A Highly Available Resource Location Service

Andrew Berkheimer, John Gu, Kenneth Yu
{andyb, johngu, kennyyu}@mit.edu

ABSTRACT

This paper describes the design and implementation of a decentralized and highly available resource location service. The service allows clients to locate servers providing a desired resource. Resources supported by the service are named independently of their physical locations. Additional resource servers may be dynamically added in multiple locations to provide greater availability for a resource, without the need for central authorities to update the location service. In fact, no centralized components of any kind are necessary to implement the service. All servers providing resources are expected to participate in the resource location service, ensuring that the availability of the location service scales commensurately with the resources that it supports.

1 - Introduction

Resource location services (RLS) allowing clients to find a server providing a desired resource are a necessity in implementing any network service. Making these network services highly available has become critically important as these services become a part of every day life. But as more work is done to improve network service availability, one weak link has been created: current resource location services are often less highly available than the network services which they support.

Additionally, most current RLS's have significant limitations that restrict their usability and scalability. Some require all of the servers for a resource to be in the same physical location, making those RLS's useless for services which provide high availability through the use of multiple servers at distributed locations. Others require a single central authority to add or remove servers for a resource, and/or require a single authoritative source of information. Another common RLS limitation is the need for user intervention, such as when end users are provided with a list of servers and asked to manually attempt to make the best server selection.

To address this weak link in creating highly available services, we propose to create a distributed resource location service that allows anyone to join in and quickly become an intimate part of the service. Each server providing some resource joins the service by running a small daemon and announcing itself and what resources it has to a few other nodes in the service. Servers join the service by learning the network address of a single node in the service. Once connected to a single node, a server requests to learn about more nodes in the service, so that it will have alternatives to fail over to in the event that its initial connection to the service fails. Then the server, in addition to providing its own resource, becomes a node in the distributed resource location service as well. Clients connecting to any single server will have access to the locations of all resources available in the service.

High availability is achieved in the service by attempting to propagate as much information about resource location to as many nodes as possible, and attempting to learn about as many nodes in the service as possible. To address scalability, most nodes will only interact with a few other nodes on a regular basis. But by having knowledge about the existence of many other nodes, a server can continue to participate in the service and maintain connectivity even when its regular links to the service fail.

It is important to note that this service provides no guarantees about the validity of the information that it provides to clients. After performing resource location, applications using this service must verify the validity of servers for themselves through their own means. This approach has a number of benefits. First, it allows different applications to use their own mechanisms for checking server validity instead of trying to impose a one-size-fits-all approach in the location service. Second, it greatly simplifies the location service itself.

For the rest of this paper, we will continue by discussing related work and background information in Section 2. In Section 3 we describe the design and implementation of our resource location service. Section 4 discusses some performance analysis of our system. Section 5 concludes with a discussion of our findings and possible directions for further work.

2 - Related Work

There are many systems in use today to direct users to the appropriate server for a resource. At the most basic level are host addresses in the Internet Protocol. IP addresses have the advantage that their resource location service is highly available: if an IP address exists and a route to the host exits, then the network layer will reliably route to it. But these addresses are tied to a single network: two servers on two networks operated by different organizations cannot have the same IP address, even if they provide the same resource. So while its availability as an RLS is good, IP addressing alone cannot support the needs of highly available applications with resources distributed in more than one location.

The domain name system addresses this inadequacy of IP addresses by providing a service that resolves text names into one of more IP addresses for a client. The service can simplistically be used to support highly available applications by including more than one IP address in its response to a client. More complex DNS server implementations may vary the response depending upon the clients' location [wcwch]. But the DNS service itself has availability problems. Information about the location(s) of each resource is stored in a limited number of authoritative name servers, typically eight or fewer servers. Some additionally availability is added by allowing clients to cache responses for a server specified Time-To-Live period. But the desire to quickly propagate updates when the list of addresses for a resource changes is an impediment to the usefulness of using the TTL to increase availability of the DNS service.

Another popular form of resource location is the ubiquitous *mirror list* commonly used by many web publishers. In these systems, the author includes with their document a list of mirror sites which also contain a copy of that document. This list is presented to the end user, who manually chooses a mirror site from the list by taking a guess as to which server will serve them best. Besides the obvious usability deficiencies of forcing users to make this decision, this method also depends upon a centralized server, as the mirror list acts as a central directory. A failure of the server containing the mirror list makes the resource unavailable to most users.

Many other projects include alternative resource location services, but most are tightly integrated with the applications they are supporting. HTTP caching servers use a number of protocols to communicate with each other to locate resources cached in other servers, including ICP [icp] and cache summaries [summary], but these protocols require manual network topology configuration at each node. In the realm of general purpose information retrieval, some examples are Napster [napster], which requires a centralized directory, Gnutella [gnutella], which has significant scalability issues, and Freenet [freenet], which supports additional features beyond pure information retrieval, including the anonymity of information requesters and providers.

3 - Design and Implementation

3.1 - Goals

Our primary goal is to create a location service that will not be the "weak link" in implementing a highly available network service. This requires that the location service itself be at least as widely available as the services which it supports, that it is flexible enough to support any possible configuration of resource servers, and that adding or removing servers for a resource should not require updating a central directory. We meet these requirements by having a few fundamental principles in our design: running location service nodes everywhere, propagating information, decentralizing all operations, and overall simplicity.

3.2 - Running Nodes Everywhere

One basic problem in supporting the availability requirements of a potentially unlimited number of resources is that creating a location service to support those resources also requires a potentially unlimited number of nodes. To address this problem, we require that any server that wishes to be supported by our location service participate as a node in the location service. Each of these nodes is responsible for being the authoritative source of information for what resources it can provide, as well as participating in the handling of other resource location requests. This automatic addition of nodes to the location service ensures that the location service scales in step with the resources that it supports.

3.3 - Information Propagation

One of the simplest ways to make information highly available is to make that information available from as many locations as possible. In our design, this means that each node actively attempts to propagate what it knows about the network to other nodes participating in the service. What a node "knows" consists of a map of which servers serve a particular resource, and a list of what other servers participate in the location service. As each node learns about more nodes, it becomes less and less likely that a node can become disconnected from the rest of the network. In order to prevent a broadcast collapse of the network, this information is propagated in a controlled fashion, with a limit on the amount of information which any single node can send in a given time period.

Information is also propagated when clients make requests to the service for a particular resource. As the request for the resource propagates through the network and the response is returned to the client, all of the nodes that participated in the search also learn about the location of the requested resource. This additionally helps to ensure that resources which are requested more often will have their location information propagated faster.

3.4 - Decentralized Operations

To further enhance availability, the service must not require a centralized authority to oversee any of its operations. In particular, adding new nodes and removing nodes should simply be a matter of turning on the node and pointing it in the general direction of the network. This is

supported by simply allowing anyone to announce that they have a resource, and for anyone to propagate this data along to everyone else. When a node receives information about a resource from more than one other node, it will take the union of the two sets, so no information can be lost. There is no need to instruct a node to remove a server from its list for a resource - it can determine for itself that a node has been removed when that node no longer responds to other requests.

3.5 - Simplicity

Our service is not very complex: the protocol is a fairly simple one consisting of two types of client-server interaction, along with some simple heuristics in nodes to manage their state. In particular, our service does not try and make any guarantees about the authenticity of whether a node actually has the resource that it says it has. It is up to the application to make that determination in whatever manner it wishes, once it has the location information returned by our service.

There are a number of examples that illustrate how this could be applied. A service that uses SSL can validate the resource name in the server certificate against the server name that it requested from our service. Similarly, a client in the SFS read-only file system [sfsro] can validate the signed fsinfo block from a server with the public key embedded in the name that was used to locate that server.

3.6 - System Overview

Our resource location system was designed in the context of providing a highly available server location information for the SFS read-only filesystem. However, we designed it to be as independent of this system as possible - the only tie to the system is our use of SFS hostids [sfs] as names for resources. As far as our implementation is concerned, the name of the resource is an opaque string.

The system itself consists of identical software running on each server that provides a resource (in our case, this resource is a SFS read-only database), as well as on clients that wish to access those resources. The software is responsible for interacting with other the nodes using the protocol described below, and for maintain a resource table which maps resource names to server addresses. This resource table is stored on disk so that it is persistent if the software is stopped or restarted for any reason.

3.7 - Basic Message Format

All messages sent between system components use the same basic format, consisting of a small header followed by a payload field. Each message contains a unique message id, to be used by nodes to associate responses with their corresponding requests. Messages also contain a field indicating their type. The payload itself contains message specific data whose meaning varies depending on the message type. For example, request messages have a TTL field, to place a limit on how long a message should be forwarded before giving up - this field is decremented by each node in the path of a message.

3.8 - Requesting a Resource

In order to request a resource, a client chooses an appropriate host from its routing table - if it has an entry for that resource, it uses one of those servers. Otherwise, it picks a server at random - and sends a message with the type *ResourceRequest*. The client can set the TTL of this message to whatever it wants, the default value is 10. The payload of this message is a string describing the resource to be located, the TTL, and the address of the client making the original request.

Each node that receives this message will first check to see if it has the resource being requested. If so it will send back a response indicating a successful lookup, with its own address in the payload. This response will be propagated back through the chain of requesting nodes, as well as being sent directly to the original client. Each node in the return path incorporates the successful servers' address into its routing table.

If a node does not have the resource being requested, it will first decrement the TTL by 1. If the TTL reaches 0, it will send back a response indicating that the TTL expired before the resource was found. If the TTL does not reach 0, the node will forward the request using the same logic as the original client - by choosing an appropriate node to forward to from its routing table.

Figure 1. below is an example illustrating a request made by the client, and the consequent order of messages the servers exchange as a result.

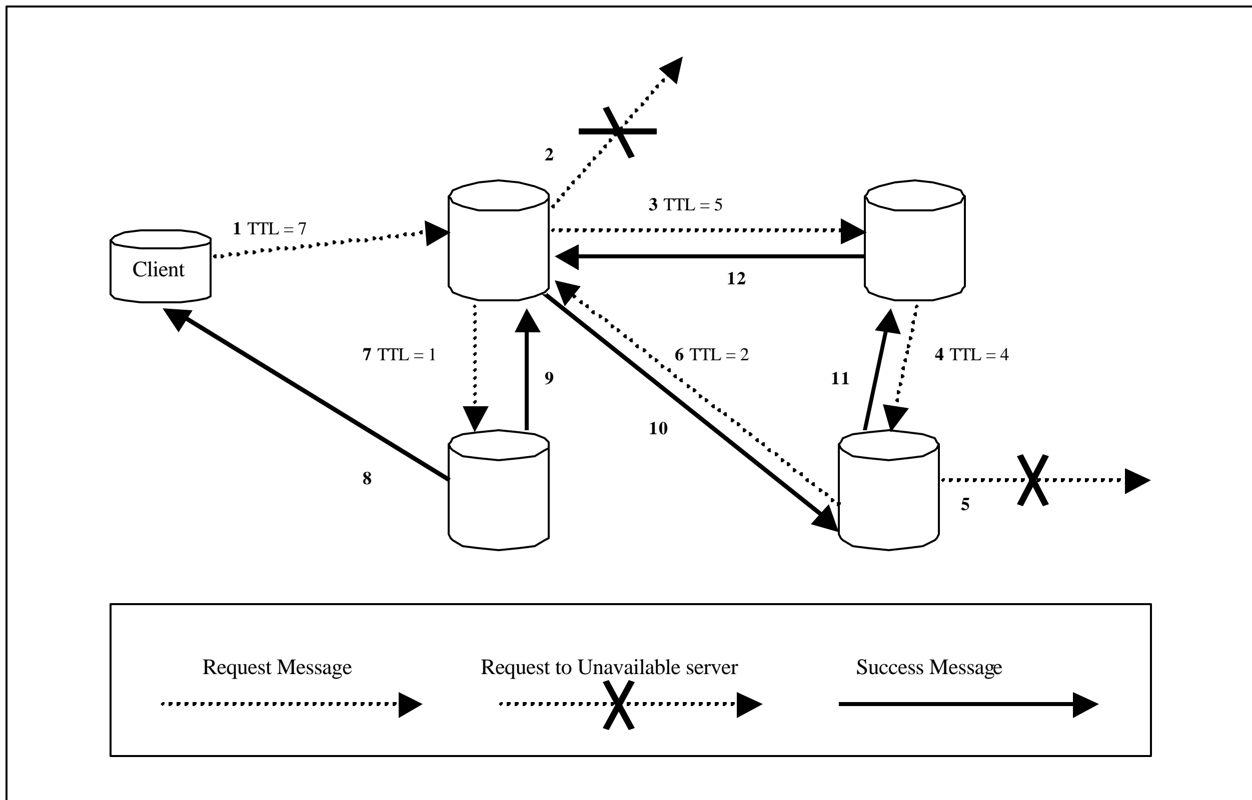

Figure 1. A sample request made by a client connected to the service

When a node receives a response indicating that the TTL expired, it will relay that response back to the requesting node. Each node also places a timeout on waiting for a reply to a request - if no reply is received within 10 seconds, the node gives up on that request, decrements the TTL by 1, and sends the request to another node. Again, if the TTL reaches 0 without success, the node will send a TTL expired response.

3.9 - Resource Table Propagation

In addition to resource requests, information in the system is replicated further through the use of selective information propagation. Whenever a node learns new resource information that it did not know before, it stores this in both the routing table and in a "new information" table. Once a minute (this interval is configurable, one minute is chosen as a typical example), the node will send a *ResourceTablePropogate* message type to a random selection of a few nodes from its routing table. The payload contains the information from the nodes "recent information" table, which may be truncated by the sender in order to prevent network flooding by limiting the amount of traffic each node generates.

The receiving nodes will incorporate this information into their own routing tables and the process repeats recursively. After a node sends out its "new information" table a few times, the table is cleared, since the information in the table is no longer new.

Figure 2. below shows the periodic broadcast every server performs that propagates new information they have acquired, either from resource requests or from servers entering the network.

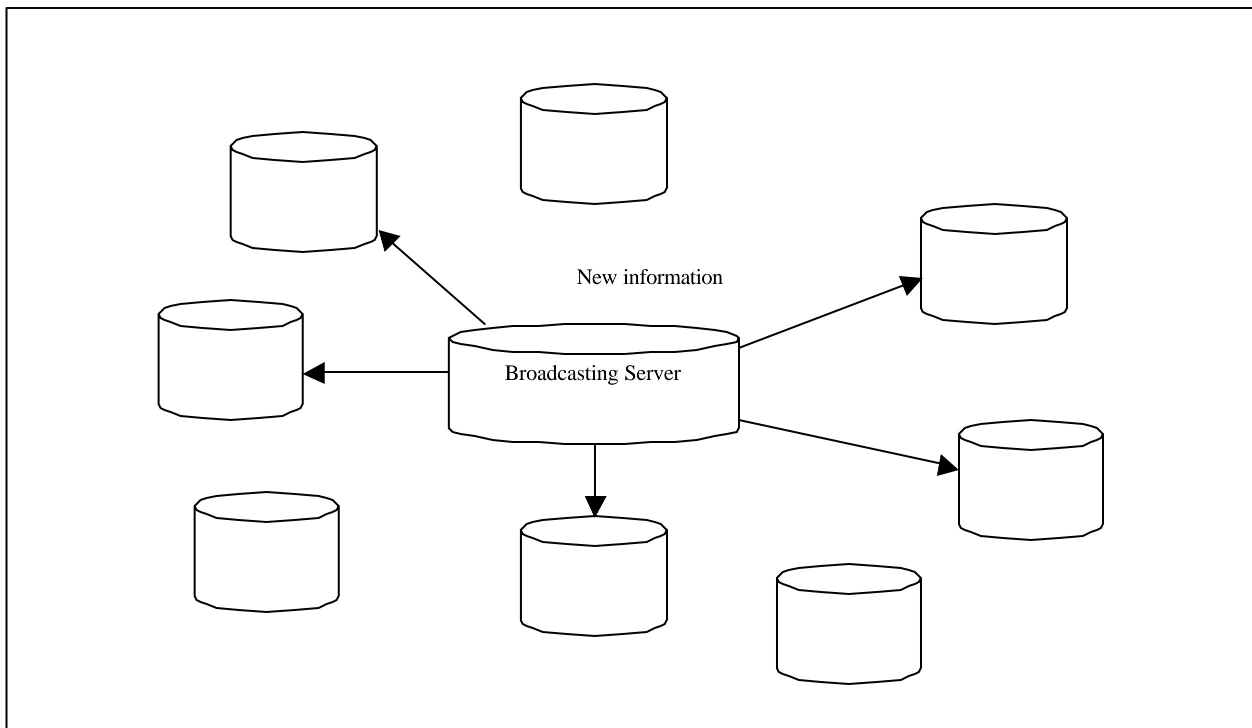

Figure 2. Resource Table Propagation – a server broadcasting new information to a random selection of nodes.

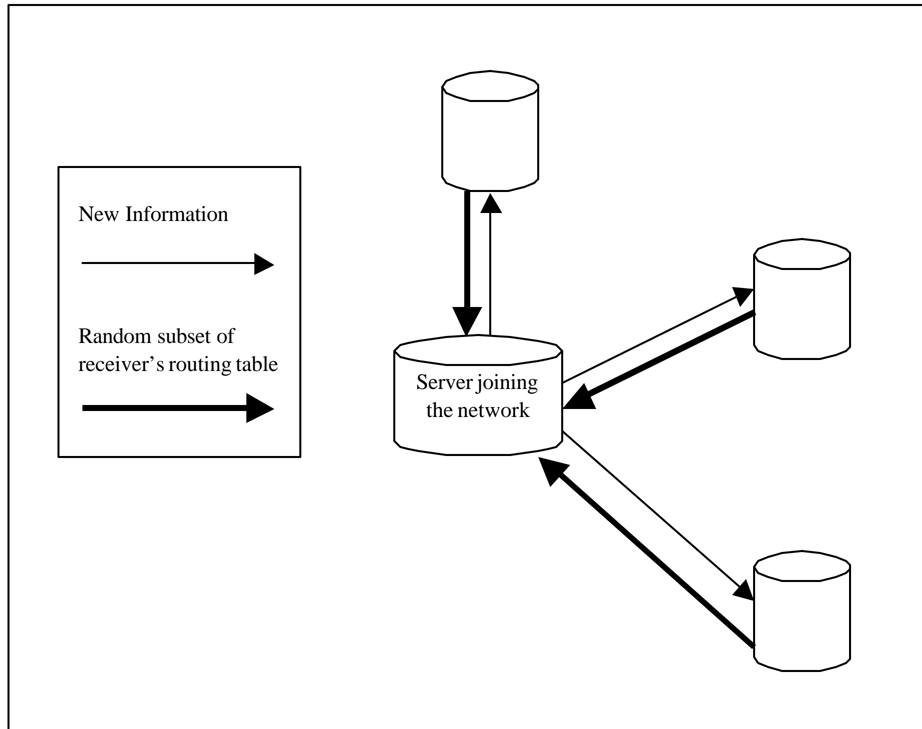

Figure 3. A new server entering the network for the first time and receiving subsets of other nodes' routing table

This message is also used by a new server entering the network that wish to announce the set of resources it holds. Figure 3. above shows a new server entering the network for the first time. When a new server enters the network, the propagating node announces the set of resources it holds, and indicates that it wants the receiver to immediately respond with a propagation of a random subset of the receiver's routing table. This mechanism allows new nodes in the network to quickly build up their knowledge of the network, while preventing network flooding.

3.10 - Code Base

The client and server programs consist of roughly a thousand lines of C++ code which we tested and ran on Linux and Solaris. Our software used the asynchronous I/O library provided by SFS, making our implementation work much easier. Our interaction with the SFS read-only client and server software itself was kept minimal, in order to provide a clean interface and to make implementation simpler (the SFS read-only code base was in an unstable state during our work).

4 - Performance

4.1 - Experimental Setup

We tested our implementation by running the distributed resource location service on a cluster of ten servers. Each server was an Athena Solaris Workstation running on Sun's Sparc Ultra 5 hardware. The cluster provided services for 25 database keys, each identifying a different SFS database. Each server served about 3 to 4 keys, such that each SFS database was redundant

against individual server failures. The routing tables of all servers were allowed to converge before testing.

During testing, the servers were flooded with requests for locations of database keys. Requests were generated by a client program that randomly picked a database key and a server to contact for the location of that database key. Database keys were picked in such a manner that some would be requested much more frequently that others. We hoped this scheme would more closely simulate real world requests where certain resources are much more popular than others.

The client ran for 30 seconds, during which time it recorded the number of replies it received from the ten servers, and how many of those replies successfully returned a location for the requested service. An arbitrarily selected server from the cluster was then shutdown, and the test repeated. The process continued until only one server remained. The entire test was then repeated for two clients simultaneously generating request to the cluster of servers. The results of our experiment are summarized below.

4.2 - Evaluation

Figure 4. below summarizes the performance of our service under varying conditions. The X-axis plots the number of servers running at a given time, and the-Y axis plots the number of total replies received by client programs. The graph demonstrates clearly that performance increases with the number of servers providing the resource location service.

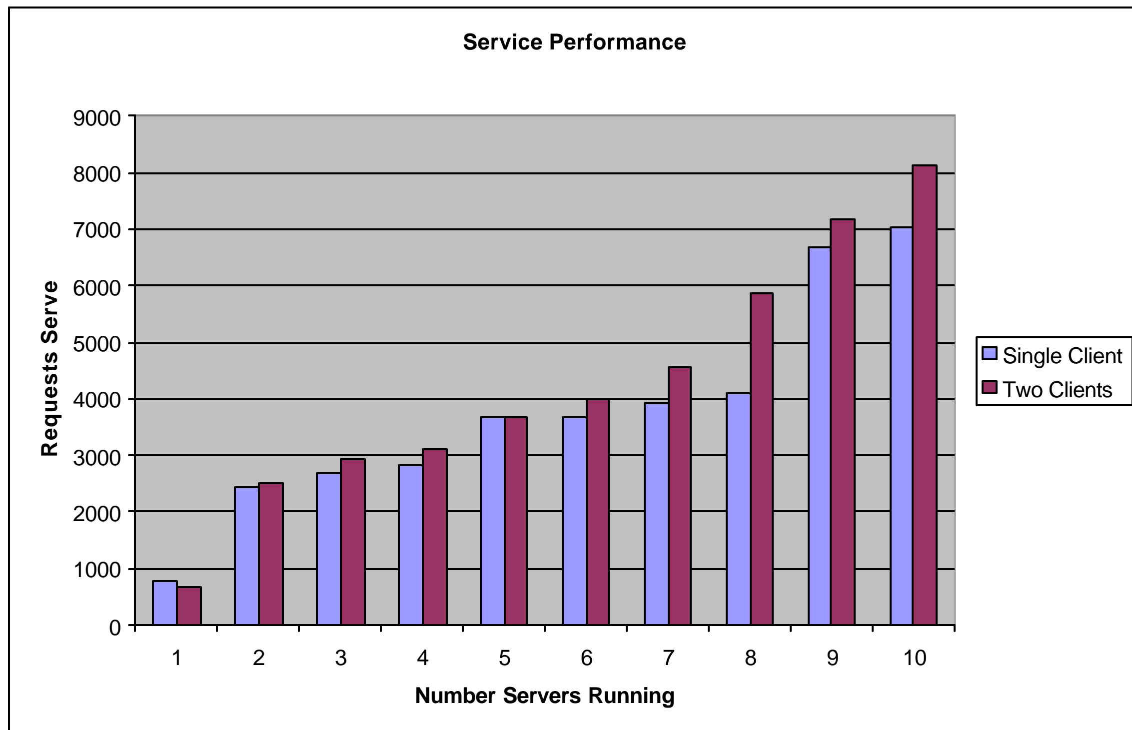

Figure 4. Performance of Distributed Location Service vs. Number of Participating Servers

Figure 5. shows the percentage of client requests that successfully returned the location of a requested resource. This figure demonstrates the increased availability supplied by our service. All client requests continued to be successfully answered even when two servers were taken down. Only when all servers serving certain SFS database keys have been taken offline did the client receive unsuccessful replies to it requests.

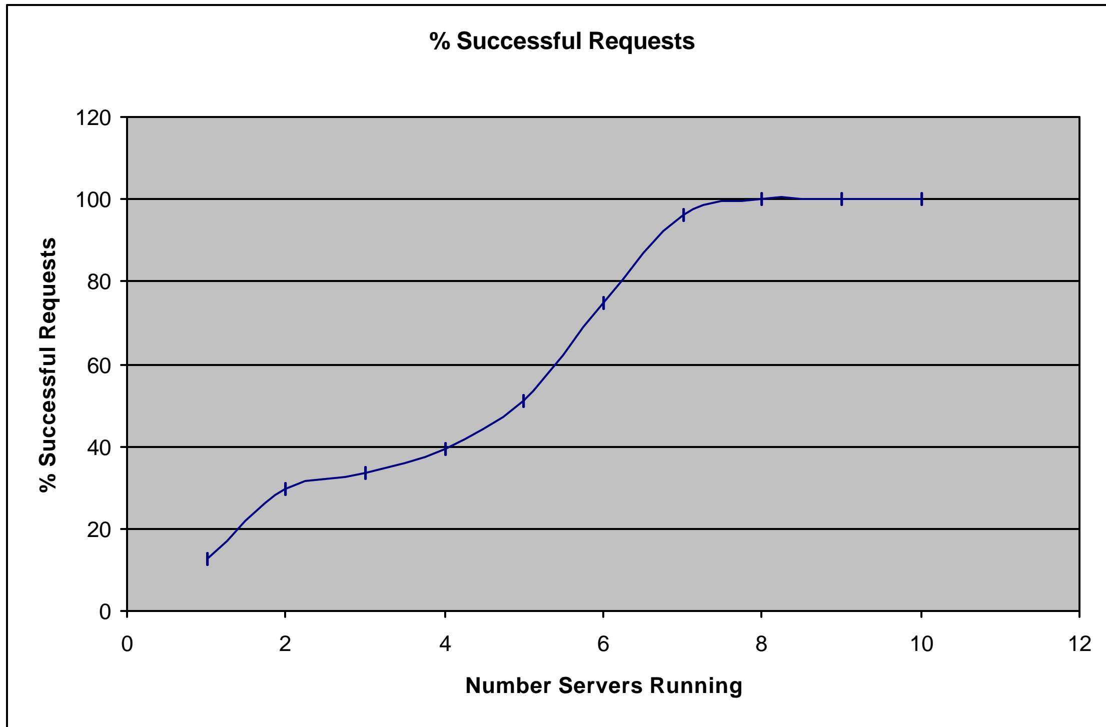

Figure 5. Availability Degradation as Servers Are Pulled Offline

5 - Discussion and Future Work

We feel that the data gathered from our experiment demonstrates two aspects of our distributed resource location service. First, the performance of our service scales with the number of participating servers. As more servers are added, the performance of our services increase accordingly as overall load is automatically balanced across different servers. Second, our service provides redundancy for against server failures. The replicated nature of our resource location service allows resources to be located even in the face of server failures. Only in the extreme case where all nodes serving a particular resource are offline will our service return unsuccessful replies to requests.

Our experiment also contains a number of weaknesses that deserves mention. First, our service was designed to scale to thousands of servers. Obviously, the numbers we tested were nowhere near that magnitude. Second, we have little guarantee that the requests produced by our client program do simulate real world traffic. Lastly, to simplify our experiment, we allowed our servers' routing tables to converge before initiating client requests. However, this situation would hardly, if ever occur in real world situations. Clearly, much more testing is needed to

identify the precise performance characteristics of our design. Nevertheless, given the time and resource constraints we faced, we feel that our experiment provided a flavor of our service's capabilities.

Although our current design and implementation achieves its stated goal of providing higher availability, it exhibits a number of weaknesses that should be addressed in future work. First, our current design eventually results in all servers sharing the same routing table. This is achieved incrementally through the broadcast protocol. This scheme, although efficient in helping to find the location for a particular resource, does not scale well to very large numbers of hosts. Although our use of efficient data structures - hash tables and binary trees - to hold our routing table allow our current design to scale to thousands of hosts, any more would present a problem. A more intelligent protocol allowing the nodes to organize themselves into a hierarchical structure might server as a suitable solution.

Second, due to time constraints, clients do not actively participate in the service in our current implementation. Our service currently only returns the first answer it finds to the client. We take the view that clients should be active participants in the service, and our original design reflected that view. Future revisions of the service should provide the client with many answers as possible. This allows the client to make the choice of which server to use, taking into account the potential for the location service's response to include extra information that may assist the client in choosing a server.

Third, our current mechanism for locating a particular resource is also ignorant of network and server load conditions. This may result in returning to a client a location for a resource that has low bandwidth, under heavy load, or both. Future work would involve making the distributed location mechanism aware of these conditions, and optimizing based on them. Lastly, our current design is extremely vulnerable to security attacks. For example, a rouge server broadcasting invalid information might degrade the performance of the service. This should be addressed in future revisions as well.

Bibliography

[freenet] I. Clarke, O. Sandberg, B. Wiley, and T. W. Hong. Freenet: A Distributed Anonymous Information Storage and Retrieval System. In *ICSI Workshop on Design Issues in Anonymity and Unobservability*, Berkeley, California, July 2000.

[summary] L. Fan, P. Cao, J. Almeida, and A. Z. Broder. Summary Cache: a Scalable Wide-Area Web-cache Sharing Protocol. Technical Report 1361, Computer Science Department, University of Wisconsin, February 1998.

[sfsro] K. Fu, M. F. Kaashoek, and D. Mazieres. Fast and secure distributed read-only file system. In *The Fourth Symposium on Operating Systems Design and Implementation*, San Diego, California, October 2000.

[gnutella] Gnutella, http://gnutella.wego.com/, 2000.

[wcwch] D. Karger, A. Sherman, et al. Web Caching with Consistent Hashing. In *The Eighth World Wide Web Conference*, Toronto, Canada, May 1999.

[sfs] D. Mazieres, M. Kaminsky, M. F. Kaashoek, and E. Witchel. Separating key management from file system security. In *The 17th ACM Symposium on Operating Systems Principles*, Kiawah Island, South Carolina, December 1999.

[napster] Napster, http://www.napster.com/, 2000.

[icp] D. Wessels and K. Claffy. Internet Cache Protocol (ICP), Version 2. IETF RFC 2186, *http://www.ietf.org/rfc/rfc2186.txt*, September 1997.