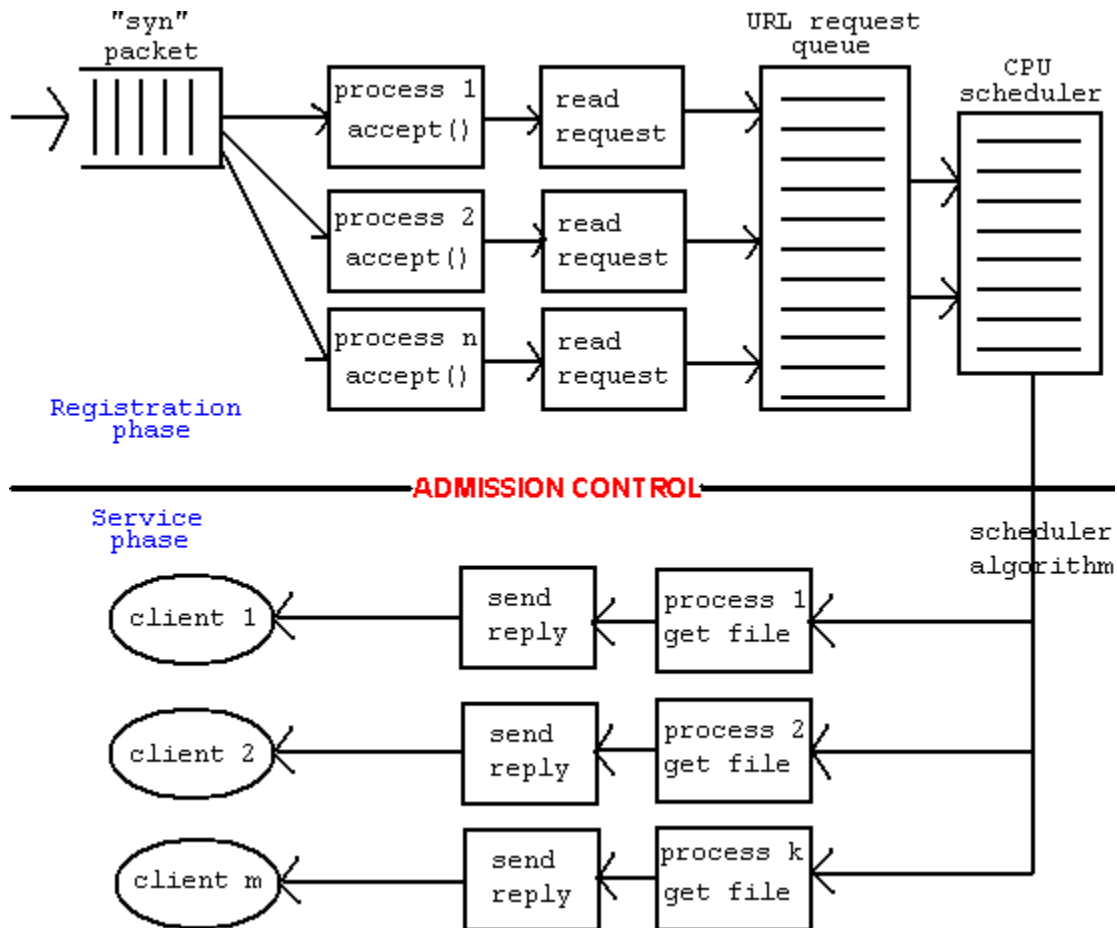
When a web server becomes heavily loaded, system overhead increases due to the lack of available resources, which may cause the system to thrash. Recent studies have shown that for every system, there is a maximum-number-of-processes that defines the thin line between system thrashing and maximum optimization of system resources. Today's web servers, such as Apache, restrict the number of processes that are running concurrently at any given moment to prevent this from occurring. However, it may be beneficial to insert a mechanism which regulates (HTTP requests) access to system resources before a request can be completed. By creating a more controlled environment, system overhead can be monitored to a greater degree, and thrashing can be prevented. The addition of such a design can also be useful for scheduling purposes as an alternative to the more conventional first-come-first-served policy for scheduling requests.

In this paper, I will examine the possibility of adding a number of admission control and scheduling

protocols to increase efficiency and decrease latency on a typical web server.

## The Admission Control and Scheduling Protocol

The main idea of the admission control and scheduling protocol is to divide the entire process of servicing a request into two phases: the registration phase and the service phase (see figure below). The first phase involves creating a TCP connection for an incoming HTTP request and parsing its arguments for later use. The second phase entails processing the request and sending the data back to the client. The reasoning behind this approach is to introduce an invisible third phase between the two phases, in which requests are queued by the parent process and dispatched in accordance with an admission control algorithm. In doing so, a number of new possibilities, such as scheduling and load-balancing on a cluster of servers to improve performance, may be explored.



Any number of approaches can be taken with the scheduling algorithm using this design. One particular manner in which to implement it may be to allow requests to be processed in an order that maximizes the performance of the web server(s), according to what resources are currently available (i.e. cache, I/O). Another method which may improve performance is to distribute the request to a process or server

that is dedicated to a specific type of request (i.e. gif images, HTML) or a specific file size. Finally, in a cluster of servers, one can apply the admission control mechanism to serve as a central point that forwards requests to other machines for load-balancing purposes. The Commonwealth Server Project at Boston University is currently researching this type of mechanism as a part of their main goal.

One of the Commonwealth Server Project's ongoing research areas is networking, which includes routing, packet rewriting, and connection migration. In a cluster of web servers, load balancing becomes an important issue because we ideally want an even distribution of workload across the servers. We must also avoid overloading a single server, since performance is significantly degraded when this occurs. Consequently, research has focused on how to create as equal a load as possible on each machine within a cluster to maximize performance. One of the solutions presented by members of the Commonwealth Server Project is to re-route requests from a central machine to another which currently has the resources to handle the request. This design has an inherent problem - the central machine can become a bottleneck when the network load is high. In response to this, another design has been introduced: the concept of packet re-writing.

Packet re-writing employs the same notion of load balancing among a number of virtual servers; however, there are two main distinctions between the two approaches. In packet re-writing, there are a number of IP addresses associated with one domain name. For instance, when a client wants to access "cs.bu.edu", the name resolves to different IP addresses, provided that the client honors the TTL limit so that subsequent clients wishing to access the same domain name do not receive a cached copy of the previously-used IP address. This ensures that the several IP addresses will be assigned in a round-robin fashion, hence distributing the load more evenly across a cluster of servers. Secondly, load-balancing is enforced further by forwarding a request from a heavily-loaded server to another less-loaded machine to avoid overloading the original server. The newly assigned server will then create a connection with the client, process the request and send back a response, using the IP address of the original machine so as not to confuse the client's application.

Packet re-writing clearly is a very logical solution to load-balancing. However, in packet re-writing, the client's request is merely forwarded to another machine, which is then responsible for establishing a connection with the client. Using this approach, there is no information available about the client's request until the second web server has established a connection. It may be useful to have that knowledge prior to forwarding a request, since we may wish to forward them based on a certain property of the request (i.e. file size requested, file type requested). The notion of connection migration achieves exactly this. The first machine to receive a request from the client establishes a connection, acquires information about the request, and forwards the request to another machine based on the contents of the request. The admission control and scheduling protocol for a single web server and the forwarding mechanism in a cluster of servers carry the same goal - to provide a hook in between the service and registration phases for forwarding or scheduling purposes.

The admission control and scheduling protocol can be implemented in conjunction with the load-balancing designs described above. Its functionality could be adopted to serve as a forwarding mechanism among machines within a cluster. Although this is beyond the scope of this project, it may well be used to incorporate it as a part of the Commonwealth Server Project for future purposes.

---

## The Apache Source Code

As a platform for implementation of the admission control and scheduling protocol, the Apache Server

was selected because of its ease of modification, reliability, efficiency and popularity in today's market. In this section, I will give a detailed overview of the Apache code structure, its functionality, and the Apache API (i.e. handlers, modules, and data structures).

## I. Structure of the Apache source code

The structure of the code can be divided into the following sections:

1. The Server Core Files
2. Third-Party Modules
3. Utility Files
4. Site Directory Files

The server core files consist of the files that deal directly with the handling of incoming requests and servicing of a request. The third-party modules provide code for execution of directives. These modules can be compiled along with Apache or can be left out by commenting them in the configuration file. The ease of modification comes partly from the fact that one can write one's own module and incorporate it into Apache's original functionality. The utility files contain basic utility functions (i.e. buffered I/O routines, allocation of resources). Finally, the site directory contains the configuration file (s), HTML files that reside on the server, and log files for record-keeping. Relevant files for this project are:

`http_protocol.c` -- all communication between the server and the client goes through this file. It includes functions for dealing directly with the client, such as reading the request, sending back acknowledgments, and error handling.

`http_request.c` -- includes functions that control the invocation of proper functions for the various phases of handling a request.

`http_core.c` -- contains the core module structure, as well as tables and command handlers.

`http_config.c` -- contains functions for reading the configuration files when running the `httpd` executable. Also responsible for dispatching to the command handler.

`http_log.c` -- contains the error log.

`http_main.c` -- the main file for parent and child processes. Includes functions that deal specifically with how parent and child processes are evoked, accept connections, and service requests. System startup, restarts, and time-out handling are also included in this file.

`alloc.c` -- contains functions specifically for allocation of resources (i.e. memory, files, child processes).

`httpd.h` -- header file for `httpd` daemon. Contains `request_rec`, `connection_rec`, and `server_rec` structure definitions.

scoreboard.h -- header file for scoreboard. Contains structure of scoreboard, which keeps track of status and process id's for all processes.

httpd.conf -- main server configuration file. Determines much of the server's start-up and run-time settings. Contains directives which oversee the operation of the server as a whole, such as logging and resource management.

access.conf -- defines server settings which affect which types of services are allowed access to, and by whom.

## II. The Apache API

The Apache application programming interface provides a gateway between the programmer and the modules, allowing the programmer's application to make requests of the operating system through them. This section will explore many of the basic concepts behind the Apache API, and how they are manifested in the code.

### Handlers

A handler is a piece of code built into Apache that performs certain actions when a file with a particular MIME or handler type is called. A handler can also be based on filename extensions or by file location, which enables a file to be associated with both a file type and a handler. Apache has a number of built-in handlers, and others can be added by using the AddHandler directive. Its built-in handlers are:

- send-as-is: Sends the file as is, HTTP headers and all
- cgi-script: Executes the file
- imap-file: Names the imagemap file
- server-info: Gets server configuration information
- server-status: Gets the server's status report
- server-parsed: Parses server-side includes
- type-map: Parses as a type map file for content negotiation

Apache handles a request in a series of steps:

- 1) Translates URL filename
- 2) Checks access authorization
- 3) Determines MIME type of file requested
- 4) Sends response back to client
- 5) Logs the request

Each of these steps are handled by examining a few pre-specified modules for the existence of a handler, invoking it if one is found. The handler returns one of the following integers:

- Request handled successfully; returns constant OK.
- No erroneous condition exists, but module declines handling the step; returns constant

DECLINED.

- Error detected; returns one of standard HTTP error codes; terminates handling of request.

## Per-Request Information

All handlers take one argument, the `request_rec` structure. This structure stores all necessary information about a single request made by a client to the server for use by various functions in both the modules and core files. Generally, a single connection results in the generation of one `request_rec` structure (information about a particular connection is held in a `connection_rec`).

```

struct request_rec {

    pool *pool;
    conn_rec *connection;
    server_rec *server;

    request_rec *next; /* If we wind up getting redirected,
        * pointer to the request we redirected to.
        */
    request_rec *prev; /* If this is an internal redirect,
        * pointer to where we redirected *from*.
        */

    request_rec *main; /* If this is a sub_request (see request.h)
        * pointer back to the main request.
        */

    /* Info about the request itself... we begin with stuff that only
        * protocol.c should ever touch...
        */

    char *the_request; /* First line of request, so we can log it */
    int assbackwards; /* HTTP/0.9, "simple" request */
    int proxyreq; /* A proxy request */
    int header_only; /* HEAD request, as opposed to GET */
    char *protocol; /* Protocol, as given to us, or HTTP/0.9 */
    int proto_num; /* Number version of protocol; 1.1 = 1001 */
    char *hostname; /* Host, as set by full URI or Host: */
    int hostlen; /* Length of http://host:port in full URI */

    char *status_line; /* Status line, if set by script */
    int status; /* In any case */

    /* Request method, two ways; also, protocol, etc.. Outside of protocol.c,
        * look, but don't touch.
        */
}

```

```

char *method; /* GET, HEAD, POST, etc. */
int method_number; /* M_GET, M_POST, etc. */

int sent_bodyct; /* byte count in stream is for body */
long bytes_sent; /* body byte count, for easy access */

/* MIME header environments, in and out. Also, an array containing
 * environment variables to be passed to subprocesses, so people can
 * write modules to add to that environment.
 *
 * The difference between headers_out and err_headers_out is that the
 * latter are printed even on error, and persist across internal redirects
 * (so the headers printed for ErrorDocument handlers will have them).
 *
 * The 'notes' table is for notes from one module to another, with no
 * other set purpose in mind...
 */

table *headers_in;
table *headers_out;
table *err_headers_out;
table *subprocess_env;
table *notes;

char *content_type; /* Break these out --- we dispatch on 'em */
char *handler; /* What we *really* dispatch on */

char *content_encoding;
char *content_language;

int no_cache;

/* What object is being requested (either directly, or via include
 * or content-negotiation mapping).
 */

char *uri; /* complete URI for a proxy req, or
           URL path for a non-proxy req */

char *filename;
char *path_info;
char *args; /* QUERY_ARGS, if any */
struct stat finfo; /* ST_MODE set to zero if no such file */

/* Various other config info which may change with .htaccess files
 * These are config vectors, with one void* pointer for each module
 * (the thing pointed to being the module's business).
 */

```

```

void *per_dir_config; /* Options set in config files, etc. */
void *request_config; /* Notes on *this* request */

/*
 * a linked list of the configuration directives in the .htaccess files
 * accessed by this request.
 * N.B. always add to the head of the list, _never_ to the end.
 * that way, a sub request's list can (temporarily) point to a parent's list
 */
const struct htaccess_result *htaccess;
};

```

The request\_rec structure holds the following main pieces of information:

- pointers to resource pool
- pointer to per-server and per-connection structures
- information about the request itself
- MIME header information
- URL/path information

The first section of the request\_rec structure contains pointers to a resource pool which will be cleared after the server finishes processing the request. A pool is a grouped collection of resources that are appointed to each invocation of the server. A pool will be created upon startup, subpools will be created for subprocesses/child processes, and released accordingly once the pool is destroyed. This functionality proves very useful for error control; if an erroneous request comes in, Apache automatically destroys all allocated resources corresponding to that request. This section also holds pointers to the appropriate conn\_rec and server\_rec structures. The second section of the request\_rec structure holds information about the request itself. The client's request is parsed and stored in variables describing what protocol the client is using, host information, request methods, and so forth. Other necessary information such as MIME header environment attributes of the client's original request, (i.e. MIME header on client's request, MIME header to be sent with response), content-type and content-encoding variables are stored in the structure. Finally, a set of character strings hold information about what file is being requested by the client. These include the URI, filename, path information, query arguments, and status flags.

### Per-Server Information

Every invocation of the httpd executable results in a fully capable, independent process that can handle all the basic functions of a web server. Since any of these instances can handle a request for any of the configured virtual or main hosts, we need to keep record of the information associated with each host. The server\_rec structure serves this purpose:

```

struct server_rec {

    server_rec *next;

```

```

/* Full locations of server config info */

char *srm_confname;
char *access_confname;

/* Contact information */

char *server_admin;
char *server_hostname;
short port; /* for redirects, etc. */

/* Log files --- note that transfer log is now in the modules... */

char *error_fname;
FILE *error_log;

/* Module-specific configuration for server, and defaults... */

int is_virtual; /* true if this is the virtual server */
void *module_config; /* Config vector containing pointers to
 * modules' per-server config structures.
 */
void *lookup_defaults; /* MIME type info, etc., before we start
 * checking per-directory info.
 */
/* Transaction handling */

struct in_addr host_addr; /* The bound address, for this server */
short host_port; /* The bound port, for this server */
int timeout; /* Timeout, in seconds, before we give up */
int keep_alive_timeout; /* Seconds we'll wait for another request */
int keep_alive; /* Maximum requests per connection */

char *path; /* Pathname for ServerPath */
int pathlen; /* Length of path */

char *names; /* Wildcarded names for HostAlias servers */
char *virthost; /* The name given in <VirtualHost> */
};

```

Data that is stored in this structure is rather self-explanatory; it holds administrative information (i.e log files, contact information) and transaction handling information that was specified in the configuration files before starting up Apache (i.e. `keep_alive_timeout`, bound address and port number of the server).

### Per-Connection Information

Every time a connection is initiated between a client and the server, the core ensures that the right

information about the connection is available to modules and other functions in order to process the request. If the server is configured not to allow persistent connections, every time a new request is received, a new `conn_rec` is created and filled in. Otherwise, the same `conn_rec` is kept until the keepalive limit is reached. This, and other information, is packaged in a `conn_rec` structure.

```

struct conn_rec {

    pool *pool;
    server_rec *server;

    /* Information about the connection itself */

    int child_num;          /* The number of the child handling conn_rec
*/
    BUFF *client;          /* Connection to the guy */
    int aborted;           /* Are we still talking? */

    /* Who is the client? */

    struct sockaddr_in local_addr; /* local address */
    struct sockaddr_in remote_addr; /* remote address */
    char *remote_ip;        /* Client's IP address */
    char *remote_host;      /* Client's DNS name, if known.
                            * NULL if DNS hasn't been checked,
                            * "" if it has and no address was found.
                            * N.B. Only access this though
                            * get_remote_host() */
    char *remote_logname;   /* Only ever set if doing_rfc931
                            * N.B. Only access this through
                            * get_remote_logname() */
    char *user;             /* If an authentication check was made,
                            * this gets set to the user name. We assume
                            * that there's only one user per connection(!)
                            */
    char *auth_type;        /* Ditto. */

    int keepalive;          /* Are we using HTTP Keep-Alive? */
    int keptalive;          /* Did we use HTTP Keep-Alive? */
    int keepalives;         /* How many times have we used it? */
};

```

### III. Apache Modules

Apache has a number of standard modules that can be used to perform extra functions beyond its basic capabilities. A module is a piece of code that executes a number of directives that can be compiled into Apache by uncommenting its name in the compilation configuration file. One can also customize Apache's functionality by writing modules of his own. Most of the Apache modules that were downloaded with the source code are not required for the purposes of this project, and will be left

uncompiled to prevent unnecessary degradation in performance.

#### **IV. Apache Directives**

Apache has approximately 150 configuration directives that control startup and run-time settings of Apache. They are located in the configuration files, and are usually set to "default" unless otherwise specified by the webmaster. Relevant directives include:

**KeepAlive:** sets maximum number of requests per connection (i.e. persistent connections)

**KeepAliveTimeout:** sets number of seconds to wait for the next request.

**MaxClients:** sets the number of requests that will be dealt with simultaneously

**MaxRequestsPerChild:** sets the number of requests each child can handle. A child server will die after having served this many requests.

**MaxSpareServers:** sets the maximum number of IDLE child servers at any given moment.

**MinSpareServers:** sets the minimum number of IDLE child servers at any given moment.

**StartServers:** sets the number of child servers at startup.

New child servers are started at the rate of one per second; therefore, on a heavily-loaded system, one must carefully decide these settings to avoid creating unnecessary overhead. Since traffic on the World Wide Web is typically bursty, it may be a good idea to have a few idle servers always running in the background to handle a sudden surge in network traffic. Heavily-loaded servers usually do well in the range of **MaxSpareServers** set to 64 and **MinSpareServers** set to 32. It may also be practical to set **MaxRequestsPerChild** to a relatively high number to prevent expensive forking overhead every time a child process dies.

---

## **How Does Apache Work?**

Apache runs in any multitasking operating system such as UNIX and Windows NT. The binary is called `httpd`, and any number of these daemons can be running in the background at any given moment. During its idle state, Apache simply listens to the IP address(es) and port number(s) that were defined in the configuration files. When a request arrives on a valid port, Apache accepts it, reads the HTTP header, and forwards the request to the parent process.

### **Basic Code Sequence**

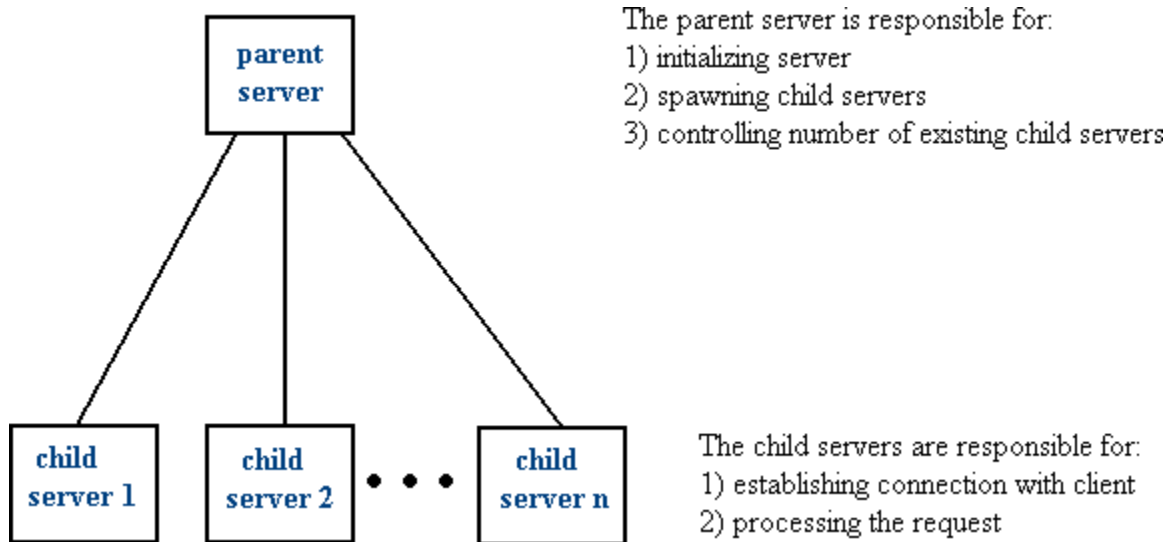
## Parent Process

- executes basic startup operations such as reading config files, setting group privileges, and opening log files.
- creates a socket, binds the local address to it, and listens to socket for incoming connections.
- forks off number of startup processes specified in config file.
- enters maintenance loop:
  - continually checks each child process to see whether it is dead or not.
  - if child is dead, it removes it from the scoreboard in shared memory, writing "SERVER\_DEAD" in the status field.
  - counts the number of idle servers. If it is less than MaxSpareServers, it creates a new child server.
  - counts the number of running child processes. If it exceeds MaxClients, it will not create a new child process.
  - stays in loop until Apache is killed.

## Child Process

- if child is idle, it checks how many child processes are currently idle. If it exceeds MaxSpareServers, it kills itself and exits.
- if MaxRequestsPerChild is reached, it kills itself and exits.
- if not, it loops waits until a request is received and updates its status in the scoreboard to "SERVER\_READY".
- once a request is received, it seizes the mutex lock and accepts the connection, then releases the mutex lock for another child process to seize.
- updates the scoreboard to "SERVER\_BUSY\_READ".
- reads the request and fills in information about the request in the request\_rec structure.
- updates the scoreboard to "SERVER\_BUSY\_WRITE".
- processes the request and sends a reply back to the client.
- then checks for persistent connections in a loop:
  - if KeepAlive has not been reached, it updates the status in the scoreboard to "SERVER\_BUSY\_KEEPALIVE".

- reads another request from the same connection and updates the scoreboard to "SERVER\_BUSY\_WRITE".
  - processes the request and sends back a reply to the client.
  - stays in the loop until the KeepAlive limit has been reached, then closes the connection, exits the loop, and waits for another request through a new connection.
- repeats entire process until MaxRequestsPerChild is reached.



The number of child processes is controlled dynamically by the parent process, thereby continuously adjusting to the current load on the server:

- Upon startup, StartServers (see Directives) are created.
- The parent process counts the number of IDLE child servers. If it is less than MinSpareServers, it will create new child servers.
- The child servers count the number of IDLE servers, during its IDLE state. If it exceeds MaxSpareServers, it exits and dies.
- Child servers keep track of the number of requests served. If it exceeds MaxRequestsPerChild, it exits and dies.
- MaxClients limits the number of servers that can run at one time. The parent server will not create new children if this limit is

reached.

Apache's implementation is rather simple. The parent process continually spawns new processes when necessary, and all child processes die by themselves when they have reached their request limit. A table in shared memory is kept for process synchronization, which keeps track of each server and its current status:

SRV #	PID	STATUS	HOST	REQUEST
0	2181	IDLE	128.197.12.3	—
1	2182	READ	128.197.12.3	GET/a.gif http/1.0
2	2183	SEND	128.197.12.3	GET/b.html http/1.0
3	2184	START	128.197.12.3	—
4	2185	DEAD	—	—

•  
•  
•

Key for STATUS field:

"IDLE" - idle, waiting for request

"START" - starting up, new child server

"KEEPALIVE" - idle, waiting for request from same connection

"DNS" - DNS lookup

"READ" - reading request

"SEND" - sending reply

"LOG" - logging

"DEAD" - server dead, open slot for another server

An important feature of this implementation is the fact that only one child server can accept an incoming connection at any time. When idle, a child server waits for an incoming request. Once a request is received, it seizes the mutex, only if it is available. If it is not available, it will wait in a queue until its turn is reached. Once the mutex is seized, it will accept the connection, immediately releasing it thereafter. Then it services the request, and then goes back to the idle state. This loop repeats until `MaxRequestsPerChild` is reached.

---

## Admission Control and Scheduling on the Apache Web Server

The design of the modified web server incorporates the very essence of the fundamentals of admission control. However, it will also maintain most of the features of the original version of Apache to preserve its reliability and efficiency. In this section, I will give a detailed overview of the changes made to the original version of Apache.

## I. Overview of Implementation

The idea behind the implementation of admission control is to divide the sequence of steps of serving a request into two phases - the registration and service phases. In order to do this, we introduced a mechanism to block processes immediately after they have read and parsed a request. This blocking mechanism is used to make a process wait until it is instructed to proceed with the actual transmission of the data. The manner in which a new process is selected to run is distributed among the child processes. Each child process, upon completion of handling a request, has the capacity to select a new process to begin handling its request. The parent process is only responsible for administrative duties such as server initialization and child process creation.

## II. Implementation

To add new functionality to the original version of Apache, we modified the main module, "http\_main.c", and added a new module, "admctrl.c". The file "admctrl.c" contains the new functions relating to admission control and scheduling: put\_child\_in\_queue, get\_next\_child, mark\_child\_free, setup/destroy\_requests\_table, setup\_semaphores, semaphore\_wait, and semaphore\_signal. The main module contains code for both the parent process and the child processes; each section was modified to handle the functions described above.

To keep information about processes and their requests during the child main loop, we created a table in shared memory for scheduling and administrative purposes. Within this shared structure, an abridged version of the request\_rec structure holds information necessary for a given scheduling policy to run. Each entry in this table holds the following information about a particular request:

- state: indicates a free, waiting, or running process
- number: arrival order
- filename: URI requested
- size: requested file size
- wait-time: total waiting time in the queue
- service-time: total processing time of a request

The rest of the shared structure contains the following fields:

- wait/no-wait counter: number of requests that wait/bypass the queue
- run\_p: number of running processes at any given moment
- queue\_l: number of processes waiting in the queue at any given moment
- other information such as cumulatives and statistics for output

The maximum number of entries in our request structure is set to 30, due to the fact that Linux 2.0.31 does not support the creation of more than 32 semaphores within a semaphore group. This limit is established by the "SEMMSL" constant declaration in "sem.h" in the kernel source. Upon startup of Apache, our modifications create StartServers=30 number of processes, as defined in the "httpd.config" file. The number of child servers is fixed throughout execution, as opposed to the original version of

Apache, which controls the number of child servers dynamically according to the load.

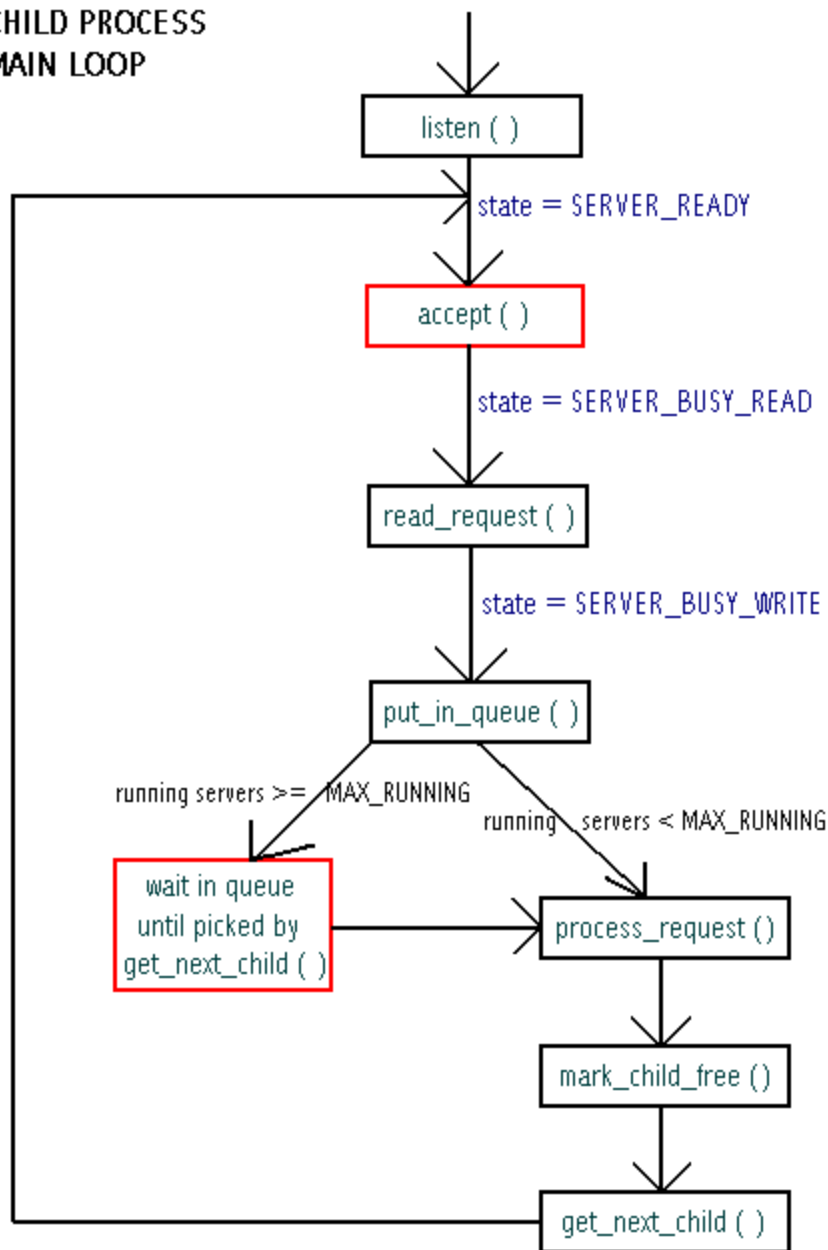
There are two main loops in Apache, one for the parent process (`standalone_main`), and the other for the child processes (`child_main`). We modified the `standalone_main` loop in order to remove the adaptive technique of creating/destroying child processes based on the number of active child processes, which is the standard behavior of Apache. To do this, the parameter `MaxRequestsPerChild` was set to 0, and `MaxSpareServers` was set to 31, to avoid killing off child processes when system load is low. A new parameter, `MAX_RUNNING`, was added to configure the server to limit the number of processes running concurrently at any given time. This parameter must be set to any number less than 30, due to the fact that each process requires a semaphore for blocking purposes. Whenever a new request arrives, the admission control algorithm checks this value and decides whether or not to block the process based on the number of servers currently active.

### III. Code Sequence

In the revised version of Apache, the use of semaphores is heavily relied upon, for two reasons. Firstly, they are used in order to block processes immediately after the registration phase has been completed. At this moment, the process is placed in a queue, where it waits to be selected to run by the scheduling algorithm. Secondly, semaphores are used to protect critical sections within the code that access shared data by all child processes.

The parent process in our version of Apache essentially handles the same duties as it had previously, with a few minor exceptions. Now, it is responsible for handling the creation and destruction of semaphores and data structures that were added to the code. It also updates data in shared memory that is needed to output statistics for a specific run.

The child process main loop is where most of the revisions were made. The figure below breaks down the steps each child process takes from the `listen()` system call until the request is processed and sent back to the client:

CHILD PROCESS  
MAIN LOOP

Each child process, upon creation, begins by server initialization and updating its status to "SERVER\_READY" in the scoreboard. The first process opens the critical section, then waits for an acceptable connection to arrive at the socket, locking around the accept() call to prevent two or more processes accidentally accepting a single connection simultaneously. Once one arrives, it releases the mutex, and the next child server enters the critical section. After the accept mutex is released, server status is updated to "SERVER\_BUSY\_READ" at which time it proceeds to read the request and fill in the request\_rec struct. Next, server status is updated to "SERVER\_BUSY\_WRITE", and the function put\_child\_in\_queue() is called. This function's main purpose is to determine whether the number of running servers has exceeded MAX\_RUNNING - if this condition holds, it will signal the process to wait in the queue before proceeding with the request. Otherwise, (i.e. there is no queue), the process

calls `mark_child_free`, which updates the appropriate information in the request table and calls `get_next_child()`. This function is responsible for the actual evaluation of the scheduling policy to select the next process in the queue to begin transmission of data in response to the request. Once this is determined, the process is awakened, and then returns to the beginning of the loop, marking its status as "SERVER\_READY" in the scoreboard, waiting for subsequent requests to arrive.

#### IV. Scheduling Policies

In the modified version of Apache, an additional queue is inserted immediately after a child process reads and parses a request and its arguments. This extra queue acts as a central point in which requests wait until they are signalled to begin transmission of data. The manner in which they are selected is determined by the scheduling policy. We conducted experiments using three scheduling policies: FIFO, Shortest-File-First, and Batch Processing.

FIFO essentially uses the same concept as the original version of Apache; requests are served in the same order that they arrive in. Shortest-File-First employs the concept of giving precedence to the processes requesting smaller files (i.e. HTML) over the larger files (i.e. .wav and .gif files). This policy is aimed at achieving better overall performance of a web server by allowing the smaller, easier-to-serve requests to bypass the larger files in the queue; thus reducing the average response time. Batch Processing similarly gives precedence to processes requesting documents that were most recently served. The purpose of this policy is to optimize the use of the cache by reducing the total number of cache misses, as well as the number of times it may be necessary to retrieve a file from main memory. The performance of these 3 policies are presented in the next section.

---

## Performance Evaluation

### I. Testing Environment

In order to assess the performance of the web server implementing the admission control and scheduling policies, we created a simulated web environment using a URL request generator tool, or SURGE (Scalable URL Reference Generator), which generates a sequence of URL requests in accordance with distributions obtained from statistics of real world Web activity for several different factors, including document popularity, document sizes, request sizes, temporal/spatial locality, and embedded document count. SURGE is a multiprocessed and a multithreaded application, enabling us to run several experiments using different parameters to simulate all levels of load to test our server. Typically, we ran SURGE with 5 client subprocesses for 5 minutes on each of the three DELLs. The number of threads per client determined the load factor; this number varied from 10 (representing a light load) to 60 (very heavy load). The systems we used for test purposes were Pentium II 233 MHz systems.

The testing setup was completed by compiling each version of Apache running different scheduling policies (FIFO, Shortest-File-First, and Batch Processing), with different MRP's (1, 2, 5, 10, 15, 20, and 25) and running each server separately throughout the duration of every test. To preclude discrepancies in data from the experiments, we repeated each test a minimum of 5 times.

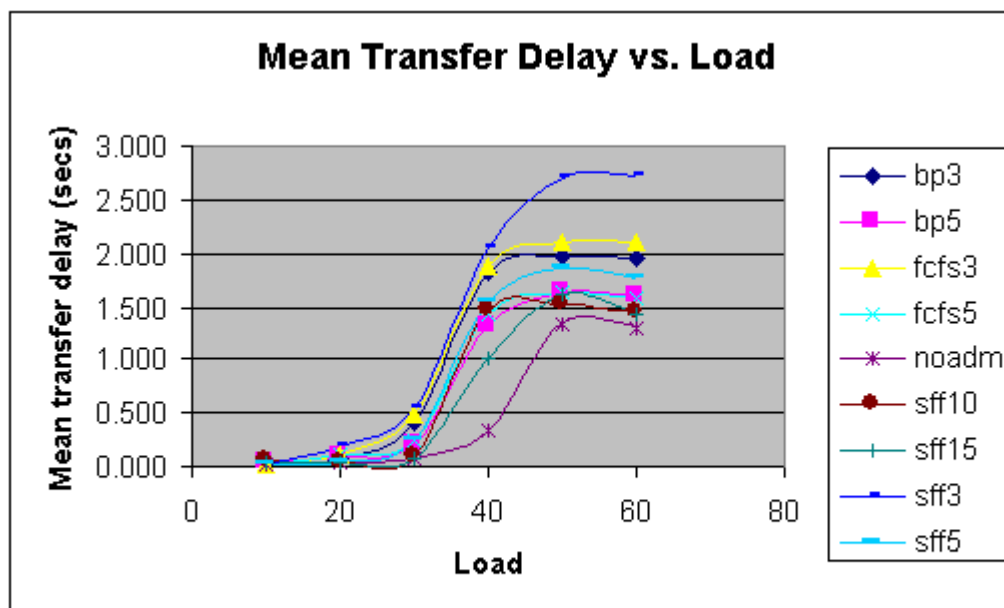
### II. Data Gathered

Data was gathered from 3 sources for analysis for our web server. SURGE provided us with statistics from the client side, such as mean transfer delay and throughput. Internal measurements within Apache were taken as well, to gather statistics for the changes we made within the server to implement scheduling/admission control. Finally, WebMonitor provided us with statistics at the kernel level, such as CPU utilization and time spent making system calls.

### III. Results

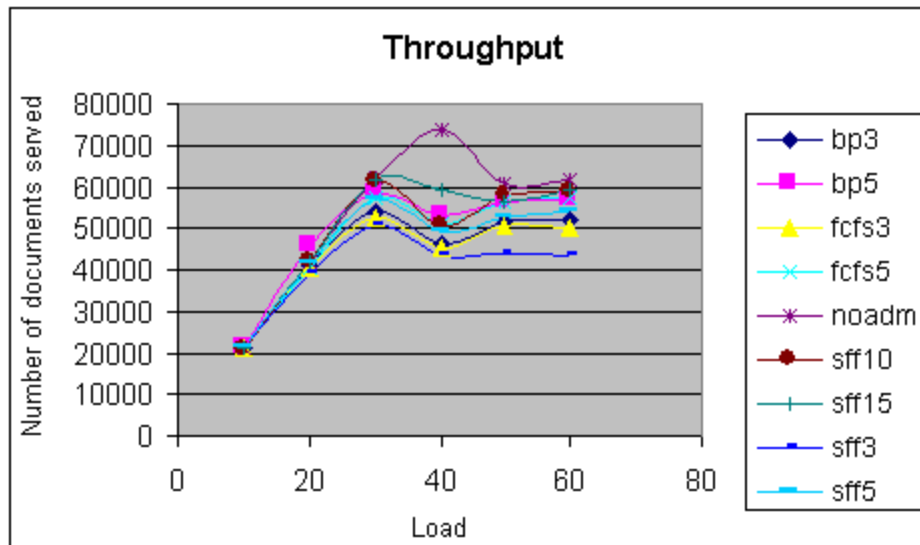
#### a) mean transfer delay vs. load

A good indication of whether a server performs well is the measurement of mean transfer delay from the client's point-of-view. We compiled all the data for most of the different servers running different scheduling policies, and compared the mean transfer delay with the original version of Apache:



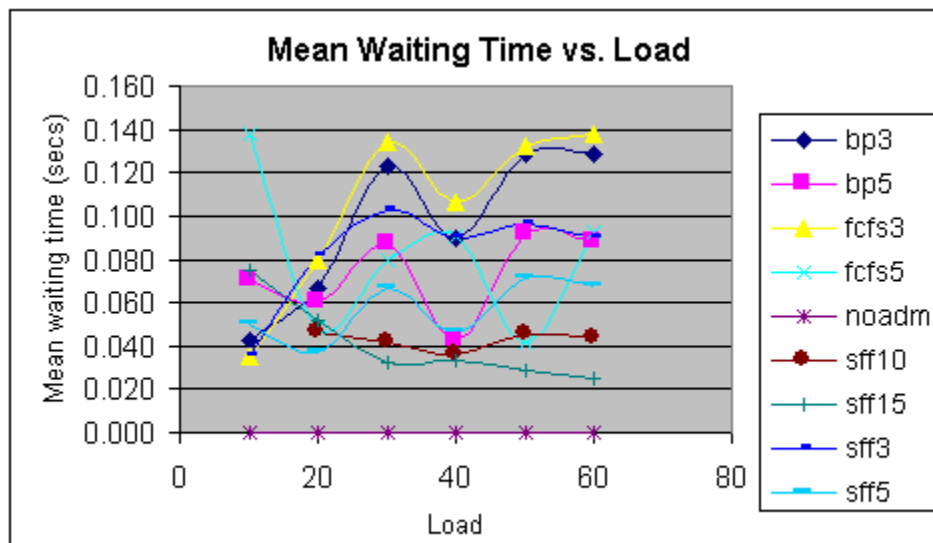
From this graph we can easily see that the original version of Apache is capable of sending back information to the client at a slightly faster rate than the others which run admission control and scheduling. This is to be expected, due to the fact that there is overhead introduced by adding functionality such as queuing requests in between the registering and service phases of a request. However, the servers running Shortest-File-First with a MRP of 15 and 10 are only a fraction of a second behind the original version.

#### b) throughput vs. load



Again, throughput is optimal for the original version of Apache due to the fact that a certain amount of overhead is introduced when adding admission control and scheduling. Still, Shortest-File-First running with MRPs equal to 10 and 15 are very close to the original version with no admission control. There is also a significant performance hit when load reaches 30 (medium load), for no apparent reason, which declines when load approaches 40. This performance degradation may be significantly harmful to a single web server; however, when admission control and scheduling is implemented in a distributed environment, the performance decline will become negligible as the number of documents served increases.

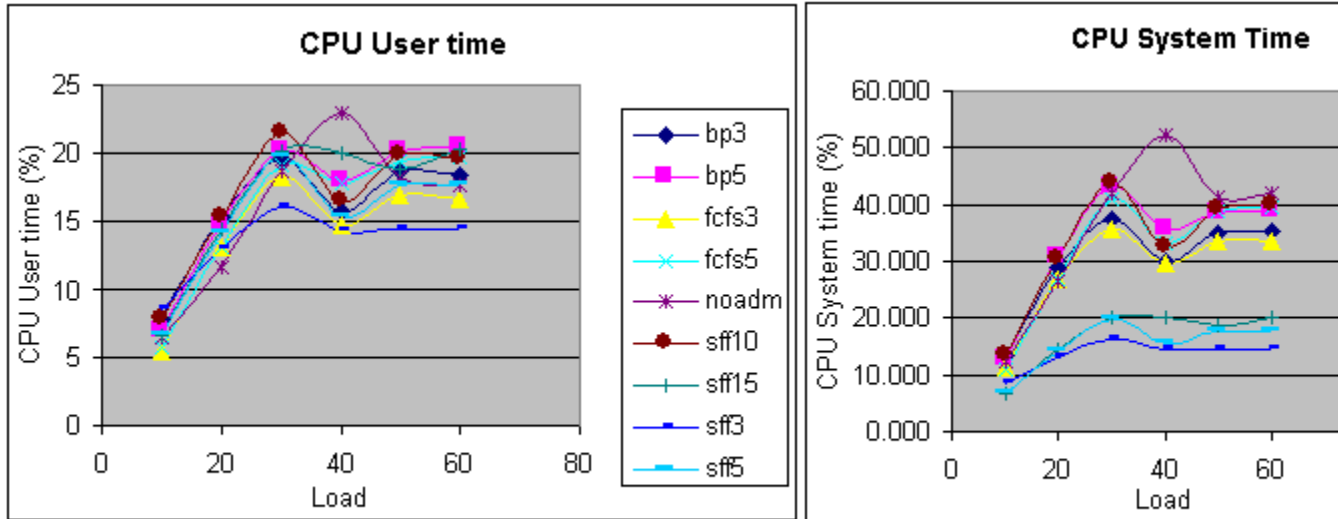
### c) mean waiting time vs. load



The mean waiting time represents the average time a request spent in the queue, waiting to be dispatched by the scheduling algorithm. In low loads, most requests did not have to wait in the queue, because the MRP limit was not reached. However, for higher loads, requests were placed in the queue, and the

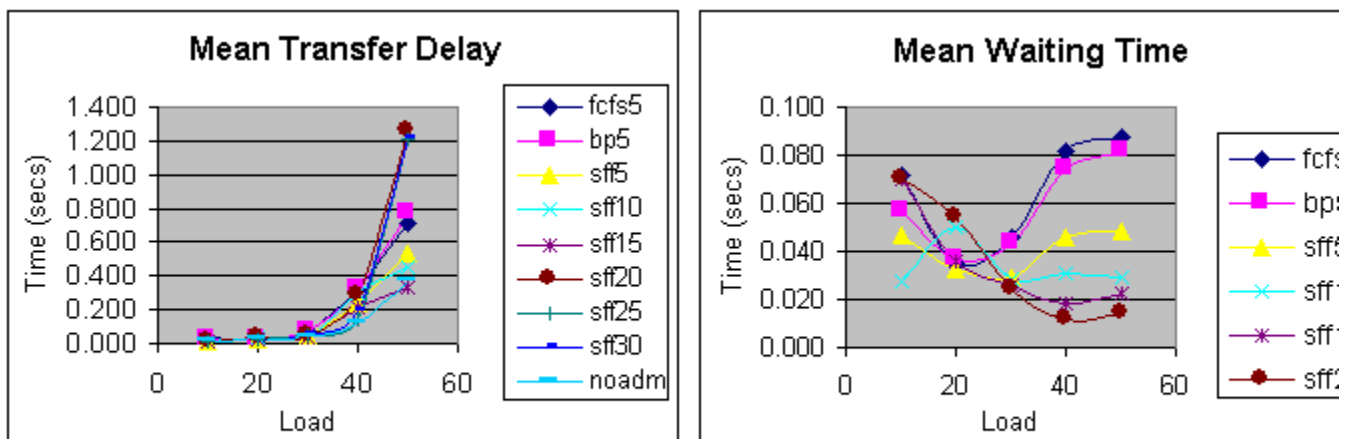
effects of the different scheduling policies can be seen more clearly. As seen before, the Shortest-File-First running with MRP's equal to 10 and 15 perform the best between loads 30 and 60.

#### d) CPU User/System Time



From these graphs, it is apparent that when the load is equal to 30, CPU user time and system time hit a peak. The same is seen for the server without admission control and scheduling, most likely for the same reason, but at a different threshold (load = 40). Once load increases, the percentage of CPU used decreases. Also, CPU system utilization for Shortest-File-First with MRP's equal to 3, 5, and 15 are significantly lower than the rest of the servers shown in the graph, possibly due to cache optimization achieved by allowing the shorter, more popular files to be served first, as page swaps are accounted for in the system, not the application.

#### e) comparison of SFF, BP, and FCFS



Again, it is apparent that the Shortest-File-First scheduling policy performs the best among the three scheduling policies, for various MRP's. Mean transfer delay does not differ much when the load is low,

but as it increases, and the number of requests that have to wait in the queue gets longer, performance for Shortest-File-First becomes increasingly better. For Mean Waiting Time, the numbers are a little inconsistent, but as the load exceeds 30, all of the Shortest-File servers perform significantly better than Batch Processing or FIFO.

---

## Current and Future Work

The advantages attained through the implementation of admission control and scheduling can be extended beyond the scope of a single-server environment. The added functionality of being able to distribute requests according to their nature can be applied to a distributed environment for the purpose of forwarding tasks from heavily loaded machines to another that has the resources to handle the specific request. This is one of the objectives of a project currently being researched by the Commonwealth Server Group. The technology, called [Distributed Packet Rewriting](#) (DPR), handles connection routing for forwarding requests in a distributed manner, effectively balancing the load among the individual machines. This contrasts with other centralized approaches, in which all requests are directed through one machine; creating a potential bottleneck for very high loads. Currently, DPR is implemented in two ways. One method relies on a table stored in each machine, with the current loads of each machine, to determine which machine to route a certain request to. The other incorporates the idea of IP-level hashing, and uses the randomly-generated source port number to produce a value to decide where to forward packets to. The admission control and scheduling policies could very well be applied to the concept of DPR with the added functionality of being able to read a request, analyze its contents, and forward it according to the scheduling policy.

Socket migration centers around the same idea, but at the TCP level as opposed to the IP level, for actually creating the connection and serving the request. It uses DPR to rewrite the source IP address when sending back packets to the client, so as to not confuse the client.

There are many ways one could envision implementing admission control for a distributed web server. The main advantage is that any machine will be able to extract useful information about a request, such as file type requested, file size, and URL to re-route the request as needed to another machine that can better handle the request. The individual machines could be organized in a way that each machine dedicates itself to a [specific range of file sizes](#), file types, or actual files. When an incoming request arrives at a specific machine within the cluster, it will first parse the header of the request, then either serve the request itself, or use socket migration to re-route the request to another machine. In this way, an optimal balance of load can theoretically be achieved.

Socket migration and DPR can also be implemented in conjunction with a task assignment policy, as mentioned above. The only difference would be that a "collective" centralized point to which the incoming requests arrive must be present in order to create a queue for scheduling purposes. For example, in a cluster of servers, a few machines could be dedicated to receiving incoming requests from clients. Another set of machines, invisible to the outside, could be dedicated to actually serving the requests, based on file size. A request will arrive at any of the machines accepting incoming connections, get inserted in a queue, and deployed according to a scheduling policy, such as Shortest-File-First or Batch Processing. When the request reaches the beginning of the queue, the machine will migrate the connection to the server handling the specific range of file sizes, which, in turn will serve the request.

---

Created on: 1997.08.21  
Updated on: 1998.05.26