MASH: The Multicasting Archie Server Hierarchy

Adam Rosenstein
adam@cs.ucla.edu

Jun Li
lijun@cs.ucla.edu

Si Yuan Tong
tong@cs.ucla.edu

Abstract

The `archie`[1] system is a replicated, centralized directory server for all Anonymous FTP sites in the Internet. This centralized approach has not scaled well. The `march`[2] system provides an alternate solution, using IP Multicast to distribute directory queries directly to FTP hosts. `march` suffers from highly redundant broadcast messages during expanding-disc search. We propose a solution to the problem of multicast flooding during expanding-disc searches by utilizing an automatically-configured hierarchy of query servers. This system, dubbed `MASH`, confines the multicast flooding primarily to those nodes at the frontier of each new search radius. We provide a mechanism and protocol for building a self-organizing, two-level hierarchy of `MASH` servers. We also provide an experimental implementation built on `march`[5].

1 Introduction

Anonymous FTP is a service allowing any user connected to the Internet to retrieve files stored on publicly accessible file servers. FTP is command-line and connection oriented, thus requiring users to log into an individual FTP server in order to determine the file contents of that server. While this interface is adequate for users who know which servers contain the sought-after information, it fails to address the problem of locating a particular file amongst the more than 2,500 anonymous FTP servers currently listed in the Anonymous FTP FAQ[3].

This paper presents a new approach allowing a user to search the directory information of all the anonymous FTP servers without requiring a server to centrally collect the information before-hand. We present an architecture called MASH (Multicast Archie Server Hierarchy) which runs on servers at each of the anonymous FTP sites. The MASH design incorporates a dynamic self-organizing group formation mechanism which forms a hierarchy, grouping MASH servers topographically. The rest of the paper is organized as follows: in Section 2 we describe the current solutions and state some drawbacks associated with them. In Section 3 we describe the MASH approach and compare it to the related work. Section 4 describes the dynamic self-organizing group formation algorithm. In Section 5, we discuss our implementation of MASH. Finally, we conclude with a short, reflective summary.

2 Related Work

2.1 `archie`

In 1992, `archie`[1] was created to provide an easy way to find where files are stored on the Internet. `archie` is a service which collects directory information from a large set of FTP servers and makes this information available to `archie` clients, via a query/response mechanism. Because each of the 2,500+ FTP sites produces complete directory listings ranging from 2 kilobytes to 25 megabytes in size[3], `archie` servers are heavyweight database servers. The problems with `archie`'s approach are:

- The directory information is disbursed but `archie` is centralized. Thus, it is costly for a central `archie` server to frequently probe so many FTP sites, or to do so frequently enough to ensure consistently up-to-date information. Consequentially, the information in `archie` servers is frequently out-of-date, and incomplete.
- Because information must be centrally collected, there can only be a few `archie` servers for a feasible implementation (public `archie` servers currently number 38, worldwide[3]). Therefore, the query/response system has a bottleneck at the server.

2.2 Multicasting

IP Multicast, first proposed by Deering, et al. [4] is a mechanism by which packets may be efficiently routed from one source to many destinations. Other than the efficiency with which queries may be distributed to many destinations simultaneously, there are two other advantages afforded by multicast for this application:

1. A client may query the set of servers without knowing their explicit locations (this capability

is available irrespective of the IP Multicast routing protocol);

2. A client may use TTL-based scope control in order to contact the topographically closest servers first.

2.2.1 A Multicasting Archie Service — `march`

Given the capabilities of multicasting, it is possible to design a system that efficiently transports directory queries to a set of distributed directory servers. These servers would ideally be located at the actual sources of the directory information (e.g. one directory server at each anonymous FTP site). There is, at least, one such implementation, called `march`[2]. `march` is a multicasting distributed directory database system which relies on a single multicast address for all directory servers. It uses TTL scope-limiting to constrain the impact of individual queries. By iteratively expanding the TTL, a `march` client finds the closest (topographically speaking) FTP site containing the requested information. This type of search is often referred to as an "expanding-ring" search. Expanding-ring searches are inherently robust, as any servers that fail to receive a query during one iteration have another opportunity to receive the query during the next iteration.

However, there are drawbacks to the `march` approach. In each iteration of expanding-ring search, queries must be routed to all `march` servers that were reached on previous iterations. Such flooding (even inside a limited TTL radius) of a pervasive multicast group can result in unnecessary traffic. Owing to this re-querying of the inner (previously queried) search rings, such searches may be more aptly referred to as "expanding-disc" searches. Although the inner disc does not service repeated queries in the `march` architecture, the routers do not understand this and must still route all repeated queries to the same servers at potentially great cost with no additional benefit. Should `march` become a popular service, this poor scaling factor could contribute to Internet congestion.

3 A Hierarchal Approach

In order to address the problems posed by expanding-disc searches, we propose a hierarchical approach. Our solution is to construct a two-level hierarchy of `march` servers. This service is characterized by one well-known, pervasive multicast group (G_{global}), and a number of topographically localized subgroups. The well known group is much like `march`'s group, but its number of members is comparatively small, and is self-adjusting as the number of servers in operation scales up. The subgroups each have their own multicast address. The servers dynamically organize themselves into these groups. Each group includes one (only) member of the pervasive group. This "parent member" receives queries from clients on the global multicast address, G_{global}, and dispatches these queries to its subordinate servers via its unique local group multicast address.

3.1 An "Expanding-Wheel" Search

In expanding-disc searches, the client completely controls the impact of its searches. Since only a TTL limit is used, maximum radius is the only dimension which may limit the scope of a search. In a hierarchy, each hierarchical layer can share the burden of restricting search scope. What results is a search pattern whose impact, with respect to the multicast traffic it generates, is greatest at its frontier, and restricted to minimal "spokes" en-route to the frontier. Thus we call our search an "expanding-wheel" search.

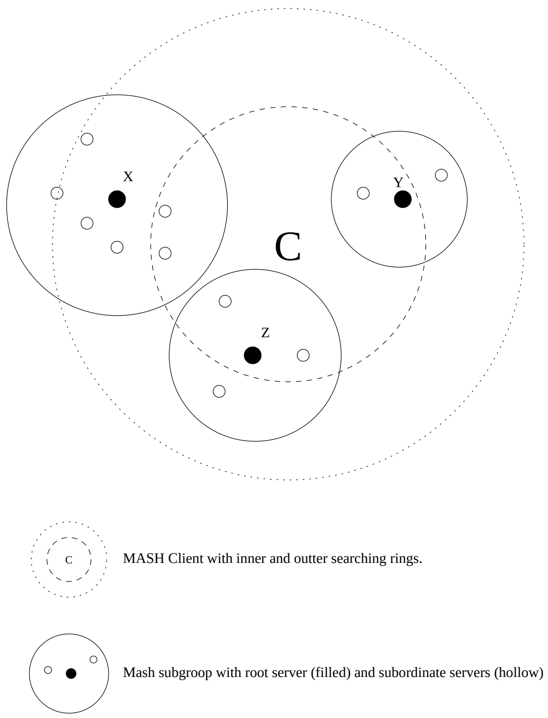

Figure 1: Expanding-Wheel Topology

Figure 1 shows an example of this approach. First, the client **C** sets its TTL limit to some small initial value (the dashed inner circle) and transmits its query to the global multicast address. Only the root level servers (dark filled-in circles) listen to this address.

In Figure 1, **C**'s first transmission will reach servers **Y** and **Z**. These servers will either respond themselves (if their local databases match the query) or they will retransmit **C**'s query by multicasting it to their respective groups' multicast addresses. If any server hearing this query can respond, it does so directly to **C**. If **C** hears no responses for some time, it will increase its TTL and retransmit its query again on the global multicast address. In our example, the rebroadcast query will reach root servers **X**, **Y**, and **Z** (the dotted, outer circle). **Y** and **Z** will ignore the retransmitted query (by checking a record of recent queries) but **X** will respond in the same manner **Y** and **Z** did the first time. The advantage of this method over expanding-disc search is that, on TTL-expansion, the multicast infrastructure will have to carry the un-needed retransmission only to **Y** and **Z**. In expanding-disc, however, the retransmission would go to all of **Y**'s and **Z**'s members (all previously reached destinations). Under significant scaling conditions (scaling up the size of the search, not the number or size of the groups) expanding-wheel can lessen the multicast traffic load tremendously. This reduction is due to the fact that the majority of servers are children who do not even subscribe to G_{global}, and thus are never involved in the multicast distribution tree for the repeated queries. See Section 3.2 for more details.

3.2 Comparison of Expanding-Wheel and Expanding-Disc Approaches

For a given set of servers, the expanding-wheel architecture will generate longer path lengths for a query to reach each server in that set than the expanding-disc architecture. In addition, the expanding-wheel architecture generates overhead associated with maintenance of the hierarchy and uses more multicast groups than expanding-disc. Expanding-wheel's main improvement over expanding-disc is its reduction of multicast traffic when a failed search elicits an expanded, reiterated query. Therefore, for expanding-wheel to be an improvement over expanding-disc, the amount of traffic generated on the second, third, and further "wheel diameters" must be reduced to offset the cost of increased path lengths, protocol overhead, and its greater number of multicast groups.

For a two-level hierarchy, the amount of this reduction is proportional to the number of members in each group. For example, in a hierarchy with 10 servers per group, an expanding-wheel query will reach only $\frac{1}{10}$ of the servers reached in the previous iteration, plus those servers being reached for the first time in the current iteration. This is a significant savings over the expanding-disc search for those cases where the search fails to elicit any responses for several iterations. In addition, the savings afforded by expanding-wheel allows more liberal use of repeated search iterations, such as in searches that attempt to gather a large set of responses, as opposed to halting with the first reply.

4 Self-organizing Groups

In order to implement an expanding-wheel search, a hierarchy of servers must be established. The higher-level servers are "parents" of the lower-level servers, the "children". A two-level hierarchy suffices to demonstrate the feasibility of expanding-wheel search groups.

4.1 Group Formation

In order to form groups automatically, servers must communicate with one another. The most obvious channel of communication is the multicast groups that will already exist for the forwarding of queries from clients to high-level servers, and from parents to children. In the following discussion, it is assumed that the highest level multicast address, G_{global}, is information known to all servers upon startup. In addition, the following nomenclature for servers will be used:

Parent A server that has elected to become a parent. It has no servers above it in the hierarchy, but may have a group of child nodes associated with it.

Child A server that has accepted a parent and will therefore not accept children (ours is a two-level hierarchy). A child will have only one parent.

Newborn A server that has recently been started (or restarted). It has neither accepted a parent, nor any children.

4.1.1 Newborns

Newborns begin their lives by sending out a multicast request to join nearby groups. This *adoption* request is sent to all parents via G_{global}. The newborn uses a traditional expanding-disc search methodology in order to find the parents closest to it that are willing to accept it. After each adoption request is multicast, the newborn waits for a short period of time to give parents time to respond.

During this waiting period the newborn records any *invitation* messages which existing parents send to it. When the period is over, the newborn checks its invitations. If it received any, it selects the best parent and joins its group. This is accomplished by sending a *member acknowledgment* message. This message is also used during group maintenance and is discussed in Section 4.2.2.

Many metrics could be used to judge potential parents. Intuitively, we use parent loading (ratio of a parent's active children to its potential number of children) as a measure of parent worth. Parent distance is also a vital measure, but, owing to the nature of expanding-disc searching, the closest parents will not need to compete with those farther away.

The mechanism parents use to determine whether to issue an invitation message in response to a newborn's adopt message is a more complicated matter, and is deferred until Section 5.1. However, it is important to mention here that the parent needs one vital piece of data in order to judge newborn worth: the newborn's distance to itself. Since the newborn uses expanding-disc searches, it knows that the maximum possible distance to a parent responding in each search iteration is equal to the TTL used in that iteration. Therefore, the newborn's current adoption search radius is placed as data in the adoption packets so parents may know the maximum distance to the child requesting adoption.

4.1.2 Child/Parent Decision

For each search disc that returned no invitations, the newborn may expand its TTL limit and reissue it's query. This ensures robustness in the adoption protocol. If any messages are missed (going in either direction) the next query cycle will rectify matters. In light of this, parents must react to each adoption message they see, even if it is a duplicate message. Indeed, if a duplicate message is detected, the parent must eliminate any state associated with responding to the previous message, and repeat the steps it took the first time (issue invite message, record child information, etc.).

Parents that accept a newborn will send an invite message. This message must contain enough information for the newborn to communicate with the parent, and for the newborn to make informed decisions about which parent to choose. The former requirement can be filled by supplying the parent's local multicast address in the invite message. The latter, as discussed previously, could be accommodated by including a measure of parent loading in the invite message. Upon electing to join a parent's group, the newborn ceases its search, and begins responding to queries and control traffic heard on the parent's local multicast address (G_i, for the i'th group) only.

If a newborn continues to expand its search disc without receiving invitations it must stop at some point. One simple choice of stopping criterion is a predetermined radius limit, R_{max}. Once the search radius has progressed past this limit without yielding any invite messages, the newborn must elect to become a parent. This is accomplished by ceasing the adoption search, acquiring an unused multicast address to use as the local group address (G_i), and beginning to respond to newborn adoption requests and client queries heard on G_{global}.

4.2 Group Maintenance

Once established, parent-child relationships are kept current with a parent-initiated query cycle. This *member-query* cycle occurs at infrequent, predictable intervals.

4.2.1 Member queries

Parents issue a query to all of their children at regular intervals. Parents already possess a simple means of communication with their children, their local group multicast address, G_i. According to Section 4.1.1, parents use newborn distance to elect those adoption requests to respond to. This implies that parents must keep track of the distance of their active children as well (for comparison purposes). Also, because the member query cycle serves to maintain the group integrity, it should be robust. All of these observations indicate that expanding-disc searching is suitable for member queries.

When a child responds to a member query, the upper bound on the distance between the parent and child is equal to the radius used to send the member query. If the parent includes this distance in the member query message, each child will also know this bound, and may incorporate that information into its response.

Expanding-disc searching also exhibits the desired robustness. The member-query cycle consists of parents multicasting a *member-query* packet to all group members on G_i. The member-query packets must contain the hop count used so that children may indicate this value when responding. A query identifier is also associated with each batch of queries (one query ID for all iterations of the expanding-disc query). A suitable stopping condition for the expanding-disc query is when all children have responded or when the farthest previously known child distance is reached. It should be noted here that the failure of a child to respond to a query may be detected by the parent, but we have not said what action is taken in this case. As will be explained in Section 4.2.2, the correct action is none-at-all.

4.2.2 Member acknowledgments

Upon receiving a member query, a child server will unicast a corresponding *member acknowledgment* to its parent. This acknowledgment will contain the hop count of the query and the query ID of the query, taken from the query itself. Children will not respond more than once to a given query cycle, indicated by the query ID.

Because of the robustness of expanding-disc searching, children may use the absence of parent queries to indicate parent death, or loss of communication with the parent. No farther action need be taken on the part of the child to determine parent status. In practice, a child need only wait the member query interval plus several times the estimation of the one-way trip time from the parent to the child (allowing for missed queries to be recouped on subsequent disc-expansions). If a child hears no query in that time, it *orphan*s itself; i.e. it restarts itself as a newborn. It will then rapidly find the closest live parent, or become a parent itself. To avoid all of a dead parent's children from synchronizing (possibly missing each other as potential parents), a random delay can be introduced between the decision to orphan oneself, and actually carrying out the act.

If a parent does not receive a member acknowledgment from a child, it need not delete that child's state. As explained in Section 5.1, the parents use a greedy algorithm, and never relinquish a child (even a dead one) unless a topographically closer replacement is at hand. Thus, a child may send its acknowledgment late and be reinstated, provided the parent has not had a pressing reason to delete it. However, if a parent does delete the state associated with a child, and then later receives an acknowledgment from that child, the parent must explicitly orphan the child by means of a unicast *orphan* message. A child that receives an orphan message from its current parent orphans itself, as described above.

4.3 Observations

The self-organizing mechanism is controlled by the global parameter C_{max}. This parameter controls the maximum number of children a parent will accept, and it is discussed in more detail in Section 5.1.1. Adjustments to this parameter greatly effects the performance of the query/response system. When all servers use a well-known, static C_{max} the automatic group formation protocol can become saturated, causing new servers to become *lonely* (having neither a parent nor children). This section addresses these problems and suggest some potential solutions.

4.3.1 Some Thoughts on Group Size

For a querying client outside the topographic radius of a group, the average path length for a query to reach all members of that group is roughly equal to the path length in expanding-disc plus the radius of the group. Therefore, the smaller the group radius, the smaller the path length necessary to reach all nodes. Also, because the purpose of expanding-disc searches is to find information nearest the querier, smaller group radii offer a finer "resolution" of searching to the clients.

However, as group size is reduced the performance of the system approaches that of expanding-disc. In particular, if the group radius is reduced to 0 (all groups having one high-level member listening to G_{global} and no children) the system behaves exactly as expanding-disc, with some additional control overhead.

For larger group sizes, the savings in traffic on repeated iterations of the expanding-wheel search is greater. This can be seen by noting that expanding-wheel searches always need to go through the high-level servers (even if those servers ignore the repeated queries). Larger group radii will produce a smaller ratio of high-level servers to low-level servers for a fixed number of overall servers. Thus fewer nodes will initially receive queries sent on the high-level multicast group address.

4.3.2 Load Balancing

In a large, distributed, self-organized system, the problem of maintaining an optimal state can be difficult. One such optimization concern in MASH is the task of evenly distributing the child nodes amongst the available parents. Because no node can easily acquire a global picture, parents must rely on local information to determine if they are in or out-of-balance with their neighbors. The simplest criterion for a parent to use is its own number of children.

4.3.2.1 Lonely Parents

If a newborn appears inside of a densely populated area, it is possible that it may not be adopted by any parent because all parents have C_{max} children. If this new child then becomes a parent, it may always have very few children. If a child is born nearby, this parent will adopt it even if the other nearby parents are closer (due to their "full" status). This will allow overlapping groups, which would ultimately cause more multicasting traffic and less resolution for clients wishing to control the radius of their expanding-wheel.

Another possible cause of this effect is large-scale link failure. If many links fail, the children attached to these links will not be able to reach their parents and will orphan themselves. Upon restarting they will hear no responses to their adoption requests (links are still down) and will become parents. Now, when the links are once again fully functional, there will be a population consisting of mostly parents.

These two very different causes each produce the same locally observable result: "lonely parents." Lonely parents are parents whose number of children is small. Determination of what value constitutes "small" need not be a complicated computation. What is important is that the *population* of parents (as a whole) react in a manner that:

1. Distributes the available children fairly amongst the parents
2. Recovers from an environment that is saturated with parents

One possible solution to this problem is for such "lonely parents" to send a *lonely parent* message to other nearby parents indicating their lack of children. This could be done periodically (on the order of several member query cycles). For example, whenever a parent has fewer than $\frac{1}{2}C_{max}$ children. The radius of such a packet should be kept just large enough to reach only those parents that are very nearby (certainly bounded by R_{max}).

4.3.2.2 Responding to Lonely Parents

Non-lonely parents who hear a lonely parent packet could respond in several ways. In order to ensure fair distribution of children, parents could orphan some (or possibly all) of their children, allowing fair competition for them via the "nearest parent first" nature of the adoption packets. This practice would allow localized, on-demand, load balancing to occur.

Alternatively, neighbors of lonely parents could respond by inviting lonely parents to join their groups as children. This would address parent-saturated environments by demoting unsuccessful parents back to child status. In a parent-saturated environment this is the right thing to do, even if the lonely parent has some children that would be orphaned by this procedure.

The correct response upon receipt of a lonely parent packet depends on the environment in which the event occurs. In a parent-saturated environment, offer an invite packet to the lonely parent. In a overburdened parent environment, orphan some children. Rather than attempt to determine the status of the environment, each parent should base its response on its locally observable situation; i.e. whether or not it is lonely too. A summary of responses to receiving a lonely parent packet is given in Table 1.

In either environment, parent-saturated or newborn-poor, the lonely parent P′ will be able to become "unlonely" and balance the child load. In a mixed environment (some parents with near C_{max} children while many others remain nearly childless) P′ will be in a position to decide to restart or gather the newborns other parents have released for it based on a comparison between how attractive (distant) the newly available children are versus how attractive (distant) the willing parents are. Table 1 shows that when node P (which hears the lonely parent message from P′) is also lonely, it pursues some action to re-balance its load, as well as aid P′.

4.3.2.3 Other Load Balancing Procedures

In order to prevent a poorly situated parent from alternating between lonely parent and newborn, lonely parents should only send lonely parent messages infrequently. In the interim, parents with C_{max} (or near C_{max}) children can occasionally orphan their worst (farthest) child as preventive maintenance. More appropriately, all parents can exercise a random orphaning regime in the interest of global load balancing. As a parent's number of children increases so does the likelihood and/or frequency with which it orphans its children. This likelihood could be controlled via a probability function such as:

$$\frac{f}{(C-(C_{max}+1))^2} \tag{1}$$

where f is the frequency (in member query cycles) with which a child is orphaned and C is the current number of children. Refer to Figure 2.

4.4 Issues of Node Performance

Hop count is certainly not the only measure of a node's value in the server population. Other concerns are the bandwidth and latency of its network interfaces, the size of its database and the speed of its processor (amongst others). The protocol described in this paper will prefer a slow child node to a fast one, if the former is topographically closer. This practice will not necessarily damage the overall performance. If a child is connected to a parent by a slow link it need only reply to member queries within the allotted

Is P lonely	Situation	Reaction
Yes	P has fewer children than P′	P orphans itself
Yes	P has more children than P′	P invites P′
No	Losing some children will not make P lonely	P orphans farthest child(ren)
No	Losing some children would make P lonely	P invites P′

Table 1: Response of parent P, after receiving lonely parent message from P′

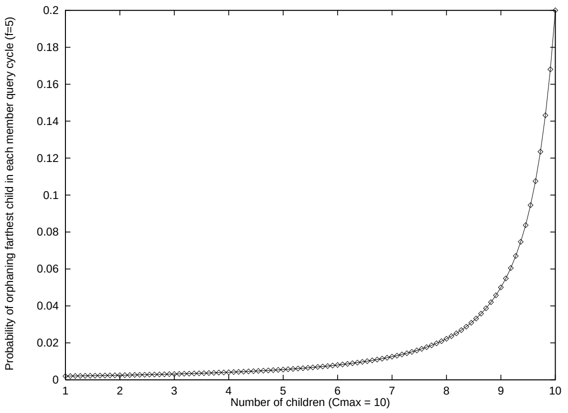

Figure 2: Equation 1 with $f = 5$ and $C_{max} = 10$

time. Because a slow node could be the only node in the network to carry the sought-after information, its response to database queries is vital. There will be no damage if it takes longer to respond than other nodes because the response is unicast directly to the client. If a node is too slow to respond to member queries, then it will be marked inactive and potentially orphaned. If it is consistently slow, it will not be able to acknowledge an invitation in time, either. In this case, a slow child may choose to become a parent. This activity is damaging to the whole system and we therefore must prefer that slow nodes be permanently relegated to child status.

Because of the difficulties involved in determining link bandwidth and latency, simple measures are the most effective strategy for dealing with slow children. The simplest approach is to allow manual determination of nodes which are connected only via slow links. A MASH server's administrator could prevent a node from choosing to become a parent, via a configuration option, for example. Such nodes would transmit adoption requests up to R_{max}, and, if they receive no invitation, orphan themselves. This method can also be used to prevent very slow servers and servers with very small databases from becoming parents.

A more complicated solution is for parents to collect statistics concerning their ability to service database queries. Parents could then exchange this information and make judgments concerning their own performance. Servers which experience difficulties receiving and handling database queries will observe that their number of serviced queries is significantly lower than that of the neighboring parents. Such nodes could orphan their children, and then themselves. This approach requires the addition of a statistics module for collecting the relevant information, a protocol for the exchange of this information with neighboring parents, and a decision making mechanism to determine each parent's reaction to the information. The statistics could include not only the number of queries serviced, but also the number of queries responded to by the parent. In this way, the set of parents could evolve from a random set of nodes into the set of nodes having the largest databases. Because database queries are routed to parents before they reach children, having the largest, most prominent FTP sites become MASH parents makes sense.

5 `MASH` — An Implementation

The `MASH` prototype[5] was built directly on top of `march`, using code from the latter to deal with client queries, establish multicast and unicast communications, and filter duplicated client queries. The client software itself is almost completely unchanged from `march`. A highly modular design ensures that improvements to the group maintenance protocol can be implemented without impacting the database or query/response system.

5.1 Selecting and Tracking Children

Parents use a greedy algorithm when selecting children. They will never abandon a child unless they are certain that a replacement exists. On the other hand, parents will readily orphan a child in exchange for an assured better child. The mechanism that keeps track of all active, inactive, and unverified children for a parent is a data structure called the CTABLE.

5.1.1 CTABLE Structure

The maximum number of children a parent may have is given by the predetermined constant C_{max}. This is the number of entries in the CTABLE. The CTABLE keeps track of active children, inactive (dead) children, and unverified (pending) children. Each child in the CTABLE has five fields associated with it: identifier, active bit, distance, timer, and last QID. The Query ID (QID) is a number uniquely identifying each query cycle that a server initiates. For any child on the CTABLE, its status can be determined as follows:

active — if the active bit is set

unverified — if it is not active, and its timer has not elapsed, and it successfully acknowledged the most recent query (or was just invited, indicated by a QID of -1).

dead — if it is not active or unverified.

5.1.2 Updating the CTABLE

The CTABLE is updated on any one of three events:

Member Acknowledgment Arrives. For the acknowledging child's entry, if the QID field in the member acknowledgment message is equal to the current QID (or it is a special value used to indicate that the child is responding to an invite message), then set the active bit to 1, set the distance from member acknowledgment message's hop count field, and set the last QID field to current QID.

Adopt Arrives. Examine the CTABLE, add the child if there is room (removing the furthest dead child if necessary). If the table is full, but the potential child is closer than the farthest live child, expand the table and issue it an invitation. Schedule a Table Reduction to occur giving the potential child enough time to accept or decline the invitation (silence is an implicit declination).

Table Reduction. Remove the worst child. If there are dead children, remove the one with the farthest distance. If not, remove and send an orphan message to the live child with the farthest distance.

The act of growing the table means that, temporarily, the table size exceeds C_{max}. This occurs when the CTABLE is full, all children are live and there is an adopt request with a small distance (at least a one-step improvement over the farthest child in the table). In this case, the parent extends an invitation to the preferable child, but greedily retains all of its children because the act of issuing an invitation does not ensure the child will accept it. The child is likely to receive many invite messages from all of the neighboring parents and it will choose the best one. So, instead of killing a perfectly good child for the chance of getting a better child, extend the invitation tentatively, and schedule a table reduction to occur after all children have had a fair chance to respond to outstanding queries.

5.2 MASH Protocol

`MASH` servers communicate via five message types. These messages are passed either via the global multicast group address, the local group address, or unicast via a TCP connection. The message types are:

Adopt — Multicast on the global group by newborns and orphaned children in an expanding-disc fashion. Used to find the nearest parent willing to adopt it.

Invite — Unicast from a parent to a newborn or orphan in response to hearing its Adopt packet. Contains information necessary for the newborn to join the parent's group.

Memb_Query — Parent multicasts to its children using incremental TTLs so that children can know the high level server is still alive and to solicit Memb_Ack messages.

Memb_Ack — Unicast to the parent by children in response to a Memb_Query. This allows the parent to know that the child is still alive, and what its distance is. This message is also sent by a newborn in response to an Invite message.

Orphan — Unicast from a parent to one of its children to prune off that child. A child that receives an Orphan message will become a newborn, acting as though it had just been restarted.

6 Summary

The advantages of an expanding-wheel search appear to be quite significant for a highly connected multicast-capable network densely populated with servers. However, such a network does not yet exist. Support for multicast in the Internet is still partial. Also, it is difficult to deploy a self-adjusting mechanism that is in an experimental state.

6.1 Analysis

The first step to be taken towards deployment of a distributed multicast directory service would be to ground the soundness of the idea in simulation. Our efforts are currently focused in developing a suitable simulation testbed in order to evaluate the appropriateness of the group formation protocol and to quantitatively measure the amount of traffic reduction over the `archie` and `march` approaches. Although the simulation engine was not complete in time to include the results in this paper, the MASH group is currently involved in modifying the ns-2 simulator[6] to simulate arbitrary applications running in any network topology.

While the modifications are still unfinished, we have performed hand-simulations in a random multicast-capable network topology. In an experiment with 104 nodes, a centrally located client, sought-after information 6 hops distant, 9 groups, average group size 10, and 1-hop radius increments, the MASH approach showed a 27.9% reduction in the number of links traversed compared to the `march` approach. When complete, the ns-2 simulation results will be made available on the web[5].

6.2 Further Observations on the MASH Paradigm

MASH uses a hierarchical approach with group self-organizing capability to implement the recovery of distributed directory information. We recognize that this paradigm can be extended to recovery of other similar, distributed information. For example, indexing words from web pages, audio/video program listings, library metadata, or any other indexable information stored on many machines. For any of these applications, the basic idea is the same: let the machine which stores the information also provide a searchable index which the clients may query via multicast allowing the distributed indices to respond if they may. By doing so, a client can get up-to-date and complete information without encountering a bottleneck problem.

To aid in querying distributed information servers, MASH organizes the information servers into hierarchical groups. This approach reduces query traffic a great deal (compared to expanding-disc approaches, such as `march`). The group organization is dynamically self-adjusting, resulting in the lightest possible query traffic.

We recognize that the mechanism of group self-organization adopted in MASH is also of general interest in the Internet community. The protocol used in MASH is very simple, yet robust. The control message are infrequent, except at a server's "birth." Yet the system responds to changes in the topology or community of servers automatically. The act of orphaning a server is viewed as a safe response to any non-optimal grouping which may (temporarily) arise. Such a simple, competitive scheme may prove useful for self-organization in other applications. Because our implementation is highly modular, the self-organizing component may be extracted easily, for use in other systems.

6.3 MASH with a Hierarchy of More Than Two Levels

What we have proposed above is a two-level only hierarchy. We are currently researching the design of a MASH with more than two hierarchical levels. In order to avoid creating hotspots at the highest-level servers, we are considering bottom-up propagation of query messages. Such a search would still retrieve the topographically closest responses with the added benefit of exponentially increasing the number of new servers reached in each iteration.

7 Acknowledgments

We would like to thank Dr. Lixia Zhang of UCLA for her guidance, many poignant insights and excellent advice. We are also grateful for many valuable comments and feedback made by the reviewers.

References

[1] A. Emtage and P. Deutsch. "Archie: An electronic directory service for the Internet." *Proceedings of the Winter 1992 Usenix Conference*, pages 93-110, January, 1992.

[2] Kashima, Hiroaki, et. al. "Searching Internet Resources Using IP Multicast." *INET '95*, August, 1995.

[3] Rovers, Perry. "Anonymous FTP Frequently Asked Questions List." Copyright 1993-1995, Perry Rovers. FTP Host: `rtfm.mit.edu`, Directory: `/pub/usenet/news.answers/ftp-list/faq`.

[4] Deering, S. "Host Extensions for IP Multicasting." Request for Comments 1112, August 1989.

[5] Rosenstein, Li, and Tong. "The Multicasting Archie Server Hierarchy." Project Home-Page. URL: `http://www.cs.ucla.edu/~adam/mash.html`.

[6] "ns-2: The LBNL Network Simulator," Lawrence Berkeley National Laboratory, URL: `http://www-nrg.ee.lbl.gov/ns/#version2`