A Web Service for evaluation of load balancing strategies for distributed web server systems

Digvijay Singh Lamba
Senior Undergraduate,
Dept. of Computer Sc. and Engg., IIT Kanpur

Dr. Pankaj Jalote
Dept. of Computer Sc. and Engg., IIT Kanpur

Dr. Dheeraj Sanghi
Dept. of Computer Sc. and Engg., IIT Kanpur

Abstract

Many large web sites get millions of hits everyday. They need a scalable web server system that can provide better performance to all the clients that may be in different geographical regions. Common approach to improve performance is to have fully replicated web server clusters in different geographical regions with replicated servers. In such an environment, one of the most important issues is selecting a server for servicing a request. Client requests should be directed to a server such that the time taken for servicing the request can be minimised. Different policies are possible for server selection and it is difficult to determine the impact of different policies.

In this paper, we describe a web service that can be used to evaluate the performance of different schemes. We provide a web based testbed to perform this task. This service utilizes a dedicated testbed that emulates the World Wide Web and on which the performance of various strategies can be studied. A user gives the values of different network and load parameters and the specifics of the policy. The testbed is configured to emulate a web server system using the specified policy and performance testing is done. Data from the tests is collected, analyzed and graphically displayed to the user. The service also allows new approaches to be tested.

Introduction

Number of users accessing the Internet is increasing quite rapidly and it is common to have more than 100 million hits a day for popular web sites. For example, netscape.com website receives more than 120 million hits a day. The number of users is expected to continue increasing at a fast rate and hence any website that is popular, faces the challenge of serving very large number of clients with good performance. Full mirroring of web servers or replication of web sites is one way to deal with increasing number of requests. Many techniques exist for the selection of the nearest web server from the client's point of view. Ideally, selection of best server should be done transparently without the intervention of the user.

Many of the existing schemes do only load-balancing. These schemes assume that the replicated site has all the web servers in one cluster. This is alright for medium sized sites, but beyond a certain amount of traffic, the connectivity to this one cluster becomes a bottleneck. So large web sites have multiple clusters, and it is best to have these clusters geographically distributed. This changes the problem to first select the nearest cluster and then do load balancing within the servers of that cluster. Of course, if all servers in a cluster are heavily loaded then another cluster should have been chosen. So the problem is more complex in such an environment.

Designing such a system involves making decisions about how the best server can be selected for a request such that the user gets a response in minimum time and how this request is directed to that server. In some strategies, a server is selected without taking into account any system state information, e.g. random, round robin etc [6]. Some policies use weighted capacity algorithms to direct more percentage of requests to more capable servers [7]. Some strategies select a server based on the server state [7] and some others take client state information into account [7]. There is always a trade off between the overhead due to collection of system state information and performance gain by use of available state information. If too much state information (of server or clients) is collected, it may result in high overheads for collection of information and performance gain may not be comparable to overheads.

The performance of any load balancing approach depends on a host of features like network delays, packet losses, transmission errors, rate of requests, server load etc. It is usually very hard to analytically determine the performance of a policy given some conditions. Simulations also have limitations in that they can only take limited variables into account and with the complexities involved in this case effect of all the variables cannot be analytically determined and simulated. Hence testing for performance by setting up a testbed is a reasonable way to evaluate different strategies.

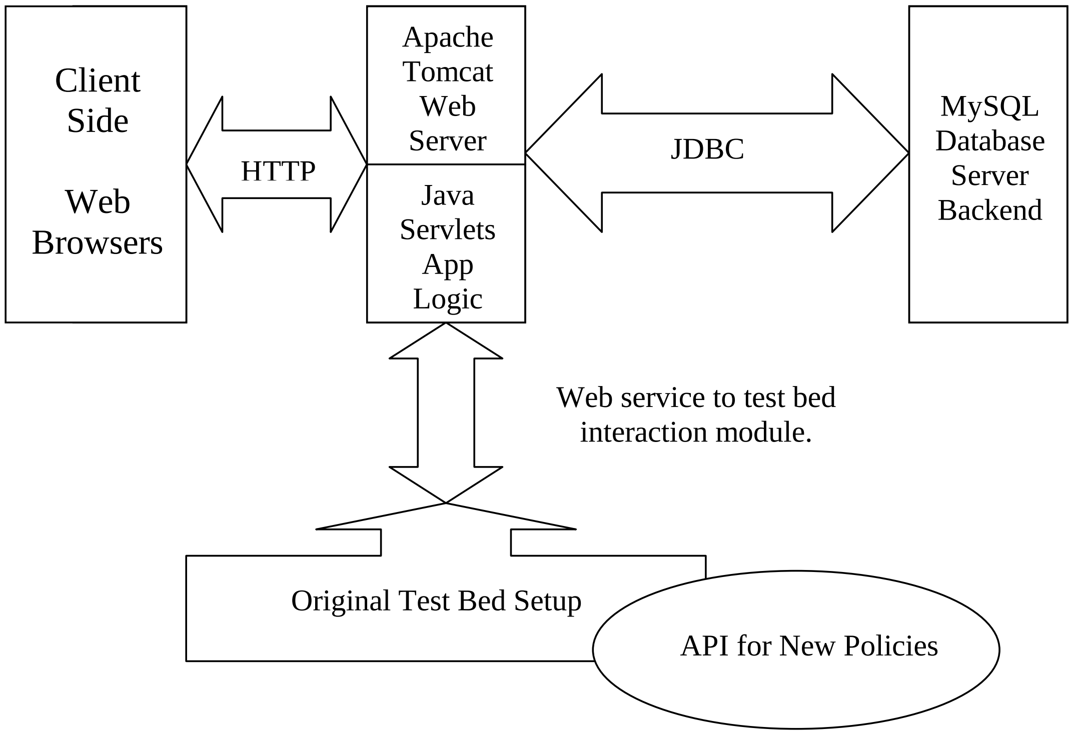
This testing will in general require a dedicated testbed on which performance studies can be done. Such a testbed should be configurable for different strategies and network characteristics. As such a testbed is likely to be a dedicated set up with a focussed purpose, it will be useful if a web service can be created for evaluating a policy. In this, all parameters will be set in the web service, which will then drive the testbed to obtain the results. Such a service will make a dedicated testbed accessible across the world. Such a service should fulfil a number of requirements. It should be possible to study the impact of different parameters on the performance of various strategies and to compare them. It should be possible to submit new strategies and compare their performance with existing ones or to identify conditions where the policy performs best.

In this paper, we describe a web service for evaluating load balancing strategies for distributed web server systems. The web service interacts with a testbed to automate the process of submission of parameters, testing and result generation to allow the user to test and compare load balancing policies through in a highly configurable manner. We have pre-defined some popular strategies which can be evaluated for different parameter settings. The service also allows users to test new policies for load balancing and compare their performance with that of existing ones on a variety of parameters and under various conditions and settings.

System Design and Architecture

The entire system consists of three main components that interact to provide the previously described services.

- The Test bed at the back end is a physical set up which can be configured to simulate a variety of network conditions, load conditions, server architectures and load balancing policies.
- The web service at the front end is where the interaction with the users takes place and which automates the process from submission of test parameters by the user, configuring the test bed to follow those parameters, running tests and display of results to the users. It includes the web server, application logic, database server and the interface to the testbed.
- Also an API has been designed and implemented using which new policies can be submitted for testing on the testbed. It includes standard libraries (API's) using which a set of interfaces have to be implemented by the user according to the policy he is submitting.

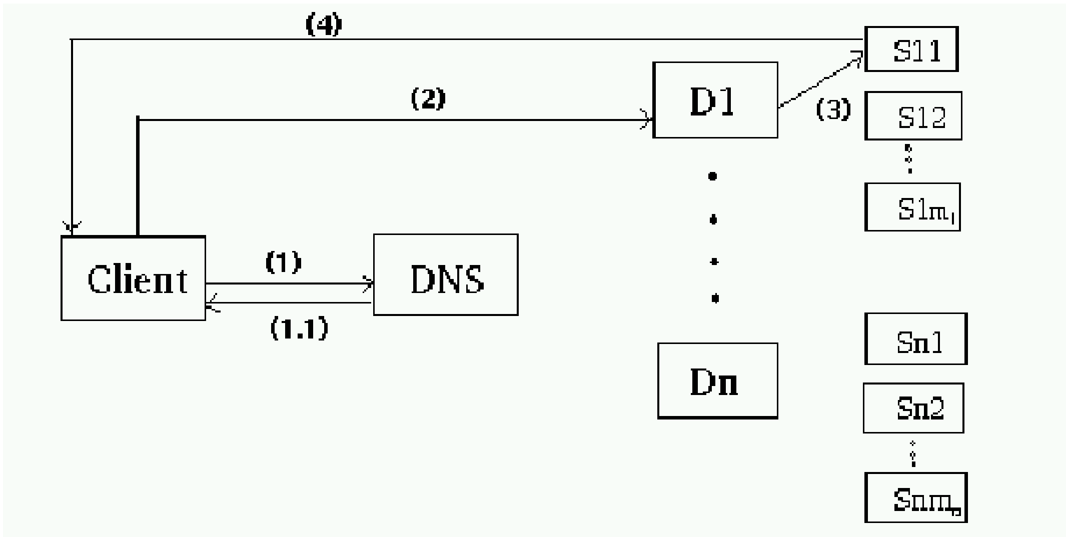


We will now describe each of these parts and their implementations one by one. The testbed we are using at the backend is from [1]. It has however been modified for our purpose and these modifications are included.

The Original Test Bed Setup

The load balancing mechanisms have their relative pros and cons and it is not easy to demonstrate the superiority of the one over another. To compare various policies for request distribution at server side, a test-bed was designed and implemented by Puneet et al [1] which tries to emulate real network scenarios and implements a highly configurable web server system which can be configured to use a variety of load balancing policies. All standard components used in the Internet are used in this test-bed, for example, BIND (Berkeley Internet Domain Name Server) for DNS and Apache web server. WebStone was used for generating HTTP requests [4]. WAN delays and bursty packet losses which are common on Internet links have been modelled using Nistnet Software[5]. The test-bed can not be used to evaluate client based approaches. It can, however be used to evaluate the performance of all other categories of load balancing approaches.

The Web Server System of the testbed

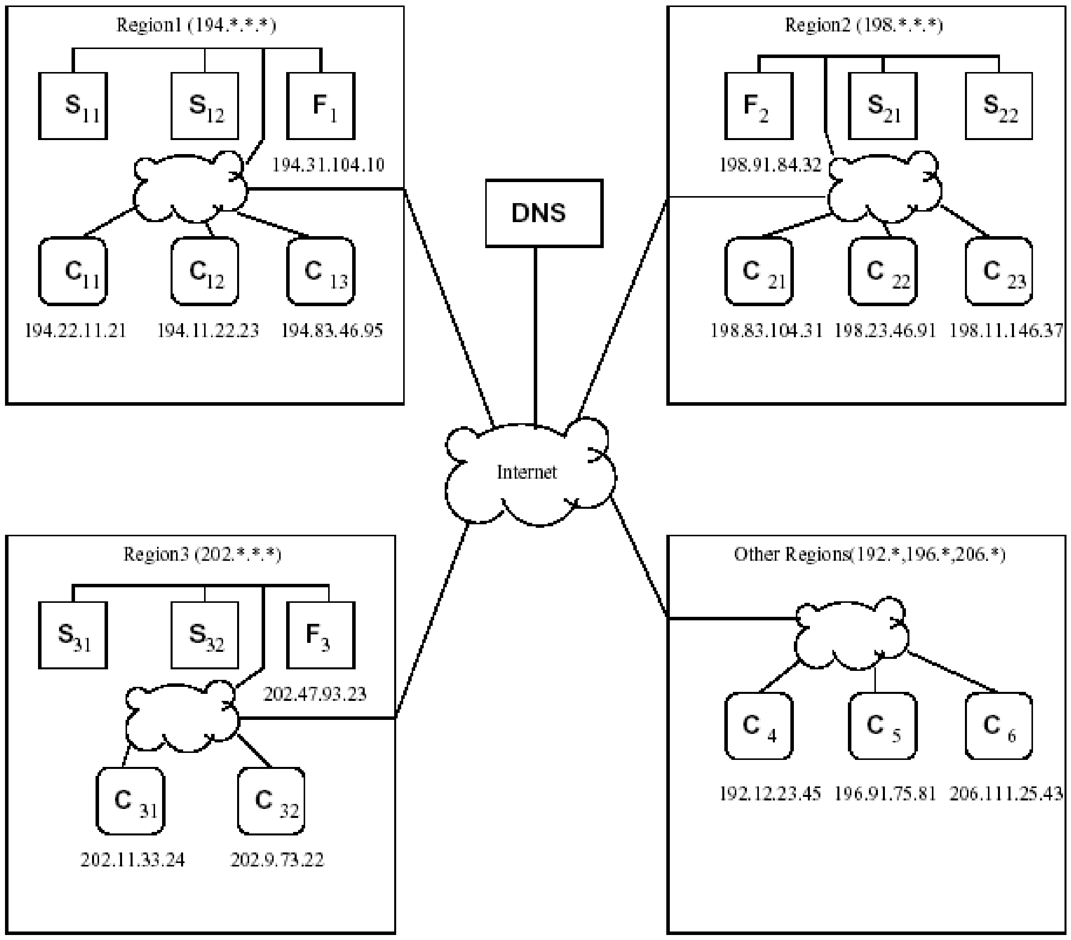


To permit implementation of various policies, a general architecture was chosen as shown in Figure for the web server system. In this architecture, the replicas are organized as a set of clusters, each cluster having a front node and some servers. A request is directed to a server in two steps. In the first step, the request is directed to one of the clusters by the DNS by returning the IP address of the front node of that cluster. In the second step, the client sends the request to the front node of that cluster, which decides the server that should serve the request, and forwards the request to that server. The selected server then services the request.

This architecture allows the implementation of both DNS based and dispatcher based strategies as well as combination of the two. It permits modelling the situation where servers are located in different geographically locations to provide better service to customers in different regions. It can also model single server systems easily.

Experimental setup of the testbed

The web service has been designed as an extension to the set up described in Puneet et al [1]. It had three clusters on different logical networks representing three different geographical regions. Each cluster had one front node and two servers. In the original setup, there were eleven clients to generate requests. A DNS was also setup to resolve IP addresses of clusters. Actual test-bed setup used for performing experiments is shown in the figure below. For more details please refer to [1]37-44.



The Entities

Before we start to explain the service we would enumerate and explain the various entities used in the explanations. These include two kinds of users and three kinds of entities that are maintained per user in the testbed.

1. **Policy owner.** A policy owner is a user who either submits a new policy or runs tests by configuring existing policies. The tasks of the policy owner are displayed in the following diagram.

2. **Administrator.** An administrator is a user who is responsible for the backend test running process, for implementing new policies, collection of results etc.

3. **Policy.** A policy is a load balancing strategy that is implemented on the testbed and can be used as a base strategy on which policy owners can define tests.

4. **Submission.** A submission is a newly submitted strategy that is not implemented on the testbed for general use. A submission goes through several stages before becoming a policy.

$$\begin{tikzcd}\boxed{\text{NOFILE}} \arrow[r] & \boxed{\text{FILES}} \arrow[r] & \boxed{\text{SUBMITTED}} \arrow[r] & \boxed{\text{IMPLEMENTED}}\end{tikzcd}$$

   Initially the policy owner defines a new submission specifying its name, its type (at DNS or at Front Node) and a description for it. At this stage the policy has the status “NOFILE”. Later the policy owner uploads the implementation files for the submission. The status then changes to “FILES”. For all submissions with status “FILES” administrators have an option to download the implementation files. After the download the status changes to “SUBMITTED”. The administrator will then test and implement the policy on the testbed and make it available to the users. At this stage the status changes to “IMPLEMENTED” and the submission becomes a policy.

5. **Test.** A test is a combination of a policy and a set of parameters. A test is a representation of one set of tests to be done by policy owner. A base policy is defined for the test and several parameters can be configured. This is how a policy owner requests running of tests on the testbed. The administrators download tests and run them on the testbed. A test will follow the following stages.

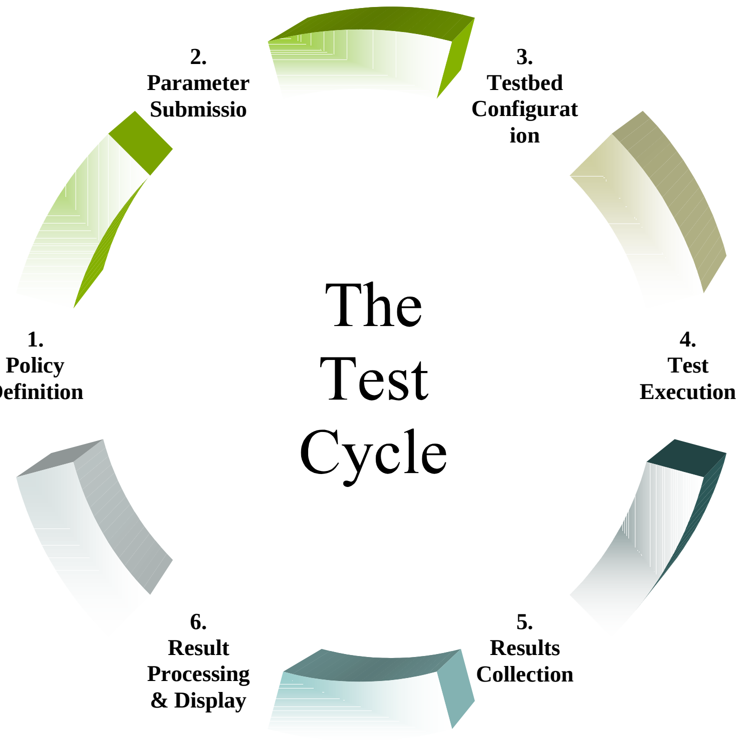
$$\begin{tikzcd}\boxed{\text{NOPARAMS}} \arrow[r] & \boxed{\text{PARAM\_DONE}} \arrow[r] & \boxed{\text{SUBMITTED}} \arrow[r] & \boxed{\text{RESULTS}}\end{tikzcd}$$

   Initially the policy owner defines a test with a name, a base policy and a description. The status is “NOPARAMS”. Later he specifies the parameters for the test. After submission of parameters the status is “PARAM_DONE”. For all tests where the status is “PARAM_DONE” the administrators may download the script files for configuring the testbed or automatically transfer them to the testbed for configuration. Once the files are downloaded the status is “SUBMITTED”. The administrator then runs the test on the testbed and on completion the results are uploaded on to the web service. The status now becomes “RESULTS” and the test is considered complete.

The Test Cycle

The main function of the web service is to drive the testbed described above, automate it and allow easy testing and result browsing. In this section we will describe what the web service can do in terms of what we call the test cycle.

There are various tasks that a web service of this kind can attempt to perform. We will however limit the scope of our service to the implementation of the test cycle. The test cycle is a cycle of events starting from user input and ending with the user receiving the results of the tests. These events are a sequence of tasks that must be performed in order to configure and run the testbed automatically. At each step of the cycle several design decisions must be made and the same will be outlined. Detailed description of each decision will follow.



The test cycle consists basically of the following parts:

1. Policy definition.

Here the user defines which policy it is that he wants to test. He may do this in one of the following ways:

- ***a. Use existing policy.***

  He may use any of the policies that have been implemented on the test bed. For this he defines a new test and chooses any of the existing policies as the base policy for that test.
- ***b. Define a new policy.***

  Alternatively the user may have come up with a new policy that he may want to submit to the test bed and run tests on. In that case this task will include the submission of the policy by the user and its implementation on the test bed.

2. Parameter Submission.

The user after having defined a test defines several parameters that define how the test bed will be configured for that test. These parameters include the following:

a. ***Test parameters***. These are used to configure the running of the test.

b. ***Network parameters***. These are used to configure packet delays, losses, bandwidth limitations etc.

c. ***Generic test bed parameters to define test bed configuration***. These specify which clusters and servers are to be used in the test.

d. ***Policy specific parameters***. Some policies may be configurable based on certain parameters. This is where such parameters may be defined.

3. Test bed configuration.

In this phase based on the parameters defined earlier the test bed is configured. This includes generation of appropriate scripts and configuration files that follow the parameter settings defined earlier. These scripts and configuration files are then transferred to the appropriate machines on the test bed. The service then executes required scripts and commands on each machine so as to configure it appropriately.

4. Test execution.
In this phase the tests are actually run on the test bed. This is started after the machines are appropriately configured. At the end of the configurations the test bed is in a state that automatic rebooting of all the machines will automatically start the tests and have them run for the specified time.

5. Results collection.
The tests end with various test parameters being observed and performance parameters as seen by the clients being measured. In this phase these results in the raw form are collected on to a central test bed machine which is pre specified. At the end of this phase all needed results are available at a central location in the raw format.

6. Result processing and display.
In the final phase the centrally collected raw data is studied and processed. At the end of the processing a variety of results are available in the graphical format. These graphs are then shifted to the web server and made available for viewing by the user. Thus at the end of this step the results of the test are available for the user to study.

These six steps make up the test cycle and in summary completely define the functions of the web service. Each of the steps, however, is made up of several sub-steps. We will now describe each of the steps one by one in detail.

Policy Definition

This is the first step of the test cycle where the policy owners define the base policy on which they wish to run the tests. There are two possibilities.

1. **Use one of the existing policies for test definition.**
   The policy owner defines a new test and chooses one of the existing policies as the base policy for the test.
2. **Define a new policy for test definition.**
   The policy owner has a new strategy whose performance he wishes to evaluate. In this case the policy owner first submits his new policy in the defined manner and waits for it to be implemented on the test bed. The testing and implementation of new policies is done offline by the administrators. The details of how a new policy is to be submitted are explained later in the paper. Once this policy is implemented on the testbed the policy owner may use it as the base policy for a new test as described above.

Submitting a new policy

The most important part of the web service is its ability to accept new policies. The load balancing policies of today are far from optimal and new strategies are likely to develop in the future. The developers of such policies need a platform to develop and test their policies. Such a platform should be flexible enough to allow a variety of ideas to be implemented and tested and there should be provisions to evaluate the performance of the policy being developed against different parameters. This will allow developers to recognise the grey areas of their strategies and find out under what conditions the policy performs good and under what conditions the policy does not do so well. Also comparisons with existing policies will allow judgement of where the new policy stands as compared to existing ones.

The web service has been developed to provide such a platform. All the above functions are available on the service. We will now describe how a developer can submit his policy to the web service. Before we begin, however, we would like to mention that currently the web service is at a very early stage as far as submission of new policies is concerned. A lot of development to improve this particular feature is possible. However the current implementation does allow fulfilment of all the above goals as we shall see.

We shall begin by briefly describing how a new policy can be submitted. We have adapted the testbed such that the policy followed depends entirely on one of the services running on the DNS (or Front Node) machine. This service interacts with the testbed using network sockets. Thus, in order to implement a new policy it is only this service which must be stopped and restarted with the changed executable. We have also completely implemented the interaction of this service with the main DNS (or Front Node) service. Also implemented is a system of load measurement at various clusters (or servers) and an API to access this load information. The developer of the new strategy has only to download the API and the code for the service we mentioned earlier. He then has to implement exactly one method according to his policy. That will be enough to implement the policy on our testbed. Let us now discuss this process in detail.

Limiting the policy domain to work on

To assume that any and every random policy designed by anyone will automatically fit with the testbed is very unrealistic. We need to define exactly what kinds of policies can use this service effectively. For example if a policy needs large scale changes or recompilation of the DNS and/or the kernel such a policy will not fit it in to our service. We define our domain of working as the set of all policies that can be effectively and easily configured on our testbed without any changes to the testbed itself.

The domain of working D is defined as:

1. All DNS based policies where the decision by the DNS regarding which cluster will serve a clients request depend on no more factors than the following:
    - Client id and load generated by this particular client.
    - Request rate from the client.
    - RTT from the client.
    - Geographical proximity of the client.
    - Cluster load information in terms of:
        - Number of nodes in the cluster.
        - Current rate of connections.
        - Average rate of connections.
        - Number of active connections.

        - Estimated max on connections.
        - Number of distinct clients connected.
        - RTT from the server.
        - Geographical proximity from the server.

2. All front node based policies where the decision by Front Node regarding which node will serve the next request is taken based on no more than the following factors:
    - A load information from each node in the cluster exactly same as the cluster load information above in 1 b.
    - Information regarding the client and the load generated by it similar to the information above in 1 a.

Any policy that mixes the above two approaches and uses both the DNS and the Front Node is not currently supported. The two modules as of now MAY NOT interact with each other directly but this can be supported in the future versions.

Any policy that uses parameters other than those provided above is not currently supported. The testbed may however be enhanced to provide more parameters so that better policy decisions may be made.

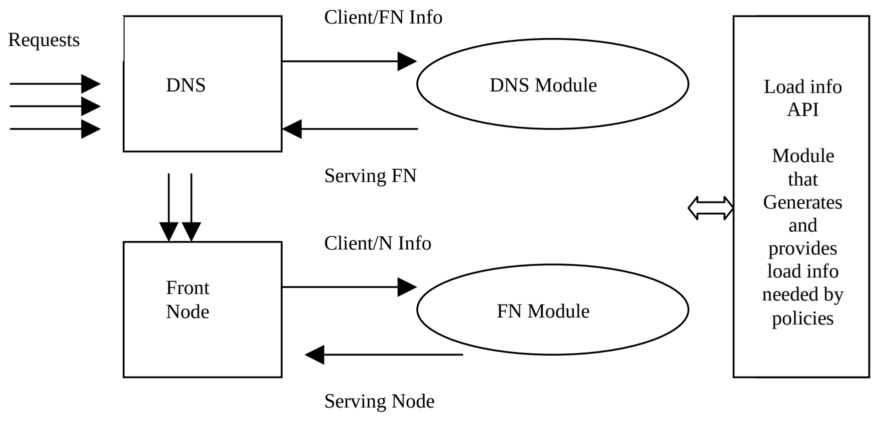
Design of framework for new policy submission

The following diagram explains how the framework works in general. A policy is implemented as a DNS module or a Front Node module. The DNS and the front node request dispatcher code have been altered to do the following;

1. They first look if a new policy is available.
2. If yes, they provide the client load information to the new policy's module and wait for a response on where to route the client's request.
3. If no or if the response of the module above has an error it used its default policy for the request and logs the same. This ensures that a bug in the policy's implementation does not hang the system.

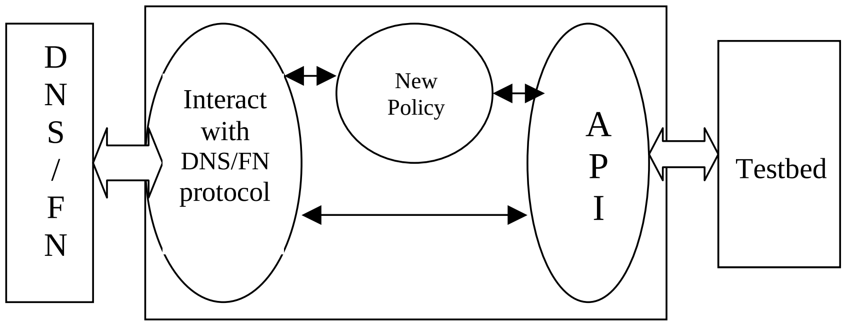
Thus the DNS (or Front Node) does not directly make its own decision for the question, "Who will service this request?" Rather this is left to the appropriate module. All interaction with the modules takes place using TCP/IP protocol and is implemented using network sockets. This ensures that changing policies do not need the whole DNS (or Front Node) to be recompiled.

The protocol followed in the interaction between the DNS (or Front Node) and the policy module has been pre-defined and implemented. As we shall see in the discussion of the module's design, the knowledge of this protocol is not necessary for the developer of a new policy. He simply uses our implementation of the protocol. The protocol basically consists of the DNS (or Front Node) requesting the name of a cluster to service a particular request and sending the IP address of the client that generated that request. Thus DNS sends the clients IP and receives the IP of the Front Node which will receive the request. The Front Node at a lower level simply asks the service for the best server and gets the IP address of a server according to the policy to which the request is then forwarded.



Architecture of the DNS/Front Node Modules

The DNS (or Front Node) module is where the main implementation of a DNS based policy is done. However, this module consists of several sub modules and interacts with the DNS (or Front Node) on one side and with the API for retrieving load information on the other. We will now discuss the architecture of this module and see what parts are already implemented and what still need to be implemented.



DNS/FN MODULE

The DNS/FN modules basically consist of three parts.

1. A sub module that implements the interaction with the DNS/FN.
2. A sub module that behaves like an API for the developers of new policies and interacts with the testbed to retrieve load status and other data needed to make intelligent best server decisions.
3. A sub module that is policy specific and needs to be implemented separately for every policy. It is this module that has to be implemented by anyone wishing to submit a new policy to our service.

Of the three modules the first two have been fully implemented by us. The first one for interaction with the DNS (or Front Node) calls the policy specific module when the DNS (or Front Node) requests for a cluster (or Server) to service a client request.

The method called in case of DNS is “select_cluster” with the following declaration:

```
/*
* Returns: index in DNS RRSET for selected cluster IP.
* Parameter clientip: the ip of the client requesting connection.
*/
uint16 select_cluster(struct in_addr clientip);
```

The method called in case of Front Node is “select_server” with the following declaration:

```
/*
* Returns: index in DNS RRSET for selected server IP.
*
*/
uint16 select_server(void);
```

The policy implementer is required to implement these methods in a file “select_cluster.c” or “select_server.c”. The main policy will be implemented here though he may include additional methods and files. For deciding on which cluster to return and to find what the DNS RRSET or Server RRSET for the chosen cluster is the implementer may make calls to the API module.

The API module is already implemented and contains functions and data structures that may be used to maintain and access information about load, request rate from clients etc that is maintained by the testbed. This information can be used to in deciding which server should service a particular request. The architecture of the API modules is different for DNS and Front Node based policies as different information is available at the two levels. The two API modules are described next. The complete API is explained in Appendix B.

Architecture of the DNS API sub module

The testbed implemented in [1] has a built in scheme to measure and maintain several variable that show the load state of the servers and request rate of the clients. For more information on the implementation of the same please refer [1]26-35. The DNS API module provides methods to access these variables for implementation of new schemes.

The following variables are maintained by the testbed that can be accessed by the DNS level policies:

1. Following information is recorded for clients having request rates above a threshold:
    a. The request rate from the client.
    b. Round trip to time to a few (say 3) nearest and not loaded clusters.
2. Following information is recorded for each clusters.
    a. IP address of the cluster.
    b. DNS RRSET value of the cluster.
    c. Load info latest or not.
    d. Cluster load information.
    e. Number of servers in the cluster.
    f. Number of servers with latest load info.
    g. Requests served in the last minute.
    h. Bytes transferred in the last minute.
    i. Number of distinct clients being served.

j. Number of connections in the last time quantum.
k. Average number of connections in the past.
l. Total number of active connections.
m. Number of connections that can be served at moderate load.
n. Maximum number of connections that can be served.

3. Following overall state information is maintained.
    a. Total number of clusters.
    b. Total number of servers.
    c. Total number of clusters with latest load info available.
    d. Overall average request rate.
    e. Total free capacity of the system at moderate load.
    f. Total maximum capacity of the system.

All the above can be easily accessed by the API which provides methods to access these. The maintenance of this data is done by the testbed and need not be done by the user. The user may however use functions in the API to request some information maintenance tasks like requesting latest information for nodes or setting threshold for storing client request rates etc.

The functions in the API that are used for the above purpose can be seen from the description of the API in Appendix B.

Architecture of the Front Node API sub Module

As mentioned earlier the testbed maintains load information for the servers [1]35. The Front Node API module provides methods to access these variables for implementation of new schemes.

The following variables are maintained by the testbed that can be accessed by the Front Node level policies using the Front Node level API:

1. Following information is recorded for clients having request rates above a threshold:
    a. The request rate from the client.
    b. Round trip to time to the servers in the cluster.
2. Following information is recorded for each server.
    a. IP address of the server.
    b. DNS RRSET value of the server.
    c. Load info latest or not.
    d. Number of connections in the last time quantum.
    e. Average number of connections in the past.
    f. Total number of active connections.
    g. Number of connections that can be served at moderate load.
    h. Maximum number of connections that can be served.
    i. Load info from sysinfo command of unix.
    j. Average % cpu utilisation.
    k. Detailed load information from /proc/stat
    l. Number of requests served in last minute.
    m. Bytes transferred by the server.

3. Following overall state information for the cluster is maintained.
    a. Total number of servers in the cluster.
    b. Total number of servers with latest load info available.
    c. Total number of connections in the last time quantum.
    d. Average number of connections per unit time served.
    e. Total number of active connections.
    f. Total free capacity of the cluster at moderate load.
    g. Total maximum capacity of the cluster.
    h. Load information for the front node from sysinfo command.
    i. Load information for the front node from /proc/stat
    j. Total number of requests.
    k. Total number of bytes transferred.

All the above can be easily accessed by the API which provides methods to access these. The maintenance of this data is done by the testbed and need not be done by the user. The user may however use functions in the API to request some information maintenance tasks like requesting latest information for servers etc.

The functions in the API that are used for the above purpose can be seen from the description of the API in Appendix B.

Thus we see how using the API and implementing the said functions a user can implement his own policy and submit it on the test bed for testing and evaluation. We have used the scheme above to submit the standard policies as well as one non standard policy called the nearest server policy [1]52 to the testbed. These schemes both at DNS and Front Node level are to serve as examples for future submissions. They also serve as a test for the flexibility of the schemes.

Parameter Submission

After the policy owner has defined a new test and specified a base policy for it, which may be his new strategy or any of the old ones, he defines the parameters for his test. This is where the user may configure the testbed to simulate particular network conditions, configure the physical setup of the test bed and specify how the testing is to be done in terms of generated load, number of tests etc. Such testing can be used for two main purposes.

a. By varying parameters across tests the user can determine the effect of particular parameters on his policy. This may be further used to determine conditions under which a policy performs well and conditions under which it does not.
b. By testing across a number of policies for the same parameter configuration the user may compare the performance of different policies under the same conditions.

Current implementation allows following parameters to be specified and configured. For any parameter a default option is provided and implemented.

1. Testing parameters.
    These parameters are used to control the running of the test.
    a. *Number of iterations per data point.* The default is 10 iterations.
    b. *Time duration of each iteration.* The default is 1 Minute.

    c. *The requested file size.* This can be set to 5KB or 50KB currently. The administrator may change the sizes or add more variation easily. The default is KB
    d. *Data points for testing.* Every data point specifies a different request load. The user may choose to test with number of clients generating load between 10 and 120 in multiples of 10. For ex. At (10,30,50,70,90,110) or at (20,40,60,80,100,120) etc.

2. Network parameters.
   Here the policy owner may specify how the network will behave. This is done by attaching two behaviours to one or more of the testbed machines. The behaviours attached to each machine will affect all the incoming packets to that machine. One of the behaviours is for packets from nodes in the same cluster and the other behaviour is for packets from nodes outside this cluster.

   The following parameters may be specified for behaviours.
    a. Mean Delay time in milliseconds.
    b. Correlation between delay times of successive packets.
    c. Standard Deviation of delay times in milliseconds.
    d. Percentage of packets to be dropped.
    e. Correlation between successive packet drops.
    f. Percentage of duplicate packets.
    g. Correlation between successive packets being duplicated.
    h. Bandwidth limit in bytes/sec.

   One may also specify default network settings for any or machines. In the default setting following network configuration is implemented.

   In the default setting smaller delays are introduced for IP packets sent and received between clients and servers in the same geographical region and relatively higher delays for packets between clients and servers in different geographical regions. These delays are generated randomly within specified range (say, 10-50 ms round trip delay in the same region and 50-250 ms delay across the regions).

   Similarly we have lower packet losses with higher correlation between drop of packets to model bursty lower packet losses in small distance links for links in same geographical region and higher packet losses with high correlation between successive packet drops for links across different geographical regions (e.g. 5% loss with .9 correlation on links in same region and 10% loss with 0.85 correlation on links connecting different regions).

3. Physical testbed parameters.
   Here the user may specify the physical configuration of the testbed. This includes specification of which clusters are to be online for the test and which are to be switched of. Also within clusters which servers are to be online may be specified. Same holds for clients. This is done by choosing appropriate flags for each machine. Note that if the Front Node is switched off the cluster is automatically switched off.

Testbed Configuration

After the user has specified parameters for his test, the status of that test changes to "PARAMS_DONE". All tests with status "PARAMS_DONE" are presented to the administrators with the option of submitting these tests to the testbed. Currently this can be done in two ways. The administrator is presented with the following options.

1. The administrator can download the scripts as a zip file and manually load them on to the testbed.
2. The administrator can ask the web service to automatically transfer the scripts to appropriate machines on the testbed.

In either case the configuration includes generation of scripts, transfer to the testbed and execution of the scripts on the testbed. At the end of these steps the testbed will be configured and ready to run the appropriate test. These steps are explained in details now.

1. Generation of scripts.
   Following scripts are generated by the web service. These include scripts for the three classes of parameters specified earlier and for configuration of base policy on the testbed. Examples of each script are in Appendix A. In case of

   a. Configuration of base policy.
      The DNS (or Front Node) machine is configured such that the decision of which cluster (or server) should service a request is made by a service different from the DNS (or Front Node). When a policy is implemented an appropriate version of this service is compiled and kept at the DNS (or Front Node) machine. This script contains a command to run the appropriate executable for the base policy as the decision making service.
   b. Configuration of test parameters.
      This is a set of scripts for the Webstone root machine which configures the Webstone load generator and tester appropriately. These include the conf/filelist and conf/testbed scripts. The examples of these scripts are in Appendix A.

   c. Configuration of network parameters.
      For this a script is generated for every node in the testbed. Each script sets two kinds of behaviours for a machine. One for packets from machines from the same geographical region the other for packets from the machines from a different geographical region. Default scripts are pre-generated. These scripts are named "delay" and are to be kept in /root directory of the machine.

   d. Physical Configuration of the testbed.
      For this we change the default DNS configuration file for BIND on the DNS machine. The testbed is set up in such a way that even if a cluster/server is up and sending load updates, if its IP address is not in the DNS configuration file, the IP is not resolved. Thus by simply changing the DNS configuration machines can be included or excluded from the testbed. Also for completely automated testing machines are rebooted after configuration (as explained later). Automated rebooting

is controlled to exclude the machines that are switched of in the parameters.

e. Configuration of webstone root.
The webstone root machine's rc.local script is edited to include commands for starting the webstone test. This command is according to the policy being run and the data points at which we want to test. This command executes the "run" script in the /root directory of the webstone root. This script contains commands for the appropriate data points. This script is also generated by our web service.

2. Transfer to the testbed.
This can be done either automated or manually. When automated the testbed needs to be dedicated and all machines need to be up and running. Appropriate scripts are transferred using FTP protocol to the correct location of correct machines. In manual case a zip file is downloaded by the administrator which contains all the scripts and a file "whereto.txt" which specifies where each script should be placed.

3. Execution of scripts on the testbed.
In order to automatically execute the configuration scripts so as to configure the testbed appropriately before running tests we have edited the "rc.local" scripts of all machines to include commands that execute the scripts. Thus rebooting the machines will automatically configure the testbed.

Test Execution

This is a comparatively simple procedure. The "rc.local" files of all machines also have appropriate commands to automatically start the services and executables for running the test. Thus when after placing the configuration scripts either manually or automatically the machines in the testbed are rebooted, the test execution starts. This automation includes executing appropriate webstone commands.

Current implementation requires this step to be done manually. However, in case of a dedicated testbed, this may be automated. The automation will also need implementation of test scheduling and handling of simultaneous test requests and other such features.

Result Collection

After the tests finish, results are collected from all the client machines and merged at the webstone root. At this stage we have results separately for each data point. The results include the following information (as the example below) for iterations at each data point.

```
-----------------------------------------------------------------------------------------
WEBSTONE 2.5b3 results:
Total number of clients:        100
Test time:                      1 minute
Server connection rate:         196.57 connections/sec
Server error rate:              0.40 err/sec
Server thruput:              8.49 Mbit/sec
Little's Load Factor:        96.87
Average response time:       0.493 sec
Error Level:                 0.20 %
Average client thruput:      0.09 Mbit/sec
Sum of client response times: 5812.40 sec
Total number of pages read:  11794

11794 connection(s) to server, 24 errors

                        Average    Std Dev    Minimum    Maximum

Connect time (sec)      0.174289   0.646997   0.008103   9.149462
Response time (sec)     0.492827   1.254416   0.045947   30.167284
Response size (bytes)   5396       0          5395       5396
Body size (bytes)       5120       0          5120       5120

60385280 body bytes moved + 3252653 header bytes moved = 63637933 total
-------------------------------------------------------------------------------------------------
```

Thus at the end of results collection we have one log file for every data point. Each log files containing information similar to above for every iteration. This is what we call raw data.

Result Processing and Display

At the end of result collection, the test results are found in the raw format as discussed earlier. However, for analysis and presentation we need to merge the data for different iterations at a data point as well for different data points and process it in a manner that it is suitable for display. In this step the data above is processed over a number of sub steps and finally made available graphically.

The following sub steps are undertaken to process the data:

1. All the log files for a test are analysed and data collected and stored in one file.
2. The log file generated above is analysed and data across iterations is averaged and the average values for each data point are stored.
3. The average log file is used along with "gnuplot" to out put graphs in the *.fig format.
4. This format is converted to image files for display on the website.

At the end the following graphs are output for any particular test:

- Average Response Time vs. Number of Client Processes.
- Maximum Response Time vs. Number of Client Processes.
- Connection Rate vs. Number of Client Processes.
- Total Throughput vs. Number of Client Processes.

It is also possible to merge the same to graphs from any two different tests for comparison between them. Also the *.fig files may be downloaded for any test for further analysis by the policy owner.

Implementation

We have explained the detailed design and architecture of the web server system and some basic implementation issues. In this section we will probe deeper into how exactly the Web Service has been implemented.

The architecture of the web service was described earlier. The platform for implementation was also shown in the figure. We shall in brief describe it here. The web service uses Jakarta Tomcat Web Server and implements the application logic using Java Servlet Technology. The database at the backend is MySQL and the connection is using JDBC. Most of the web service is thus written in Java language.

The interaction with the testbed is by generation of scripts, file transfer and development of API for new policies. The scripts depend on WebStone[5] for test configuration and Nistnet[6] for network configuration.

The following policies have been implemented successfully to demonstrate submission of new policies to the Web Service. We started with no policies implemented on the test bed and by submitting the following policies one by one, each of which have now been implemented. These are explained in [1]49-52

1. Random Selection at DNS.
2. Round Robin Selection at DNS.
3. Weighted Capacity Selection at DNS.
4. Nearest Server Selection at DNS.
5. Weighted Capacity Selection at Front Node.
6. Round Robin Selection at Front Node.

Conclusion

We have designed and implemented a Web Service for performance evaluation of load balancing strategies. This service is very flexible and can be used for evaluating existing as well as new policies. Users can evaluate the performance of a variety of policies under conditions defined by them. They can also submit new policies designed by them, evaluate their performance under various conditions and compare them to other policies.

This paper has described the design and use of the testbed in detail. It also describes how new policies can be easily developed and tested using our service as the platform for development. It describes the test cycle which is a series of steps necessary to run custom tests on policies and describes how the test cycle is implemented in the Web Service.

We have implemented a number of standard policies on the testbed as base policies for evaluation and comparison. These were developed using the new policy submission method described in the testbed.

We have also used the service for automated testing and demonstrated its ability as an extremely useful tool for further research in this area. We conclude that the service can be very useful for designers of new policies or users studying the performance and effects of existing policies.

Future Extensions

Several extensions are possible in the future. We shall enumerate some of them.

A high level language for specifying and submitting new policies can be developed. This will allow easy implementation of new policies and solve issues related to security and compilation problems of submitted files.

Complete automation of testing can be done by implementing a test scheduler for the testbed. In this the users can run tests in real time without any administrator interference. The issues to be resolved include simultaneous submission of tests and automatic scheduling and queuing of tests as the testbed can run only one test at a time.

The testbed currently implements maintenance of a particular set of load and request rate parameters from clients, front nodes and servers that can be used in choosing the cluster or server to service a request. A broader set of parameters here will allow a much broader range of policies to be tried on the testbed.

References

[1] Agarwal, P., May 2001. MTech Thesis, CSE, IIT Kanpur. "A test bed for performance evaluation of load balancing strategies of Web Server System"

[2] Crovella, M. E. et al., June 1995. Proceedings of the 3rd IEEE Workshop on the Architecture and Implementation of High Performance Communication Subsystems (HPCS'95) "Dynamic server selection in the Internet" http://www.cs.bu.edu/faculty/crovella/paper-archive/hpcs95/paper-final.ps.

[3] Mehmet Sayal et al, 1998. Proceedings of the workshop on Internet Server Performance. "Selection Algorithms for Replicated Web Servers". http://www.cs.wisc.edu/~cao/WISP98/final-versions/mehmet.ps.

[4] Trent, G. et al. February 1995. "WebSTONE: The First Generation in HTTP Server Benchmarking". http://www.mindcraft.com/webstone/paper.html.

[5]Nistnet software from National Institute Of Standards and Technology. http://www.antd.nist.gov/nistnet/

[6]Kwan, T. T. et al., November 1995. "NCSA's World Wide Web Server: Design and Performance". IEEE Computer, no. 11, 68-74.

[7]Cisco Systems Inc. "Distributed Director White Paper". http://www.cisco.com/warp/public/cc/cisco/mkt/scale/distr/tech/d_wp.htm

Appendix A

Testbed Configuration Scripts

A number of parameters can be specified in the web service for automatic configuration of the testbed. This automatic configuration is done using scripts that are generated, transferred to appropriate machines on the testbed and executed to configure those machines. In this appendix we give examples of these scripts.

1. Base policy configuration script.

For the DNS machine the following commands are included:

```
##shell script
#This line is modified depending on policy
POLICYNAME = rr
# copy appropriate script as DNS service
cp $POLICYNAME dns
```

2. Configuration of test parameters.

Following scripts are generated.

1. conf/filelist

```
#list of files with relative chances of being requested.
/file500.html          350          #500
/file5k.html           500          #5125
/file50k.html          140          #51250
/file500k.html           9          #512500
/file5m.html 1 #5248000
```

```
#inour case
/file5k.html #100
```

2. conf/testbed

```
### BENCHMARK PARAMETERS – according to submitted param
ITERATIONS="3"
MINCLIENTS="8"
MAXCLIENTS="128"
CLIENTINCR="8"
TIMEPERRUN="30"
```

3. Configuration of network parameters.
Different scripts are generated for each machine. Below is a sample for a script for a node. In this case the server S11. The script below is edited to provide a good idea of what is done.

```
#!/bin/bash
# Edit delay,drop and delsigma,drop_correlation limits for
# appropriate source addresses

#edit delay,drop range for same region machines
delay=15
delsigma=5
idrop=2
idrop_correlation=.9
#edit delay range for other region machines having clusters
delay1=150
delsigma1=50
drop=5
drop_correlation=.8

#region1 addresses
c11=194.22.11.21
c12=194.11.22.23
c13=194.83.46.95
#region2 addresses
c21=198.83.104.31
c22=198.23.46.91
c23=198.11.146.37
#region3 addresses
c31=202.11.33.24
c32=202.9.73.22
#region4 addresses
c4=192.12.23.45
c5=196.91.75.81
c6=206.111.25.43

dest=0

# Turn on the emulator
/sbin/insmod nistnet
/usr/local/bin/cnistnet -F
/usr/local/bin/cnistnet -u

/usr/local/bin/cnistnet -a $c11 $dest --delay $delay $delsigma --drop
$idrop/$idrop_correlation
/usr/local/bin/cnistnet -a $c12 $dest --delay $delay $delsigma --drop
$idrop/$idrop_correlation
/usr/local/bin/cnistnet -a $c13 $dest --delay $delay $delsigma --drop
$idrop/$idrop_correlation
/usr/local/bin/cnistnet -a $c21 $dest --delay $delay1 $delsigma1 --
drop $drop/$drop_correlation
/usr/local/bin/cnistnet -a $c22 $dest --delay $delay1 $delsigma1 --
drop $drop/$drop_correlation
/usr/local/bin/cnistnet -a $c23 $dest --delay $delay1 $delsigma1 --
drop $drop/$drop_correlation
/usr/local/bin/cnistnet -a $c31 $dest --delay $delay1 $delsigma1 --
drop $drop/$drop_correlation
/usr/local/bin/cnistnet -a $c32 $dest --delay $delay1 $delsigma1 --
drop $drop/$drop_correlation
```

Appendix B

The API for accessing state information of testbed

Here we outline the main data structures and a few functions that will highlight the kind of API that has been developed for the testbed. The complete API is a part of the Web Service and can be downloaded for development.

At DNS Level the following files, functions and data structures are available.

1. "clientinfo.h"

The functions and data structures defined here should be used to maintain and access information regarding the clients.

The main data structure is:

client_info_node: This is used maintain a list of clients, rtt to that client and other information about the client. Mainly about the rate of requests from the client.

```
/*
* Each client info node is part of two lists, one list is used to
* quickly locate client info, if already present and another to
keep
* track of classof request rate for this node
*/

typedef struct client_info_node_tag {
    client_info client;         /* Info about client */
    rttinfo info[NUMSERVSTOPROBE];/*server and rtt in microsec*/
    int t_lastprobe;            /* timestamp of last probe  */
    struct client_info_node_tag *hash_next;
    /* pointer to next client in hash table chained list */
    struct client_info_node_tag *req_next, *req_prev;
    /* pointer for next and prev node in doubly linked list
     * req_rate_queue sorted on num_requests */
} client_info_node;
```

The following functions should be used to access the client information.

print_client_info: Outputs the info on standard output.

```
void print_client_info(request_rate_info * prrinfoqueue);
```

search_client: Used to find a particular clients record. The node returned is described above. This is called for a client to find all info related to it.

```
/*
 * funtion search_client returns node for a client info, if
 * already present in hash table, otherwise returns NULL
 */
inline client_info_node *search_client(struct in_addr clientip,
                              request_rate_info * prrinfoqueue,
                              client_info_hash_table cinfotable);
```

2. “nodeinfo.h”

This contains data structures and functions to maintain and access information regarding various nodes.

node_info_node: Used to store information about a cluster. Each cluster is referred to as a node.

```
/* each node info node is part of two lists, one list is used to
 * Quickly locate node info, if already present and another to
 * keep track of class of request rate for this node */
typedef struct node_info_node_tag {
    struct in_addr node_ip;/* IP address of node */
    short updated;      /* Whether node has sent update or not */
    short id;        /* ID of cluster, i.e. index in DNS rrset */
    struct sysinfo info;        /* System load information  */
    uint16 numnodes;  /* Number of nodes in the system */
    uint16 nupdated;  /* Number of nodes that have sent update */
    uint32 nrequests;  /* Number of requests served in last min */
    uint32 nbytes;     /* Bytes transferred */
    uint32 nclients;   /* Number of distinct clients */
    uint32 num_conns;/* Number of conn handed over in last tick */
    uint32 avg_conns;/* Avg. No. Of connection in past */
    uint32 act_conns;/* Total number of active connections */
    uint32 avail_conns;/* Number of conns that can be served by
              server with moderate load conditions on server */
    uint32 max_conns;  /* Max. no. of more conns that can be
                      served by server in high load region */
    msg_rttrequest_t rttreq;  /* Message to be sent to get RTTs */
    struct node_info_node_tag *next;
             /* pointer to next node in hash table chained list */
} node_info_node;
```

find_node: This function is used to access load information for the cluster with ip node_ip.

```
/*
 * finds node for given ip or creates a new node if not found
 */
node_info_node *find_node(struct in_addr node_ip);
```

At Front Node Level the following files, functions and data structures are available.

1. “nodeinfo.h”

node_info_node: This data structure contains all the needed load information about a server for which it is meant.

```
/* Each node info node is part of two lists, one list is used to
 * quickly locate node info, if already present and another to
 * keep track of class of request rate for this node
 */

typedef struct node_info_node_tag {
    struct in_addr node_ip;     /* IP address of node */
    short updated;   /* Whether node has sent update or not */
    uint32 num_conns;/* Number of conn handed over in last tick */
    uint32 avg_conns;  /* Avg. No. Of connection in past */
    uint32 act_conns;  /* Total number of active connections */
    uint32 avail_conns; * No of conns that can be served by server
                         with moderate load conditions on server */
    uint32 max_conns;/* Max. number of more connections that can
               be served by server, server in high load region */
    struct sysinfo info; /*load information got from getsysinfo */
    statinfo_t sinfo; /*detailed sys load info from /proc/stat */
    float  avg_pcpuutil;/* Average % cpu utilization */
    uint32 nrequests;  /* Number of requests served in last min */
    uint32 nbytes;      /* Bytes transferred */
    struct node_info_node_tag *next;
             /* pointer to next node in hash table chained list */
} node_info_node;
```