

# NEW

Next Eternity Web Server

## ECE299: Server Project

**NEW Developers:**

Madhureema Dutta  
Sargunjit Singh Bawa  
Akshay Ahooja

**Date Submitted:**

April 15, 2006

---

## EXECUTIVE SUMMARY

---

In February 2006, group 59 of ECE299 started on the development of NEW – Next Eternity Web Server - a web-server that manages and shares web-based applications accessible to remote computers connected from anywhere to the Internet at anytime. This report presents an in depth design evaluation and description of NEW, drawing its comparison to in-market web servers. NEW's design has met all its required specifications while being efficient, graceful and reliable and hence successful.

This report is divided into three major sections detailing NEW's design description, testing and finally its design evaluation. NEW implements several extension features including Logging, Content Types, Virtual Web Hosting, Automatic Pathname Expansion, GET and POST dynamic requests, Concurrency (Multithreading), Clustering (redirection, load balancing, and heart beating), and Load Generation Testing.

The design description contains an explanation of how each feature works; the testing section gives us an insight into how each of these features were vigorously tested, and finally the design evaluation analyzes these features. Each of these sections is supported with visuals which aid their understanding.

The report discusses all the factors affecting the design of NEW and how it evolved over time. It also explains some of the vital design decisions that were required and how they affected the final product. An overall evaluation assesses NEW to be a robust and reliable server, but not a fast one. The server does not fail, and performs exceptionally within its defined constraints. Its weakness arises when comparing to commercial products, where its concurrent connection number is significantly low, and page load time is quite high.

---

## TABLE OF CONTENTS

---

	Page #
Introduction.....	1
Design Description.....	1
Content Types.....	1
Virtual Web Hosting.....	2
Dynamic Content Handling.....	3
Automatic Pathname Expansion.....	4
Concurrency (Multithreading).....	5
Clustering (Load Balancing).....	6
Testing .....	9
Design Evaluation.....	11
Conclusion.....	12
Appendix A – Class Index .....	13
Appendix B – Component Algorithm Overview .....	14
Appendix C – New Diagnosis.....	17
Appendix D – Load Generation Testing Algorithm.....	18

---

## INTRODUCTION

---

The present report is an in-depth design analysis of NEW – Next Eternity Web Server - a web server built by group #59 as a requisite for the ECE299 course. A request for proposal was presented to the team to design a non-trivial object-oriented software system which closely imitates a commercial web server. The main purpose of the project was to gain experience in implementing a larger software system and working on software in groups.

To meet this request the team designed software written in C++ language, operating on Solaris platform which functions and has enhancements like server management, protocol processing like a commercial web server. Indeed the design of NEW meets all the expectations of the project and was graded 90% successful by the ECE299 TA.

The goal of this report is to discuss and evaluate the design of “NEW”. The report is divided into three main parts comprising of design description with a brief overview and mode of implementation of the enhancements included in the software, testing which includes the various stages of testing and the “load generator” software developed for testing the efficiency of the design and lastly design evaluation of the software NEW in comparison to commercial web servers.

---

## DESIGN DESCRIPTION

---

→ Refer to **Appendix B** for all Design Description flowcharts

### CONTENT-TYPES

To build upon the simple web server which handled only “.html” pages, it was decided to give NEW the ability serve other content types to bring our server at power

with in market web-servers. Some of these included PDF, JPEG/JPG, AVI, SWF, etc. files. This feature was incorporated by making the appropriate changes to the “Content-Type:” header before returning it to the user in *httpserver* (Refer to **Appendix A and B**).

```
If(File Extension = “.jpg”)
Return header = “Content-Type:image/jpg”
Else if(File extension = “.txt”)
Return header = “Content-Type:text/plain”
...
```

## **VIRTUAL WEB HOSTING**

The motivation behind virtual web hosting is to have the flexibility of allowing the web-server to serve multiple domain names using the same IP address.

To further illustrate its importance in the web-server, the domain names offered by the University of Toronto are used as an example: namely “eecg.toronto.edu “, and “eecg.utoronto.ca”.

As a design decision the group decided that the domain name “eecg.toronto.edu” will exclusively map to a directory “public\_www1” and “eecg.utoronto.ca” to directory “public\_www2”. The absolute pathnames are parsed from a configuration file, named *config.txt*. Thus the client request differentiated on the basis of these domain names would have access to files only in their respective directories. The following algorithm is a part of (Refer to **Appendix A and B**).

```
Extract the domain name from the URL:
If(Domain name = “eecg.toronto.edu”)
Set file directory = “public_www1”
If(Domain name = “eecg.utoronto.ca”)
Set file directory = “public_www2”
So on for as many domain names as required....
```

## DYNAMIC CONTENT HANDLING

Often clients request specific data to be interpreted by a program, and expect a custom response to their demand. A classic example of a server handling dynamic data is a google.com web-server where the client specifies a search field and is returned data corresponding to that search. In general, client dynamic requests to web servers fall into two categories: GET requests, and POST requests. NEW has the ability to supports both.

*Dynamic **GET** requests:* In this case the client specifies the parameters to the dynamic program in the URL itself. Our web-server parses the URL separates out the dynamic program and its parameters and feeds them into the dynamic program to generate an output which is then returned to the client. Once this custom output is generated and sent to the client the return file is deleted. To take our web-server a step further we incorporated the possibility of passing multiple parameters through the URL.

E.g.: `http://ugsparc213.eecg.toronto.edu?portfolio.dyn?param1=August%param2=31`

Where different parameters are separated by a '%'. In which case we parse the URL such that data separated by a '%' character corresponds to different parameters.

To test for dynamic GET requests we incorporated a program that solves the towers of Hanoi and takes the number of discs as input through the URL and returns to the client a detailed way to solve step by step the towers of Hanoi.

*Dynamic **POST** requests:* POST requests are used when the data to be inputted is quite large and can not be added to the URL. In this case our web server stores the client input into a file and uses this file as input to the dynamic program, after which it proceeds to return the output in a similar manner as GET requests.

To test Dynamic Post Handling we incorporated a simple program that concatenates 5 strings which can be inputted using telnet:

*POST /concat.dyn http/1.1*  
*Content-Length: 25*

*Parameter1 = Concatenate*  
*Parameter2 = this*  
*Parameter3 = string*  
*Parameter4 = for*  
*Parameter5 = me please.*



Output:

*Concatenate this string for me please.*

## **AUTOMATIC PATHNAME EXPANSION**

Up until this added feature, NEW connected to all clients in a similar fashion, regardless of their individual needs. Automatic Pathname Expansion adds a personal touch, by giving each user his/her own directory of servable web pages. Any relative URL (ex. where relative URL is "index.html" in www.cnn.com/index.html), which starts with a twiddle ("/~bob"), is treated as a username login.

The directory of all usernames is stored in a text file within our root directory labeled *users.txt*. This includes all usernames and their destination home folders separated by a colon. An example is given below:

*NameA: /new/users/a*  
*NameB: /new/users/b*  
*NameC: /new/users/c*  
...

On start of the server, this text file is parsed and stored into a link list called users. Once the user has connected onto the server, a very simple algorithm is followed:

*If (first letter of relative URL = "~")*  
    *Search for username in users list*  
    *If (username not found)*  
        *Return back an error page*  
    *Else*  
        *Return back default index page or directory listing from*  
            *username's home directory*

This extension is implemented inside *httpserver*, as one of the many options available after connected to NEW (refer to Appendix X). The group decided to implement this feature to allow for some security when processing information. Separate folders can be created for each user, and allow those users to only access the files we allow. If

files of different security level's are to be created with their respective files, each user can only have access to the allowed files.

## **CONCURRENCY (MULTITHREADING)**

→ Refer to **Appendix B** for redirection and heartbeat flowcharts

Multithreading NEW allowed more than one connection to be processed concurrently. Instead of having one Socket (refer to Appendix X) connecting to one client, the one Socket was used to create a new Socket within what is called a thread. A thread acts as a separate CPU. Each thread runs your program as a separate entity, and does not interfere with any other connections. We used this functionality, and wrapped each connection into a separate thread.

This extension had large implications on our design. The main class Server waited for connections and then wrapped the httpserver class's Run() method in a thread for processing the request. Multithreading in Server was implemented using the following algorithm:

*Main:*  
if (num\_threads < max\_threads)  
    Wait for connection:  
    Establish connection  
    Open thread to handle connection  
    Go back to waiting for the connection

Once the new thread was open, it called the httpserver class's Run() function which followed the following algorithm:

*Thread:*  
Update thread information (global values)  
Parse request  
Send required file/Default page  
Any error handling (ErrorHTML page)  
Kill thread (adjust global variables)



This extension was very important because it created a solid base server to work on top of. Using this extension, several new extensions were added, including server redirection in Web Server Clustering, explained later in this report.

With the ability of several threads running at concurrent times, there came a need to be able to shut down the server when a need for maintenance occurs. This ability is labeled **GRACEFUL SERVER SHUTDOWN**, and follows a very simple algorithm. All major while loops, for accepting connections and parsing requests are controlled by a Boolean variable *flag\_terminate*, which stays 0, unless otherwise specified. When the *kill -HUP <process>* is ran on command prompt, the Signal class interrupts the program and simply changes *flag\_terminate* to a 1. This forces the server to decline all new connections, but, does not close any running connections. Once all connections are closed, it shuts down the server since it cannot enter into the main loop any longer. The important factor in this shutdown is that no running threads are suddenly ended, but rather gracefully terminated.

## **WEB SERVER CLUSTERING**

Robustness and reliability are two features that test a web-server's functionality. A group decision to implement Web Server Clustering was chosen to enhance the functionality, reliability and robustness of NEW.

*The need for Clustering:* A good web-server serves thousands maybe millions of clients per second. It is not possible that a single server handle all these requests simultaneously, quickly and efficiently without failing at some point of time. Therefore we must divide and conquer this problem and to do so web server clustering was added to NEW.

*How it works:* Rather than a single computer/server handle all requests there exists a computer/server dedicated to redirecting client requests (the dispatcher) to other

servers running with the same directories and files. This balances the net load by dividing it among a number of web-servers running as helpers or Clusters to the main dispatcher.

*Design Implications:* Incorporating Web-server Clustering had many drastic design implications and can thus be justified as one of the most important design decision made by the group during this project. Web-Server Clustering could be divided into two major sections namely, 1) launching clusters and redirecting client requests, 2) detecting cluster failures and re-launching those clusters.

*General design overview:* To implement web-server clustering two new classes were added: Dispatcher and Heartbeat. These will be discussed later.

According to our design the main program (Server.C) also under went many changes. Since the same code was now required to run either as a dispatcher/cluster or a normal server, the team decided to separate code using if conditions:

```
If (operation mode = "dispatcher")
{
    Launch Clusters
    Redirect client requests to clusters.
    Request "heart-beat" from clusters.
}

If (operation mode = "cluster")
    Start "heart-beat" with dispatcher

If (operation mode = "server")
    Do normal processing without the above.
```

Launching clusters and redirecting requests (in dispatcher mode only):

The number of clusters to launch at startup is specified in the *config.txt* file. Therefore at startup the main program (in *Server*) reads the *config.txt* file and launches each of the clusters using the *Dispatcher::Launch()* command.

```
If(operation mode = "dispatcher")
While (number of clusters in the config.txt file)
    Launch each server using Dispatcher::Launch() command.
```

Once launched the dispatcher redirects incoming client requests to each of those clusters in a cyclic manner. This is done by adding a public data member 'redirectedhost' to class *httpserver*. Each time a new client connection is created it is assigned a *redirectedhost* depending on the redirection algorithm, as follows:

```
Set relay counter = 0;  
While (infinite)  
    Every time a new connection is created.  
    Assign that connection a redirected host corresponding to relay.  
    Increment relay  
    If (relay = number of clusters)  
        Set relay back to 0;
```

Once assigned a redirected host the dispatcher returns a response,

```
HTTP/1.1 302 Found  
Content-Type: text/plain  
Content-Length: 0  
Location: http://redirectedhost:54020/hey.txt  
Connection: Close
```

Telling the client web-browser to request the same data from another web-server (cluster) running at a location specified by the Location header.

With several server's running and accepting connections, it becomes very difficult to detect failures. For example, if a server that the dispatcher is redirecting to is to crash, it will be never known unless some mechanism for detecting failure is devised. For this, a heartbeat between each cluster and the dispatcher was implemented.

*NEW's Lifeline - The Heartbeat:* The heartbeat is the sending and receiving of constant characters from each cluster to the dispatcher. This is done on a separate port and Socket for each cluster, so it does not interfere with other connections.

```
Dispatcher's Run() Algorithm:  
While (forever)  
    Traverse through list of server's  
        If (current server is not connected)  
            Display message on terminal  
            Launch remote server using "ssh" command  
        Else  
            Read in character
```

*Get next Server in cluster (if last server, restart list)*

Heartbeat's Run() Algorithm:

*Create new Socket and listen on a different port*

*While (forever)*

*Accept a new connection with the dispatcher*

*Write a character to the dispatcher*

*Close connection*

These two classes constitute the heartbeat, and begin at the startup of the server. The heartbeat adds on to the robustness and reliability of any web server.

---

## TESTING

---

The testing of software of such huge proportions is a time intensive task and hence was conducted in two stages. The first stage consisted of module wise testing and the second stage was testing with a specially designed software "load generator" which was used to check the efficiency of the NEW.

### MODULE-WISE TESTING

The initial testing was performed on each enhancement separately and then testing the other enhancements of the web server with the addition of the new enhancement. Since it was a large software system, black box and white box testing on the whole system was quite improbable. Hence the team approached defensive programming mechanism right from the start of the program, assuming that all input will be incorrect, explicitly checking all requirements and exhaustively checking all conditionals. Each new enhancement was checked for utility with reasonable input, robustness by checking what happens with invalid inputs, reliability with long running tests and performance. This involved checking each module with boundary conditions and corner cases. Allocation and deletion of memory for every object created was taken into account for proper memory management. When added concurrency, the server was ran, and requested from several computers multiple times to make sure of its

reliability. After each enhancement was checked with individual test programs, they were incorporated in the main program of the server and then similar checks for robustness and functionality was performed to check that the newly added enhancement worked as per specifications with the main program and all previously added features. Every bug encountered by a team member was logged in a text file for the information of every team member.

After detailed testing and debugging of each individual enhancement with the main program and other enhancements, a software was designed to further test and measure the efficiency of the integrated web server.

#### **LOAD GENERATOR**

A special software called “Load Generator” was designed to test the efficiency of the server. The load generator consisted of three parts. The first part involved generation of static files of varied sizes in the main directory. Static files are divided into four classes on their sizes and nine files of each size was generated. The *rand()* function from the standard library was utilized to decide size of the files within each class. After populating the directory with static content, load generator connects to the web server to send browser requests for the generated files and receives the server response. For this the team utilized the `ClientSocket` class provided. Connection was established with the server by first generating a client socket and then *send()* and *recv()* functions of the socket class were utilized to send and receive files from server. A request for a file named *class0\_5.html* looks like this:

*Get /class0\_5.html HTTP/1.1*

The third and the most important part of the load generator was comparison of requests and responses for efficiency of server and statistics logging. *Stat\_log.txt* was generated for this purpose which recorded the number of connections, number of successes,

number of failures, total number of bytes received, sustained bandwidth and response time.

Hence we see that the load generator is an important piece of software for measuring efficiency of the design. Not only does it measure the efficiency but also records the statistics for easier comprehension.

The algorithm for the Load Generator which highlights the basic functioning of the program covered in **Appendix D**.

---

## DESIGN EVALUATION

---

In order to provide a fair evaluation for NEW, two factors will be taken into consideration: design, and functionality. NEW has been designed to work as a robust web server with minimal thread capacity. Due to minimal experience of the group in server programming, a concurrent web server provides a solid base to work with. The server includes all basic components of a web server, including logging of data. Up till this point NEW is just a one connection server. Adding multithreading allowed the group to enhance the server with several important features. One of these includes clustering. The use of clustering took NEW from a simple, to a moderate or even an advanced server. In case of increased load, it is very simple to add on cluster servers to deal with the overflow of requests. This not only improves functionality of the server, but also its speed. That extension, added on with the robustness of heart beating, makes NEW very reliable.

To evaluate NEW's functionality in a quantitative sense, several factors were taken into consideration. In order to compare how NEW placed among the professional servers, help from the website of Vertain Software <[www.vertain.com](http://www.vertain.com)> was taken. When testing NEW against Vertain's testing protocols, the average page speed was tested to

be 7.45 seconds. Benchmarking to all other major server's, tested on the same site, comes to an average of 4.38 seconds. A 3 second difference is large in the professional market, and will have to be worked on if decided to sell NEW. A full diagnosis is available in **Appendix C**.

When looking into extreme cases, it shows NEW cannot handle a large number of concurrent connections when in regular server mode. To compare, professional servers can handle hundreds of thousands, and even millions of connections at one time. A limit of 50 threads was installed in order to keep page loading time at its highest. It was noted that at high thread counts (past 30), the server would respond very slowly. Clustering, on the other hand, easily doubles, triples, or exponentially grows that number. Each cluster adds an extra 50 threads, which makes NEW expandable.

An overall evaluation assesses NEW to be a robust and reliable server, but not a fast one. The server does not fail, and performs exceptionally within its defined constraints. Its weakness arises when comparing to commercial products, where its concurrent connection number is significantly low, and page load time is quite high.

---

## CONCLUSION

---

Based on the engineering decisions taken during the development of NEW, as well as its design evaluation, the server is rated as being successful on its requirements. NEW includes all the features of a professional web server, while only lagging in timing and number of concurrent requests. Each decision taken has taken NEW to the next level of web serving, proving to be robust, reliable, and consistent.

---

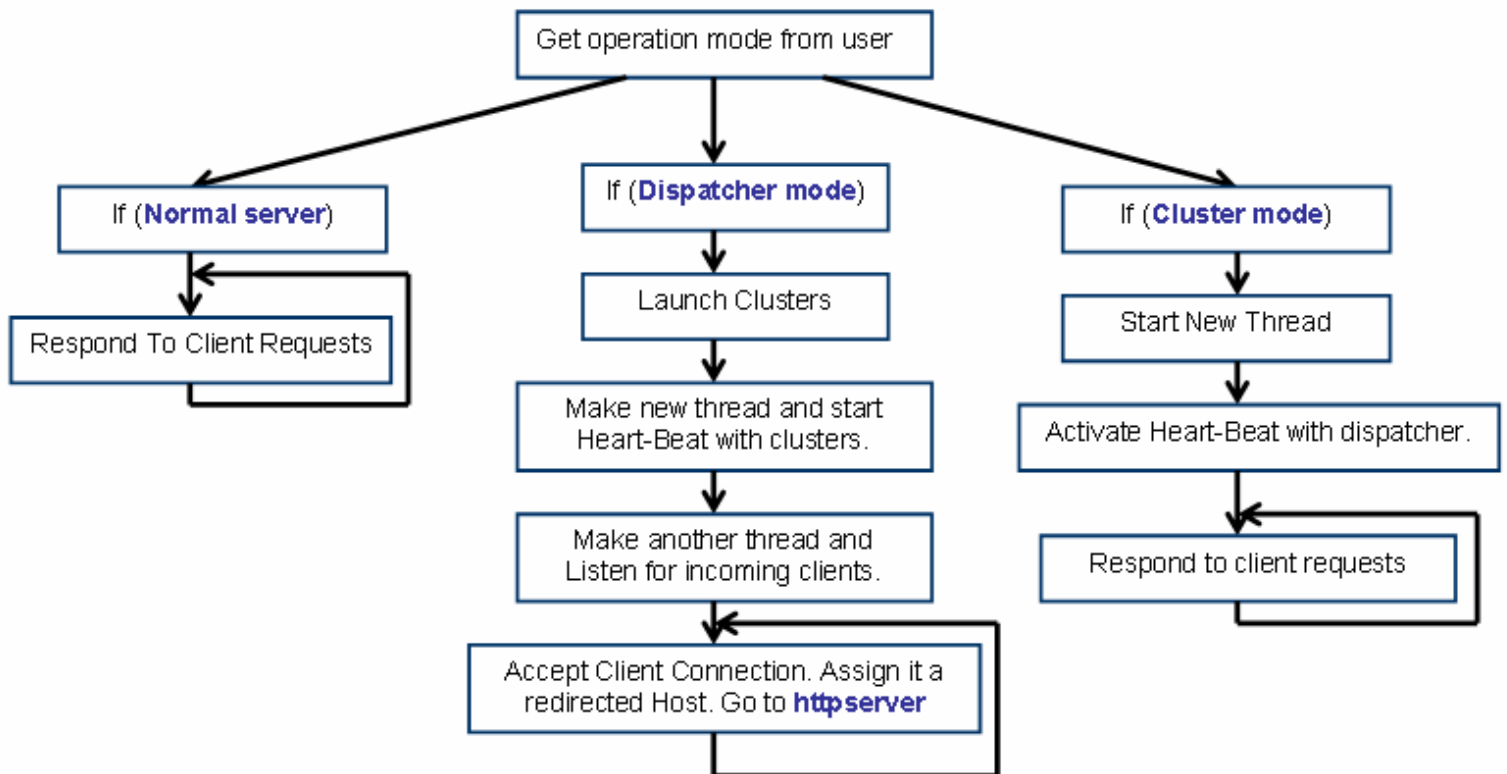
**APPENDIX A**  
**CLASS INDEX**

---

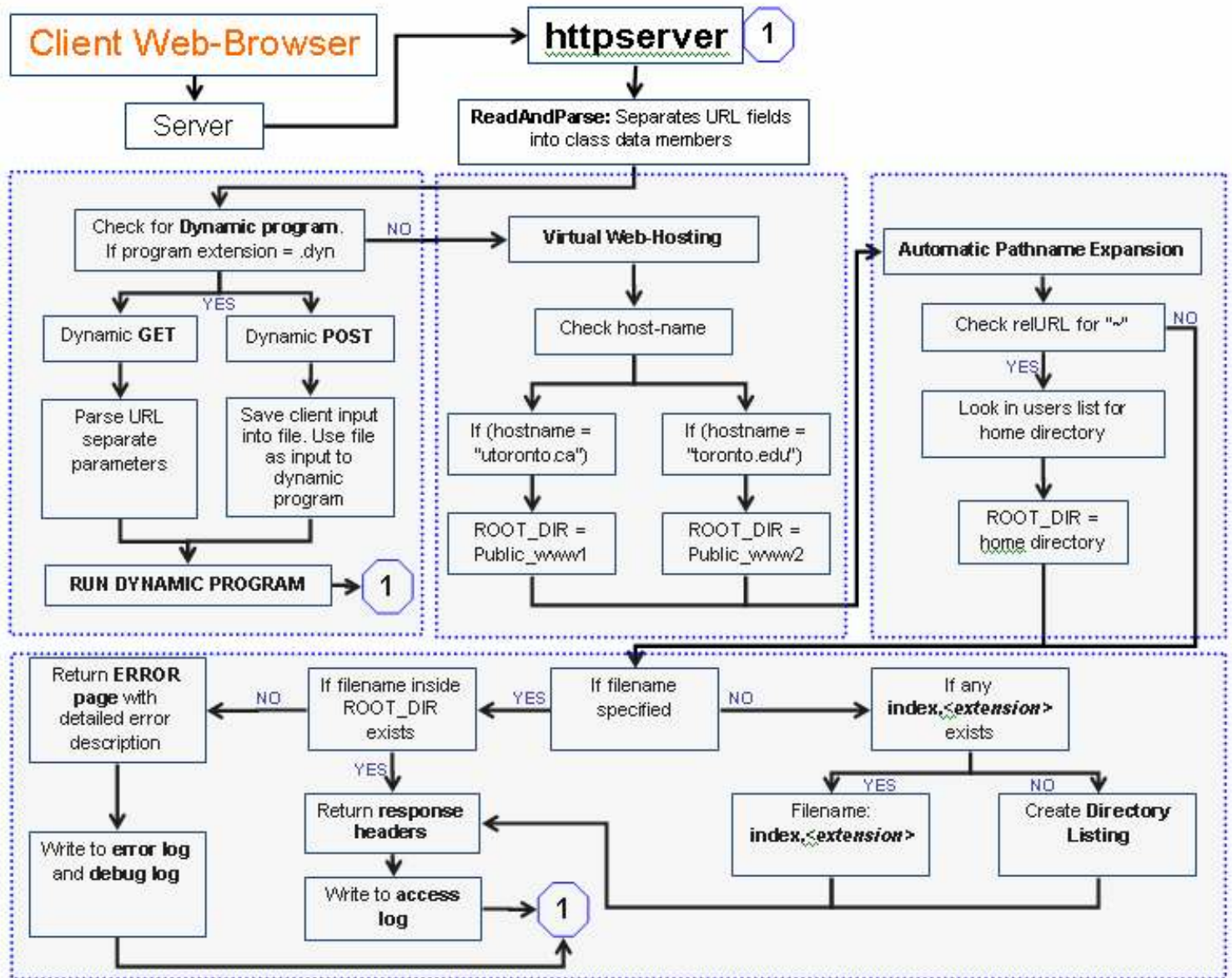
<b>Class Name</b>	<b>Description</b>
ClientSocket	A class to be able to connect to the server, as a client
Config	Creates a linked list using listNode
Copy	For Dynamic Handling
dispatcher	To accept heartbeats from each cluster, and restart clusters if needed
Hanoi	Dynamic program to solve Towers of Hanoi problem
Heartbeat	To send heartbeats to dispatcher
HTTPMessage	Parses the incoming requests
HTTPrequest	Saves the request parameters in an array
httpserver	class directed by Server to send for parsing requests, and send responses
listNode	to create each individual node for Config linked list
log_file	to create and update error, access, and debug log files
Mutex	to lock access for certain sections of code
Server	Class with main() method. Sends requests to appropriate destinations
socket	create socket connections
ssbuf	to wrap a buffer around a socket connection
Thread	to create individual threads and call the Run() method of the thread's class



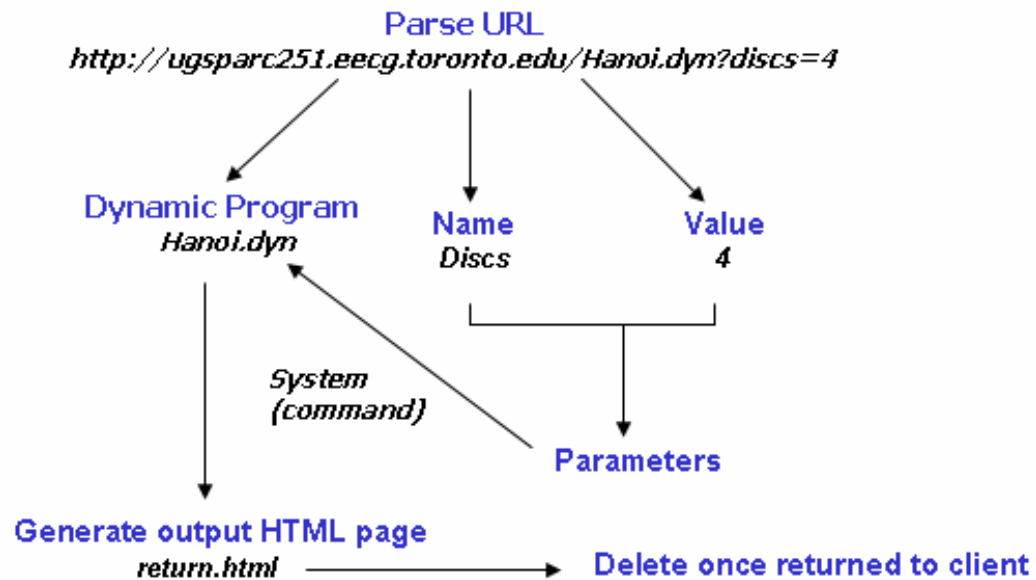
## SERVER



# HTTPSERVER

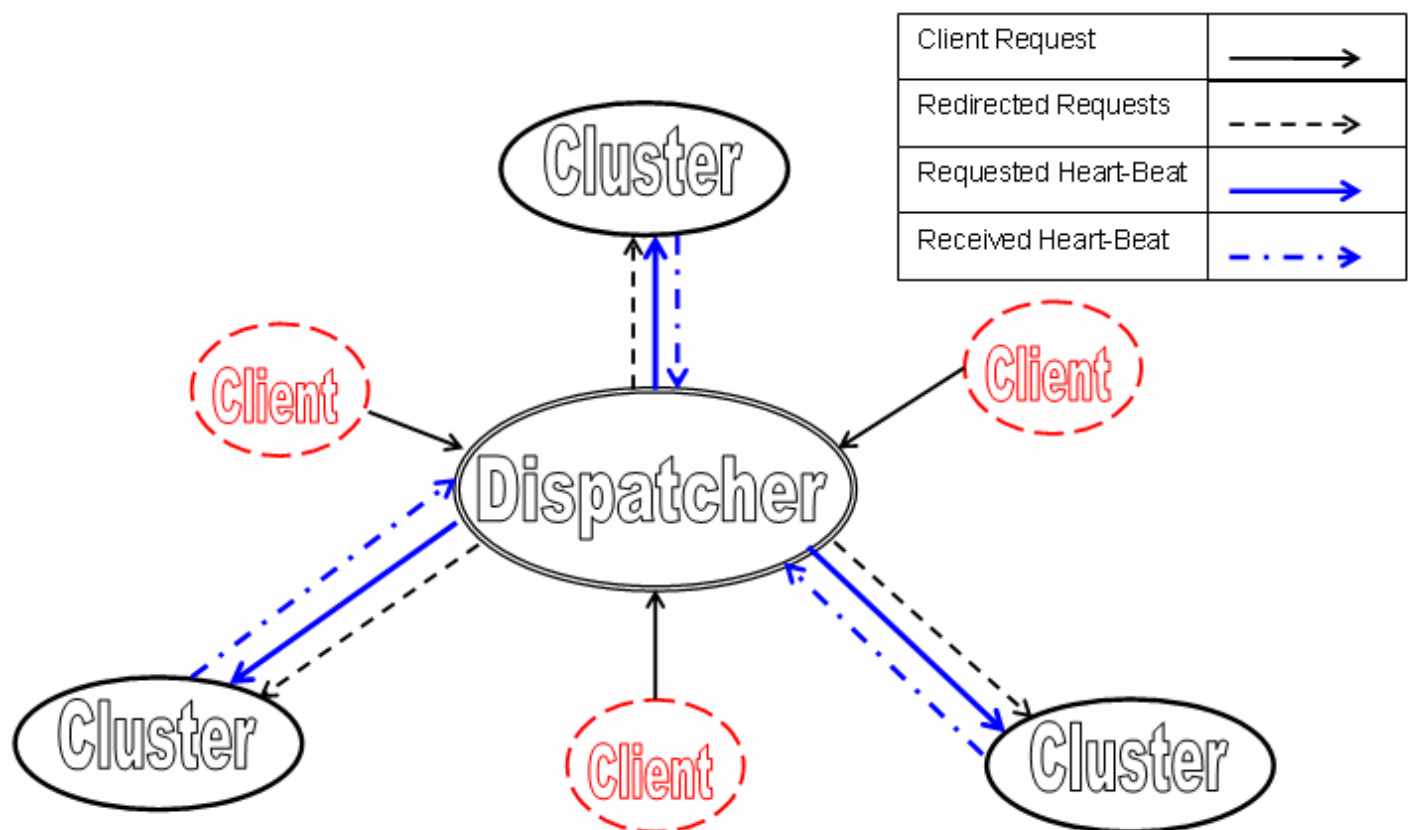


## DYNAMIC CONTENT HANDLING



## CLUSTERING

Redirection and Heartbeat



---

## APPENDIX C

### NEW DIAGNOSES

---

Diagnoses performed at: <http://www.websiteoptimization.com/services/analyze/>

<b>URL:</b>	http://ugsparc231.eecg.toronto.edu:54027/netscape.html
<b>Title:</b>	Netscape.com
<b>Date:</b>	Report run on Fri Apr 14 23:11:44EDT2006

## Diagnosis

### Global Statistics

Total HTTP Requests:	42
Total Size:	186172 bytes

Total CSS imports:	1
Total Frames:	0
Total Iframes:	0

### Object Size Totals

Object type	Size (bytes)
HTML:	72163
HTML Images:	74725
CSS Images:	0
Total Images:	74725
Javascript:	26686
CSS:	12598
Multimedia:	0
Other:	0

### Download Times\*

Connection Rate	Download Time
14.4K	144.49 seconds
28.8K	72.35 seconds
33.6K	62.04 seconds
56K	37.30 seconds
ISDN 128K	11.56 seconds
T1 1.44Mbps	1.19 seconds

\*Note that these download times are based on the full connection rate for ISDN and T1 connections. Modem connections (56Kbps or less) are corrected by a packet loss factor of 0.7. All download times include delays due to round-trip latency with an average of 0.2 seconds per object. With 42 total objects for this page, that computes to a total lag time due to latency of 8.4 seconds. Note also that this download time calculation does not take into account delays due to XHTML parsing and rendering.

### External Objects

External Object	QTY
Total HTML:	1
Total HTML Images:	36
Total CSS Images:	0
Total Images:	36
Total Scripts:	4

---

**APPENDIX D**  
LOAD GENERATOR TESTING ALGORITHM

---

```
Get populating directory from user

If( directory != directory specified in config file)
    Echo error message on terminal
    Exit program

If (directory == directory specified in config file )
    Populate directory
        Get size of file using rand()
        Generate file
        Continue populating for number specified in config file

    Get time of day using gettimeofday() = time 1
    Connect to web server
        Create new Client Socket
    Send requests using send()
    Receive response using recv()
    Continue process for all generated files
    Get time of day using gettimeofday() = time 2
    Generate Stat_log.txt
        Log in statistics :
            [no. of connections] [no of successes] [no. of failures] [total bytes
            received] [response time (time 2 – time 1)] [sustained bandwidth =
            total bytes/response time]

    Exit from program
```