Clustering and Server Selection using Passive Monitoring

Matthew Andrews, Bruce Shepherd, Aravind Srinivasan, Peter Winkler, Francis Zane

Abstract— **We consider the problem of client assignment in a distributed system of content servers. We present a system called *Webmapper* for clustering IP addresses and assigning each cluster to an optimal content server. The system is passive in that the only information it uses comes from monitoring the TCP connections between the clients and the servers. It is also flexible in that it makes no *a priori* assumptions about network topology and server placement and it can react quickly to changing network conditions. We present experimental results to evaluate the performance of *Webmapper*.**

Keywords— **Content distribution, server selection, passive monitoring.**

I. Introduction

As the popularity of Content Delivery Networks increases, many web pages are being stored in diverse geographical locations. Different users retrieve the pages from different locations depending on where they are located. However, when users make a web request they do not actually specify their location. Therefore a key component of any distributed web content storage system is a mechanism for determining where clients should obtain a given piece of content. A prime objective in the design of such a system is to minimize the expected latency for clients as a whole.

There are a number of problems with designing such a system. First of all, there are many different users in the Internet. It would be extremely difficult and expensive to keep track of users individually. Second, the content provider has no direct control over the client machine. It is difficult to tell the client to go out and measure the different content servers to determine which is the best. (Note that in our context, a content server could be a single machine or could consist of multiple web servers and caches in a single location in the network.) Third, we would like a client to be directed to a good server on its first request. When we receive the first request we do not want to have to perform many time consuming experiments with that client to determine which is the best content server. Finally, the Internet is changing. Machines are being added and removed all the time and the distance properties of the network are changing constantly.

We have built a system called *Webmapper* which addresses these problems. The main component of *Webmapper* is a clustering algorithm that groups clients according to their location in the network. We say that a group of clients form a *cluster* if, for any two clients in the group, their distances to the content servers are statistically indistinguishable. We wish to partition IP address space into a set of clusters that can each be defined by an IP address prefix.

Once we have defined a cluster and know the distance from that cluster to each content server, we are able to redirect clients in the cluster to a best content server. In order for this system to work we must be able to collect distance information. We do this by placing software on each of the content servers which measures incoming TCP connections from clients. This allows us to use many different notions of distance. For example, we can use the timing of the 3-way TCP handshake to give us a measure of the Round-Trip Time (RTT) to the client. Alternatively we could use packet loss as the metric for network distance. Third, we could measure how long it takes to download files from the content server to the client.

Once we have found the best server we still have to transmit this information back to the client. We do this in a standard way using the Domain Name System. We have a DNS front end to the *Webmapper* system which acts as the authoritative DNS server for some domain names. When a client makes a DNS request for one of these domain names the DNS server passes it to *Webmapper* which returns the address of the best content server. The DNS server can then return this address to the requesting client.

Distinguishing features of Webmapper

A key distinguishing feature of *Webmapper* is that it collects the distance and load information passively. All measurements are collected by examining TCP information as data is transmitted between clients and content servers in the course of normal communication. No active messaging is required to collect the distance information.

One problem with collecting the data passively is that once a client has been assigned to a content server we

The authors are with Bell Labs, Lucent Technologies, 600–700 Mountain Avenue, Murray Hill, NJ 07974, USA. E-mail: {andrews,bshep,srin,pw,francis}@research.bell-labs.com.

do not see data between that client and any other content server. Hence it makes sense to occasionally redirect a client to a new content server so that we can collect new data. However, we do not want to do this if it will cause the user to experience much worse performance. Our solution is to assign users based on a *testing index* that is derived from network distance but which also gives an added incentive to collect new information. This *testing index* is derived from the notion of *Gittins index* [9], which arises in the theory of stochastic processes. We also give different *Time-to-Live* values to each response depending on how likely we are to change the assignment of the client.

Another property of our system is that it adapts to changes in network distance between clients and content servers. This is a consequence of collecting the TCP information as described above. Changing network conditions are instantly reflected in the TCP information transmitted between clients and content servers. In addition, we discount old data over time so that new information carries more weight. A final feature is that our system does not require an *a priori* estimation of the location of content servers in the network.

Previous work

In this section we contrast our approach with other work aimed at clustering clients and mapping clients to content servers. An alternative notion of clustering was proposed by Krishnamurthy and Wang [13]. They use prefixes derived from BGP routing table snapshots as their clusters. They then show that these prefixes match up well with groups of clients that are under common administrative control. They validate their approach using two methods. First, they do reverse nslookups to see if the names of machines in their cluster are similar. They also use traceroute to see if the network paths to machines in the cluster match up.

Our system differs from [13] in the following ways. First, we have a different objective in clustering clients. The work of [13] aims at producing clusters which are under common administrative control. In our work, the approach is to produce clusters whose clients have similar distances to the content servers. These different goals can give rise to different clusterings. This is because some larger networks under common administrative control are spread over a large geographical area and do not fully expose the internals of their routing in BGP tables. For example, some BGP tables advertise a prefix consisting of dialup customers that are located throughout the US. With respect to implementation, one advantage of *Webmapper* is that it does not require access to any routing tables; however, *Webmapper* needs to collect distance measurements before it can produce an accurate clustering. A further difference between our work and [13] is that in addition to the clustering problem, we also examine the problem of assigning clients to content servers.

Another method for selecting the optimal content server is for the client to actively measure the distance to each content server. This approach was evaluated empirically in [6]. Although this method is effective at choosing a nearby content server, it cannot be used by a content provider for redirection of arbitrary clients since there is no way to remotely ask an *arbitrary* client to measure distances *right then*, and choose a content server based on its results. If, however, the client is under our control, we could run network monitoring from the client's side: see the SPAND work for extensive research on this and related ideas [15].

Research on a novel *anycasting* communication paradigm to support server replication and selection is reported in Bhattacharjee et al.[2]. Of the approaches to network monitoring suggested in [2], the one closest to our approach is to have probing agents (which act as proxies for clusters of clients) periodically querying the servers to measure network distance, server load, etc. As will be seen below, our work differs from this in that we do not use probing agents: our network measurements are passively carried out when clients access the various content servers. A key effect of this is that the system automatically evolves its clustering as the network changes. Since there are no probing agents, we do not need to modify their clustering as the network changes.

There has recently been a great deal of work on creating maps of the Internet. Jamin et al.[12] propose the deployment of *tracers* to create maps. Each tracer measures distances to other tracers and to selected regions of the Internet. Govindan and Tangmunarunkit [11] and Burch and Cheswick [3] have performed detailed experiments to discover the topological structure of the Internet.

The companies that run Content Delivery Networks such as Akamai [1], Digital Island [5] and Mirror Image [14] run proprietary algorithms for assigning clients to content servers. Hence we are unable to provide a comparison between their algorithms and *Webmapper*.

Several vendors of switching hardware offer products performing DNS-based redirection which make use of related techniques. The Global Server Load Balancing (GSLB) feature on the Foundry Networks ServerIron switches [8] appears most similar to our scheme, clustering client IP addresses into networks and gathering RTT data for each network. However, it differs significantly from our scheme in that its clusters all have the same prefix length (independent of the data observed), uses only very recent history (by default, data older than 2 minutes is expired), and gathers data from content servers other than the best one in a purely random fashion. The Cisco Distributed Director [4] also implicitly clusters hosts, basing its decisions on BGP/IGP information obtained by querying routers; this method has more in common with the work of [13] than with our approach. Finally, some products in this space do not perform any clustering. For example, the F5 Networks 3-DNS [7] actively probes individual local DNS servers to obtain distance information.

II. Overview of *Webmapper*

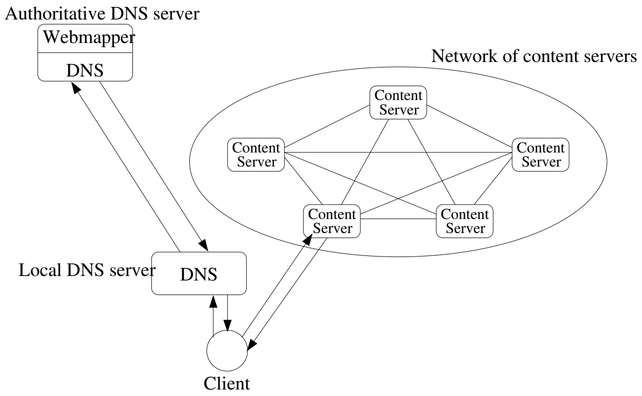

Fig. 1. The basic DNS resolution mechanism.

At a high level the *Webmapper* system works as follows. (See Figures 1 and 2.) The *Webmapper* sits behind a DNS front-end that acts as the authoritative DNS server for a domain or a set of domains. When the client wishes to resolve a domain name it sends a DNS request to its local DNS server which then forwards the request to the authoritative DNS server. The content servers collect client network distance and load information continuously and pass it back to *Webmapper*.

Periodically the *Webmapper* creates client clusters from current and previously aggregated (network distance, load) information where each client cluster represents a division or partition of the total IP address space. The clustering is important since it is inefficient to create a separate assignment for each IP address in the Internet. (There are many millions of such addresses.) Moreover, we would like to make a good assignment for addresses that generate little or no traffic. Fortunately, it is the case in today's Internet that many addresses with similar prefixes are near each another in the network; this is done to facilitate routing. We use the standard CIDR notation, $a.b.c.d/n$, to represent the cluster of IP addresses that match $a.b.c.d$ in the first n bits. We can look up an address in a table of such clusters using standard longest-prefix match.

The job of the clustering algorithm is to determine which prefixes correspond to groups of machines that are in the same place. It makes this determination using the measurements sent by the content server. Once the clustering has been determined, the assignment algorithm calculates a good content server or set of content servers for each cluster. This assignment (which we also refer to as the *map*) is then used by the DNS front-end to direct clients.

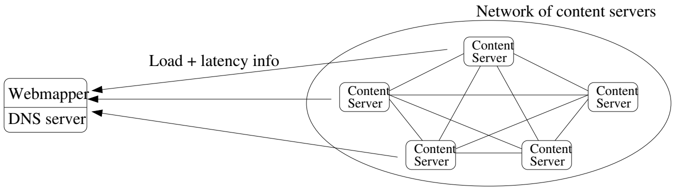

Fig. 2. *Webmapper* collects load and latency information from the content servers.

When we calculate the map we take into account server capacity. That is, we may not always wish to assign all clients to their closest server if doing so would overload one of the content servers. To circumvent this occurrence the selection probabilities are calculated so as to effect a load balancing.

One complication is that *Webmapper* may be responsible for a large number of domains (such as companyA.com, companyB.com etc.). If we create a separate map for each domain this can slow down the assignment algorithm significantly. Our solution, described in more detail in Section III, is to group domains according to where their content is placed. Each such group, which we refer to as a *domain index*, consists of a set of domains with their content in the same places (i.e., in identical subsets of the content servers). We then assign each client to a content server based on its IP address and the domain index of the domain that it wishes to access.

An example map is contained in the following table. The map can be read as follows. Consider a client with an IP address that is in the cluster 25.135.64.0/19. If the client wishes to access content in a domain that corresponds to domain index 2, then it is mapped to content server 2 with probability 1. In contrast, a client that is in the cluster 132.0.0.0/8 would be mapped to content server 2 with probability 0.5 and content server 3 with probability 0.5.

		servers		
client cluster	domain index	1	2	3
25.135.64.0/19	1	0.9	0.05	0.05
	2	0	1	0
	3	0	1	0
132.0.0.0/8	1	0.85	0	0.15
	2	0	0.5	0.5
	3	0	0	1

The remainder of the paper is organized as follows. In Section III we describe the monitoring processes that we run on each content server. In Section IV we present the clustering algorithm and in Section V we describe the assignment algorithm. In Section VI we describe a method for assigning a Time-to-Live value to each DNS response. We present our experimental results in Section VII and our conclusions in Section VIII.

III. The Content Server

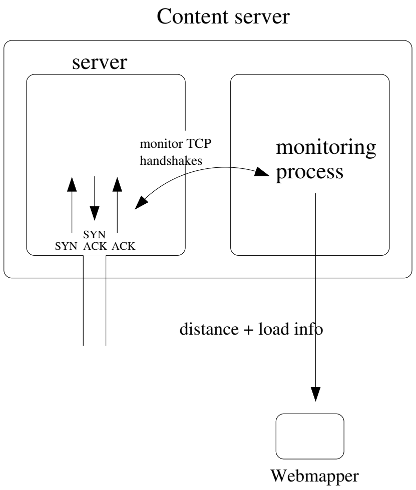

Fig. 3. Content Server architecture.

We collect our measurements using measuring software that is placed on each content server. (See Figure 3.) This software calculates a Round-Trip Time for each hit on the content server by measuring the time difference between the SYN-ACK and the client ACK in the TCP three-way handshake. Other measures of network distance could be used, such as a bandwidth estimate or a packet loss estimate. In the remainder of the paper we shall assume that the network distances being collected are Round-Trip Times.

The main advantage of this monitoring method is that it is passive, i.e. it does not produce much extra traffic on the network. A small amount of data is sent from the content servers to *Webmapper*. However, this traffic (around 20 bytes per hit) is insignificant compared to the actual content being served. This traffic can be further reduced by sampling during periods of heavy load. Crucially, this traffic is only between the content server and *Webmapper*; no extra traffic to or from machines outside the system is generated. A different approach for measuring network distance would be to use pings. However, the ping approach has two disadvantages. First, it creates extra traffic on the network. Second, many clients are configured to not accept pings and many firewalls are configured to filter pings. Therefore, it is difficult to measure the network distance to many clients using this ping approach.

Whenever a client opens a TCP connection to a content server, two tuples, a *distance tuple* and a *hit tuple* are collected by the content server for transmission to the *Webmapper*. The distance tuple has 4 components: Timestamp, content server ID, client IP address, and network distance, where the Timestamp is the time at which the network distance measurement was made, the content server ID identifies the particular content server making the measurement and the client IP address identifies the particular client accessing the content server. Network distance may be measured in the ways described above. The *hit tuple* has the following components: Timestamp, content server ID, client IP address, number of hits, and domain index, where Timestamp, content server ID, and client IP address are defined as for the distance tuple.[1] The domain index requires some explanation. It is an identification number assigned to each domain for which the *Webmapper* is responsible. Each domain index represents a set of domains that are treated as equivalent by the *Webmapper*. For example, each domain index may consist of a set of domains whose content is stored on the same set of content servers. As described later, the *Webmapper* creates a distinct mapping for each domain index. The domain index concept allows the saving of space and processing time if the *Webmapper* is serving many domains, each of which is treated the same. An example list of domains is given by the following three 3-tuples, where each tuple is of the form (domain index, constituent domains, content servers):

- (0,{images.companyA.com, companyY.com}, $\{0, 2, 4\}$);
- (1, {streaming.companyA.com, companyF.com, companyP.com}, $\{2, 3, 4\}$), and
- (2, {companyX.com}, $\{2, 4\}$).

IV. The Clustering Algorithm

The objective of the clustering algorithm is to partition all possible client IP addresses into a small number of groups, called *clusters*, of related hosts. For each cluster, the algorithm also provides an estimate of the distance

[1] In heavy-traffic situations the content server may not wish to send the distance tuple and the hit tuple for every hit. In this case the content server can summarize the data before sending it to the *Webmapper*.

from clients in that cluster to each of the content servers. This partition of client IP addresses into clusters thus compactly summarizes the client-to-server information. As new data arrives, both the partition and the distance estimates can be modified to reflect this information.

The clustering algorithm produces this grouping by analyzing data points which give distances from specific client IP addresses to specific content servers and grouping together clients with similar distance properties. This operation is usually performed at a prescribed fixed interval (e.g., every 30 seconds); we also have the possibility of running it at on demand–e.g., whenever diagnostic tools indicate that a major network event has occurred (such as the failure of a content server).

The clustering algorithm stores its information in a data structure we refer to as a *Big Tree*. (See Figure 4). This is a binary tree where the nodes of the tree represent clusters as described above. The root node is labeled "0.0.0.0/0" and denotes the set of all IP addresses, and has children labeled "0.0.0.0/1" and "128.0.0.0/1". More generally, each internal node $a.b.c.d/n$ has two children that represent the partition of $a.b.c.d/n$ with respect to the $(n+1)$st bit. For example the children of "117.104.144.0/20" are "117.104.144.0/21" and "117.104.152.0/21". In the simplest version, the tree has depth 32; its leaves, each of the form "117.104.13.10/32", correspond to individual IP addresses. However, since addresses which differ only in the last 8 bits normally belong to the same tightly-coupled network, we often restrict our trees to depth 24.

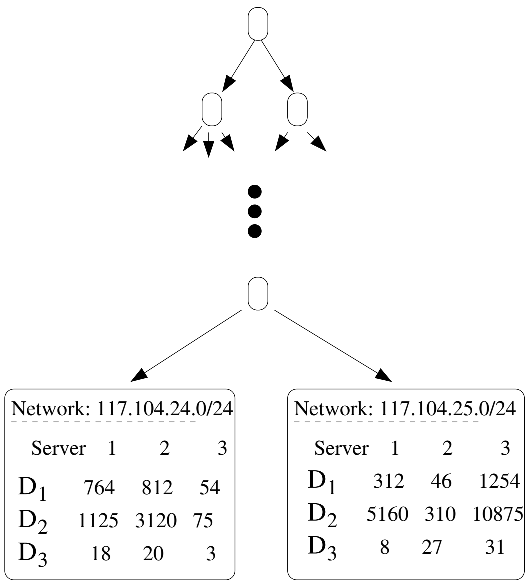

Fig. 4. The Big Tree.

Some initial prefiltering of the data, even before the data is stored in the Big Tree, may be helpful. In particular, we have found that using the logarithm of the distances, rather than the distances themselves, produces better clusterings because it is less sensitive to the heavy-tailed nature of the variations in network distance. This transformation has the effect of considering relative (such as "± 10%") rather than absolute (such as "± 10") differences in distance.

At each node, we store a summary of the data from IP addresses which belong to that cluster. More precisely, for each leaf cluster c and each content server s, we store 3 values, $D_1^{c,s}$, $D_2^{c,s}$ and $D_3^{c,s}$ which are updated as follows. Whenever we receive a distance tuple indicating a distance Δ between a client in c and server s we set,

$$\begin{aligned} D_1^{c,s} &\leftarrow D_1^{c,s} + \Delta, \\ D_2^{c,s} &\leftarrow D_2^{c,s} + \Delta^2, \\ D_3^{c,s} &\leftarrow D_3^{c,s} + 1. \end{aligned}$$

Hence $D_1^{c,s}$ represents the sum of the received distances, $D_2^{c,s}$ represents the sum of the squares of the received distances and $D_3^{c,s}$ represents the number of received distances. These quantities are chosen to allow us to efficiently test whether two clusters are similar. In addition, these stored quantities are further modified to ensure that recent data is given greater significance than old data. This is accomplished by exponential smoothing: At regular intervals, all data stored in the Big Tree is decreased by setting,

$$\begin{aligned} D_1^{c,s} &\leftarrow 0.9 D_1^{c,s}, \\ D_2^{c,s} &\leftarrow (0.9)^2 D_2^{c,s}, \\ D_3^{c,s} &\leftarrow 0.9 D_3^{c,s}. \end{aligned}$$

All new data is stored as before. Thus, the impact of past data decreases exponentially with time.

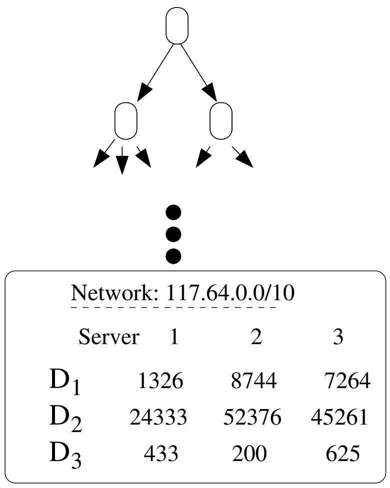

Fig. 5. The Small Tree.

The algorithm produces the partition by folding up the Big Tree into a *Small Tree*. (See Figure 5.) The leaves of the Small Tree will be the clusters to be output. The small tree is produced by repeating the following simple step: if there is a leaf node which either has no sibling, or whose sibling is sufficiently similar to it, remove it and its sibling from the tree. (We will discuss the details of testing similarity below.) This process folds up the children into their parent node, where their common data is aggregated as follows. If two children c_1 and c_2 are folded in their parent c_3 then we set,

$$\begin{aligned} D_1^{c_3,s} &= D_1^{c_1,s} + D_1^{c_2,s} \\ D_2^{c_3,s} &= D_2^{c_1,s} + D_2^{c_2,s}, \\ D_3^{c_3,s} &= D_3^{c_1,s} + D_3^{c_2,s}. \end{aligned}$$

When this process terminates, we are left with a partition of IP addresses into clusters (i.e., leaves of the Small Tree) where each leaf consists of a set of IP addresses with similar distance properties.

To complete this description, we need a test for taking two clusters (corresponding to sibling nodes of the tree) and testing similarity. Here, we use the two-sample t-test from statistics. This tests whether two sets of samples have the same mean by checking whether the difference between the means of the two sets is large compared to the variance in the data. If $x_1, \ldots, x_m$ and $y_1, \ldots, y_n$ are the data points in each set, the test computes

$$T = \frac{\bar{X} - \bar{Y}}{S_p\sqrt{1/m + 1/n}};$$

and compares it with a threshold. Here, $\bar{X} = (\sum_i x_i)/m$, $\bar{Y} = (\sum_i y_i)/n$, and S_p is the *pooled variance* given by

$$S_p = \frac{(m-1)S_X^2 + (n-1)S_Y^2}{m+n-2}.$$

where S_X and S_Y are the sample variances of the sequences $x_1, \ldots, x_m$ and $y_1, \ldots, y_n$ respectively. That is, S_X and S_Y are given by,

$$\begin{aligned} S_X &= \frac{1}{m-1}\left(\sum_i x_i^2 - \frac{(\sum_i x_i)^2}{m}\right) \\ S_Y &= \frac{1}{n-1}\left(\sum_i y_i^2 - \frac{(\sum_i y_i)^2}{n}\right) \end{aligned}$$

Given two nodes c_1 and c_2, for each content server s we use a calculation in the spirit of the t-test to decide whether the network distance to s is statistically indistinguishable for clients in c_1 versus clients in c_2. More precisely, we compute a value T_s by,

$$\begin{aligned} T_s &= \frac{D_1^{c_1,s}/m - D_1^{c_2,s}/n}{S_p\sqrt{1/m+1/n}} \\ m &= D_3^{c_1,s} \\ n &= D_3^{c_2,s} \\ S_p &= \frac{(m-1)S_{c_1}^2 + (n-1)S_{c_2}^2}{m+n-2} \\ S_{c_1} &= \frac{1}{m-1}\left(D_2^{c_1,s} - \frac{(D_1^{c_1,s})^2}{m}\right) \\ S_{c_2} &= \frac{1}{n-1}\left(D_2^{c_1,s} - \frac{(D_1^{c_2,s})^2}{n}\right) \end{aligned}$$

If T_s is below a given threshold *for all values of* s, then we declare the clusters represented by c_1 and c_2 to be indistinguishable and we merge them into their parent.

The algorithm then iterates over the clusters represented by the leaves of the Small Tree. For each cluster c and server s, the algorithm outputs $D_1^{c,s}/D_3^{c,s}$ which is an estimate of the network distance from c to s. We denote $D_1^{c,s}/D_3^{c,s}$ by $\delta^{c,s}$.

The Testing Index

Suppose that the assignment of clients is done solely on the basis of network distance. Recall that since our monitoring is passive, we only see distance tuples between a client and the content server to which it is assigned. Therefore it is desirable to sometimes change the assignment so that we collect new data. However, we do not want to do this in a way that impacts adversely the experience of the client. We solve this problem by basing the assignments on a *testing index* that is derived from network distance. We are more likely to assign a cluster to a content server for which the testing index is low. The testing index, $\tau^{c,s}$, for cluster c and server s is defined by,

$$\tau^{c,s} = \delta^{c,s}\left(1 - 1/\sqrt{D_3^{c,s}}\right). \tag{1}$$

(If $D_3^{c,s} \le 1$ (as a result of discounting), then we set the testing index to 0.)

In the case where the number of distance tuples (i.e., $D_3^{c,s}$) between the cluster and the content server is large, the square root term becomes insignificant, resulting in a testing index value that is essentially the network distance value. However, in the case where the number of distance tuples between the cluster and the content server is small, (say, less than 10) the square root term becomes significant, thereby reducing the testing index to a value less than the network distance value. Therefore, when we have less confidence in our estimate of network distance, we are more likely to assign the cluster to the corresponding content server so as to obtain more data.

Moreover, since our measure of the number of distance tuples, $D_3^{c,s}$, is discounted by a factor 0.9 over time, the testing index steadily decreases between a cluster and a content server to which it is *not* assigned. This increases our likelihood of changing the assignment and gaining new information. However, we are still unlikely to assign a cluster to a content server if our estimate of network distance is large. This notion of testing index is derived from the *Gittins index* [9] which arises in the theory of "multi-armed bandit problems" in stochastic processes.

Note that for each cluster, the testing index is initially 0 for all content servers. This encourages the algorithm to "try" each content server at least once.

V. CONTENT SERVER ASSIGNMENT

As described earlier, the creation of a map occurs in two stages. The first stage, namely the clustering algorithm, was described in the previous section. The output of the clustering operation is provided as input to the *content server assignment operation*. This operation pairs each (identified client cluster,domain index) pair with one or more content servers in the network. Each content server is assigned a selection probability. The reason that we do not assign every client to the closest content server is that we want to make sure that no content server is overloaded.

The selection probabilities are obtained by performing a min-cost network flow optimization routine which attempts to route network traffic in a more globally optimal manner to prevent or minimize network congestion at each content server in the network. Our objective is to construct a *Flow Map* so that clients are in general redirected to a content server which is nearby with respect to network distance. This should be done without promoting congestion at the content servers. As described below, this scheme uses the testing index computed by the clustering algorithm.

We describe the input to the assignment algorithm as a directed bipartite *digraph* $G = (X \cup Y \cup \{s,t\}, A)$. That is, the set of vertices of G is $X \cup Y \cup \{s,t\}$, and A is the set of arcs (i.e., directed edges) of G. The vertices are as follows:

- X is the set of ordered pairs (c,d) such that c is a cluster associated with a leaf of the Small Tree, and d is a domain index.
- Y is the set of content servers, and
- s, t are new *source* and *sink* nodes.

The arcs in A are of two types:

(A1) For each $x \in X,\ y \in Y$, we have the two arcs (s,x) and (y,t).

(A2) In addition, the input specifies certain arcs (x,y), where $x \in X$ and $y \in Y$. To see what types of arcs (x,y) will *not* be in A, recall that x is a pair of the form (c,d), where c is a cluster associated with a leaf of the Small Tree, and d is a domain index. If the content sites of domain index d do not store data at y, the arc (x,y) would *not* appear in G. Thus, if (x,y) is an arc of A where $x = (c,d) \in X$ and $y \in Y$, then this means that it is allowed for us to direct traffic that arrives at cluster c for content sites in d, to the content server y. See Figure 6.

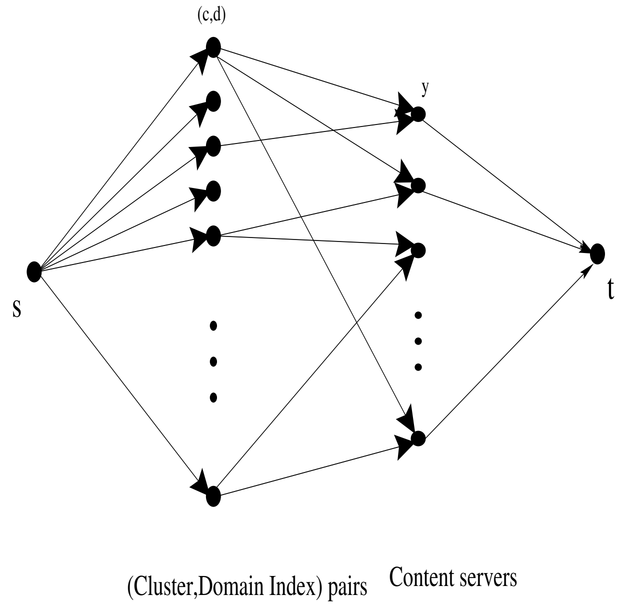

Fig. 6. The Digraph G.

Finally, the input also specifies the following numbers:

- A *demand value* r_x for each $x \in X$. If $x = (c,d)$, then r_x is the hit rate from clients in the cluster c for domains in the domain index, d. This value is computed using the hit tuple defined in Section III.
- The capacity C_y for each content server $y \in Y$. This is a measure of the total hit rate that the content server can support. It is specified either manually or experimentally.
- A *cost* for each arc $a \in A$. If a is an arc of type (A1), its cost is 0. Otherwise, suppose a is an arc of type (A2); thus, $a = (x,y)$, where $x = (c,d) \in X$ and $y \in Y$. Then the cost of a is just the testing index $\tau^{c,y}$ for cluster c and content server y.
- A threshold value g, representing a minimum acceptable testing index.

The assignment algorithm starts by computing a *capacity* u_a for each arc $a \in A$; it then finds a minimum-cost flow from s to t of value $\sum_{x \in X} r_x$. The output is a flow from s to t which determines the *selection probabilities*. Namely, for $x = (c,d)$ and content server y, if the flow produced from x to y is $f(x,y)$, then we resolve hits for domain d arising from cluster c to the content server y with a probability of $f(x,y)/r_x$.

The arc capacities u_a are computed as follows.

(a). If the arc a is of the form (s, x) where $x \in X$, then we set $u_a = r_x$.

(b). For each arc $a = (x,y)$ where $x \in X$ and $y \in Y$, $u_a = 0$ if the cost of a is greater than the threshold g; otherwise, $u_a = r_x$. (Suppose $x = (c, d)$. Then, recall that the cost is the testing index between c and y. Thus, if the testing index is more than g, we set $u_a = 0$; this will have the effect that no flow from c will be redirected to content server y, since the testing index between these two is more than our desired threshold g.)

(c). Finally, for each arc $a = (y, t)$, where $y \in Y$, the most basic strategy is to set $u_a = C_y$, the capacity of content server y. This has the effect of ensuring that the expected number of accesses to each content server y, is at most its capacity C_y. However, since content server I/O operations start becoming a bottleneck if a content server is too close to being overloaded, an alternative here is to define u_a to be, say, $0.8 \cdot C_y$.

Once the network's capacities are set as above, a fast implementation of an algorithm due to A. V. Goldberg [10] is used to solve the min-cost flow problem.

Note that the mincost flow problem could be infeasible for one or more of the following reasons:

- the threshold g is too low, thus leading to a digraph D with too few arcs;
- the capacities of the arcs (y, t) are too small, or
- for some $x = (c, d) \in X$, there is no arc in A of the form (x, y).

One possibility is to simply raise g and solve until the flow problem does have a solution. If this is not desirable, we could solve for a flow that minimizes the maximum or total violation of the capacity on the arcs (y, t). This second approach is also easily amenable to situations where, for instance, we do not necessarily aim to make content server loads proportional to C_y, but, say, want to maximize the minimum space left free on any content server. Another variant is to raise the capacity of each content server by some constant factor until a feasible flow exists.

VI. Time-to-live Calculation

In this section we describe how we calculate the Time-to-Live (TTL) that is returned with each DNS response. Our goal is to output larger Time-to-Live values in the case where we are confident that the preferred content server is not going to change for a long time. If we do not have this confidence, then we use a smaller value. More precisely, we define the Time-to-Live (in seconds) for each DNS response to a client in cluster c by

$$\mathrm{TTL} = 10^6 \cdot (\text{DNS server load})/((m+s)^2\sqrt{N}),$$

where m is the sample mean of network distance from c to the chosen content server, s^2 is the sample variance of network distance from c to the chosen content server, and N is the number of content servers. The intuition here is as follows. We would like the TTL to be:

(a) an increasing function of the load on the DNS server, so that a heavily loaded DNS server is not frequently bothered with requests. (The DNS server load is defined to be the rate at which it receives DNS requests.)

(b) a decreasing function of the variance s^2, since large variance connotes unreliable data;

(c) a decreasing function of m, as we do not want to be stuck with a bad choice of server (which corresponds to m being large), and

(d) a decreasing function of N, since if there are many content servers, we would like to check hitherto unexplored servers to see if they are a better choice.

The specific constant and exponents in our TTL formula above result from our experience with actual data.

VII. Experimental Results

In this section we describe two experiments that we performed in order to evaluate our techniques. In the first experiment we recorded all the hits on the webserver *www.bell-labs.com* for 28 days. For each hit we recorded the client IP address, the time that it occurred and the Round-Trip Time to the client. During that period, we received 1,775,020 hits from 116,773 distinct clients. Since we only have one content server we cannot use this data to evaluate the client assignment part of *Webmapper*. However, we can use this data to evaluate the clustering algorithms.

As discussed earlier, we stored the data in a Big Tree that was truncated at the 24-bit level. We then ran the clustering algorithm, producing a set of 17270 clusters. In Figure 7 we illustrate 12 of these clusters using data collected on a single day that is *not* one of the original 28 days. These clusters were chosen because they have the most number of hits that day from among those clusters with hits from at least 10 clients. The latter condition ensures that we are not examining trivial clusters with only a few clients.

For each hit, we plot $\log_2$ of the Round-Trip Time (in ms) against time of day. It can be seen that the Round-Trip Times in each cluster exhibit a high degree of correlation. In addition, for each cluster we select the client IP with the most hits and plot its hits with an "$\times$" instead of a dot. From this we see that the variability in Round-Trip Times for entire clusters is similar to that for individual clients within that cluster. This suggests that the variations in Round-Trip Time visible within each cluster are

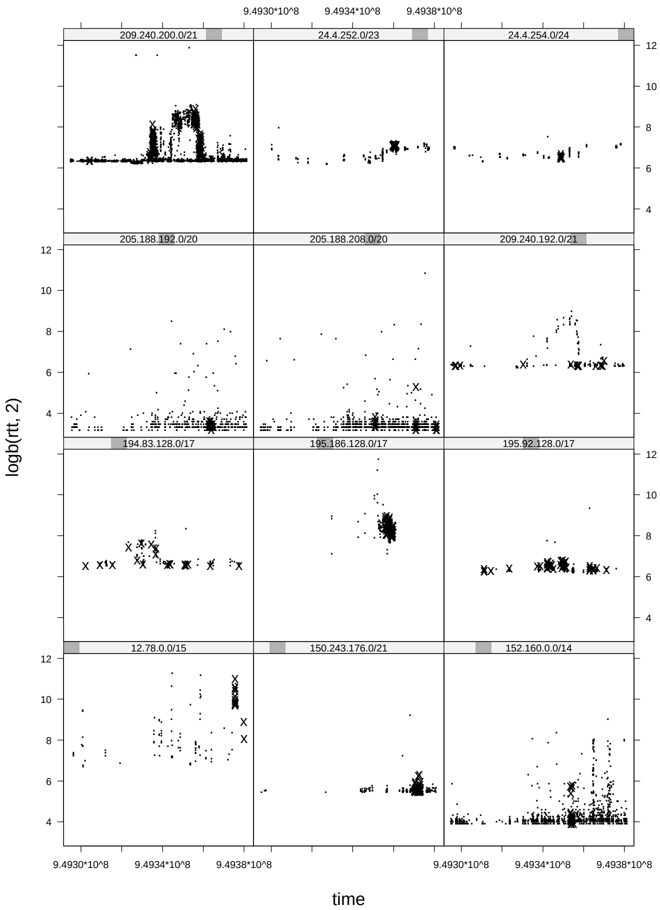

Fig. 7. Example clusters produced by *Webmapper*.

inherent, due to the variability of individual client Round-Trip Times, rather than the result of over-aggregation in our clustering algorithm.

Unfortunately, the data used in the first experiment cannot be used to test the assignment algorithms since it was all directed at one web server. Hence we performed a second experiment to test the assignment algorithms. We set up two web servers, one on the East Coast and one on the West Coast, controlled by *Webmapper*. On several web pages we had control of, we embedded a 1x1 pixel transparent GIF image which was served from the *Webmapper* system. We also placed two other images on the web pages, one of which always generated a hit on the East Coast machine. The other always generated a hit on the West Coast machine. This gave us the distance from any client to each of the servers, thereby enabling us to validate our assignment. In this experiment, we focus on the goal of directing clients to the closer (in RTT) of the two servers, and ignore other issues such as server load. Over the course of the experiment we obtained 111649 hits from 11321 clients.

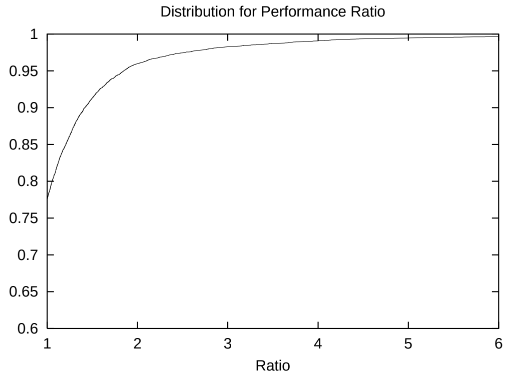

Fig. 8. Distribution of Performance Ratio.

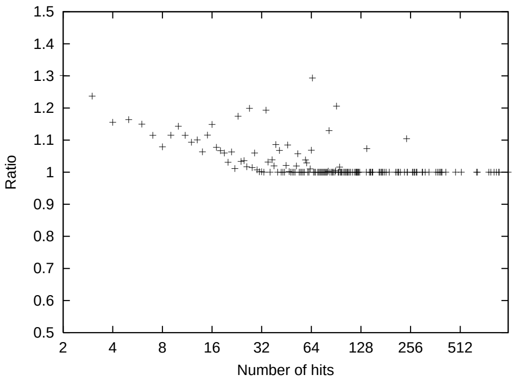

Fig. 9. Performance ratio vs. number of hits.

Recall that one of the reasons we use clustering for assignment is that it enables us to choose a content server for a client even if *Webmapper* has not see the client before. We only need to have seen other clients in the same cluster. However, in order to validate our results we need to know which server really is the best for a given client. Hence we focus on the clients that generated more than 10 hits. For each such client we calculate the mean Round-Trip Time to the web server assigned to it at the end of our experiment divided by the mean Round-Trip Time to the best web server. (By "best" we mean the server with the smallest mean Round-Trip Time.) In Figure 8 we plot the percentage of clients for which this performance ratio is at most r for all values of r. We can see that 75% of the clients are directed to the best content server. An additional 20% of the clients are directed to the wrong content server but the network distance to that server is at most twice the distance to the best server.

In Figure 9 we plot the mean value of the performance ratio for all clients from which we received a given number of hits (on a logscale). It can be seen that the ratio is small even if we have seen a small number of hits. However, not surprisingly the ratio is even better for clients from which we have seen a large number of hits, except for a small number of outliers. (These outliers are for hit counts for which we only have a small number of clients.)

There are three main reasons why we sometimes direct to the "wrong" content server. First, when the *Webmapper* sees a DNS request, it only knows the address of the local DNS server and not the address of the client itself. Hence, it is important that the local DNS server is close to the client. Moreover, we need to receive distance information about the cluster that contains the local DNS server. These conditions were not always satisfied. This problem is inherent to all DNS based direction schemes. The second reason is that for some clusters, all of the hits from that cluster arrive in a short burst (for example, an individual user session). These hits are directed incorrectly (since we did not initially have good data for that cluster), and by the time a new assignment is computed, it is too late to affect the results. This problem was exacerbated by the small number of hits obtained in our second experiment. The third reason is that some clients (especially clients on dialup links) have extremely variable Round-Trip Times to both content servers. It is inherently difficult to determine the most appropriate content server for such a client.

VIII. Conclusions

In this paper we have presented our *Webmapper* system for clustering IP addresses and assigning clients to content servers. The system operates by passively monitoring the TCP connections when clients connect to the content servers. This data is used by *Webmapper* to create an assignment map for clients. We transmit the assignment information to clients using the DNS system.

We summarize the main advantages of our approach. First, in order to assign clients to the best content server, performance must be measured in some way. Even if BGP tables are used for clustering, we still need measurements to determine the best server for each cluster.

Next, client-side measurement appears infeasible (since a scalable system should be transparent to the user) and large-scale active measurement has both administrative and technical hurdles (e.g., complaints, firewalls). Passive monitoring can be run by a simple agent process, avoids these other problems and generates insignificant traffic compared to the content being served. Moreover, monitoring the RTT values allows us to react to network congestion.

We expect to conduct a more detailed empirical validation of our system. At this stage our experiments do show "good enough" performance, and show, for instance, that when one server is noticeably better than the others, the system is wrong for less than 5% of the time.

As mentioned in Section I, other methods for assigning clients to content servers have been proposed in the literature. It would be interesting to see if a combination of these methods can produce better assignments.

References

[1] Akamai Technologies, URL: http://www.akamai.com.

[2] S. Bhattacharjee, M. Ammar, E. Zegura, V. Shah and Z. Fei, "Application-Layer Anycasting", in *Proceedings of INFOCOM '97*, 1997, pp. 1388–1396.

[3] H. Burch and B. Cheswick, "Mapping the internet," *IEEE Computer*, vol. 32, no. 4, pp. 97–98, 1999.

[4] Cisco Distributed Director, ," URL: http://www.cisco.com/univercd/cc/td/doc/product/iaabu/distrdir/index.htm.

[5] Digital Island, URL: http://www.digitalisland.com.

[6] S. Dykes, K. Robbins, and C. Jeffery, "An empirical evaluation of client-side server selection algorithms," in *Proceedings of INFOCOM '00*, March 2000, pp. 1361–1370.

[7] F5 Networks 3-DNS, ," URL: http://www.f5.com/3dns.

[8] Foundry Networks Global Server Load Balancing, ," URL: http://www.foundrynet.com/products/GSLB.html.

[9] J. C. Gittins, *Multi-Armed Bandit Allocation Indices*, Wiley, 1989.

[10] A. V. Goldberg, "An efficient implementation of a scaling minimum-cost flow algorithm," *Journal of Algorithms*, vol. 22, pp. 1–29, 1997.

[11] R. Govindan and H. Tangmunarunkit, "Heuristics for internet map discovery," in *Proceedings of INFOCOM '00*, March 2000, pp. 1371–1380.

[12] S. Jamin, C. Jin, Y. Jin, D. Raz, Y. Shavitt, and L. Zhang, "On the placement of intenet instrumentation," in *Proceedings of INFOCOM '00*, March 2000, pp. 295–304.

[13] B. Krishnamurthy and J. Wang, "On network-aware clustering of web clients," in *Proceedings of SIGCOMM '00*, August 2000.

[14] Mirror Image, URL: http://www.mirror-image.com.

[15] SPAND: Shared Passive Network Performance Discovery. URL: *http://www.cs.berkeley.edu/ stemm/spand*