Overlook: Scalable Name Service on an Overlay Network

Marvin Theimer
Michael B. Jones

April 2002

Technical Report
MSR-TR-2002-48

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052

Published in
Proceedings of the 22nd International Conference on Distributed Computing Systems,
Vienna, Austria, IEEE Computer Society, July 2002.

Overlook: Scalable Name Service on an Overlay Network

Marvin Theimer and Michael B. Jones

Microsoft Research, Microsoft Corporation
One Microsoft Way
Redmond, WA 98052, USA

{theimer, mbj}@microsoft.com
http://research.microsoft.com/~theimer/, http://research.microsoft.com/~mbj/

Keywords: Name Service, Scalability, Overlay Network, Adaptive System, Peer-to-Peer, Wide-Area Distributed System

Abstract

This paper indicates that a scalable fault-tolerant name service can be provided utilizing an overlay network and that such a name service can scale along a number of dimensions: it can be sized to support a large number of clients, it can allow large numbers of concurrent lookups on the same name or sets of names, and it can provide name lookup latencies measured in seconds. Furthermore, it can enable updates to be made pervasively visible in times typically measured in seconds for update rates of up to hundreds per second. We explain how many of these scaling properties for the name service are obtained by reusing some of the same mechanisms that allowed the underlying overlay network to scale. Finally, we observe that the overlay network is sensitive to bandwidth and CPU limitations.

1. Introduction

Name services [1, 13, 12] are widely recognized as being one of the key building blocks for distributed applications. In their most general form, they allow clients to present a name and obtain a set of data associated with that name.

Our interest in scalable name services stems from the Herald project's effort to build an Internet-scale event notification service [3]. Such an event notification service requires an Internet-scale name service to manage the name space for its event topics. The Overlook name service is designed to fill this role.

Furthermore, such a name service requires two features not found in traditional large-scale name services. One requirement is that updates to the name space must become globally visible relatively quickly (in seconds rather than hours or days). We want to enable applications to dynamically generate or change event topic names' contents on the fly and then use them almost immediately thereafter.

Another requirement is that the name service be able to handle high variations in the popularity of various parts of the name space and high variations in the lookup request rates made to various parts of the name space. In other words, the name service must support "flash crowds" that suddenly focus on one or a few names—perhaps only briefly—that were never before of interest.

Although these requirements stem from our focus on Internet-scale event notification, we believe that they represent capabilities that would be generally useful. Hence we present Overlook as a general name service design rather than one that is specific to just event notification services.

1.1 Use of Overlay Networks

Existing scalable name services, such as DNS, tend to rely on fairly static sets of data replicas to handle queries that cannot be serviced out of caches on or near a client. Unfortunately, static designs don't handle flash crowd workloads very well. We want a design that enables dynamic addition and deletion of data replicas in response to changing request loads. Furthermore, in order to scale, we need a means by which the changing set of data replicas can be discovered without requiring global changes in system routing knowledge.

Peer-to-peer overlay routing networks such as Tapestry [24], Chord [22], Pastry [18], and CAN [15] provide an interesting means of achieving the desired design. Such networks organize a collection of cooperating hosts so that they can route messages among themselves in the face of node and network link failures while maintaining only relatively small overlay routing tables and scaling to large numbers of participant nodes.

More interestingly for our work, these networks can be employed to implement the equivalent of a distributed hash table. Furthermore, replicas of hash table entries can be created along the queries' routing paths such that they can be found without requiring global state changes. Thus, query loads can be diffused both by distributing table entries among multiple nodes and by replicating popular entries along the routing paths that are followed to find the "master" copy of a table entry.

1.2 Peer-to-Peer or Not?

The use of overlay networks has become closely associated with the idea of peer-to-peer computing. Such systems take advantage of the ability to place content anywhere among cooperating nodes. However, as [20] makes clear, not all peers are equal. In that study, among other results, we see that the median peer connection time was only about an hour, that downstream bandwidths tend to be in the range of 56Kbits/s to 10Mbits/s, and that the upstream bandwidths are often a factor of 5-10 times worse than that.

As we discuss in our experimental results section, we found that the scalability of our peer-to-peer server system is sensitive to both the network bandwidth provided among the participating server nodes and the CPU processing rate of those nodes. Because of this, Overlook's design targets a setting of managed server machines connected by high-speed network links rather than one of arbitrary client machines. An example of such a setting is the Akamai "edge computing" model, in which server machines are spread among ISP data centers throughout the world.

Overlook runs peer-to-peer routing algorithms among name server nodes, but assumes that each name service client will separately connect to some server node and use it as a proxy for its requests. Thus, clients do not participate in the peer-to-peer system themselves and our design can be viewed as being peer-to-peer among the server nodes but not end-to-end to the client nodes.

In the remainder of this paper we present design requirements in Section 2, describe the design of the Overlook scalable name service in Section 3, and present and discuss experimental results obtained for Overlook via simulations in Section 4. Section 5 reviews related work, while Section 6 discusses future work and Section 7 presents a summary of our work and draws several conclusions.

2. Design Requirements

2.1 Desired Functionality

Overlook's goal is to provide an Internet-scale hierarchical name space with the following functionality:

- Directories can contain both (name, value) pairs and names of subdirectories.
- Clients can look up a value stored in the name service by presenting the fully-qualified directory name and a name within that directory whose value is to be retrieved.
- Directories can be updated by modifying or deleting an existing (name, value) pair, adding a new (name, value) pair, adding a new subdirectory, or deleting an existing subdirectory.
- Clients can enumerate a directory's contents. An enumeration returns an unordered list of directory-relative names, one per directory element.

2.2 Scalability, Performance, and Availability

The applications we intend to use Overlook for require the following performance and availability characteristics from it:

- The service should scale to allow many millions of clients to query and update the namespace during the course of each day. Scalability should be achievable by simply adding more servers.
- Queries against a given (name, value) pair should complete within less than a second, even when unavailable in a client's local cache. The popularity of any given (name, value) pair or directory should not affect the time required for queries against that (name, value) pair or directory, except during brief transient periods when their popularity is drastically changing.
- Directory updates should typically be visible to all clients within a short period of time, such as a few seconds.
- The service should be able to execute hundreds of thousands of queries per second against a single directory or (name, value) pair. (100,000 lookups/second corresponds roughly to having everyone on the planet query a particular directory or (name, value) pair once during a single day.)
- The service should be able to execute hundreds of updates per second against a single directory or (name, value) pair. (100 updates/second corresponds to having a directory of all machines in a company the size of Microsoft—about 100,000 machines—and having each machine update its entry once every 15 minutes.)
- The service should be able to survive the failure of any f machine nodes with no loss of data, where f is a pre-specified value.

3. Scalable Name Service Design

In this section we present the design of Overlook, a scalable distributed name service built on top of a peer-to-peer overlay routing network, such as Pastry. We first give a brief overview of salient aspects of Pastry and describe how we exploit its various characteristics to obtain both generalized load diffusion as well as flash crowd support. We then describe how to deal with network congestion, how to effect updates that will become globally visible quickly, and how to obtain fault tolerance against server node failures.

3.1 Brief Introduction to Pastry

The properties of peer-to-peer overlay networks, such as Pastry, that make them attractive for building a scalable name service are:

- They provide a scalable general-purpose routing mechanism that can route messages between participants in the system.
- They provide a fault-tolerant routing.
- They enable placement of application-level services at overlay routing nodes.

Paraphrasing from the published literature: Pastry forms a secure, robust, self-organizing overlay network on top of an underlying network such as the Internet, an organization's Intranet, or a data center's local network fabric. Each Pastry node has a unique, 128-bit nodeId and the set of existing nodeIds is uniformly distributed. Nodes can select suitable nodeIds by, for instance, taking a secure hash of their public key or IP address.

Destinations in Pastry are designated by means of 128-bit keys. Given a message and a key Pastry reliably routes a message to the Pastry node with the nodeId that is numerically closest to the key, among all live Pastry nodes. Thus, to implement a distributed hash table on top of Pastry involves storing each table entry on the Pastry node that is numerically closest to the hash key associated with the entry. Table entries are found by routing request messages via the hash key of each desired entry.

Pastry overlay networks are designed to be scalable. Routing is implemented by sending a message to another Pastry node that is numerically closer to the destination than the current one. Each node maintains a routing table whose entries are selected so that the expected number of routing hops to reach a final destination is O(log N) in a network consisting of N nodes. By selecting routing table entries in a manner that also reflects network topology considerations, Pastry is able to route messages in a manner that mostly avoids sending messages through far-away nodes of the network. Furthermore, the routing tables required in each Pastry node are small, having O(log N) entries, where each entry maps a nodeId to the associated nodeId's IP address.

Pastry routing is also robust in the face of node failures. With concurrent node failures, eventual delivery is guaranteed unless *l*/2 nodes with *adjacent* nodeIds fail simultaneously (*l* is a configuration parameter with typical value 16).

Finally, an important feature of Pastry that we take advantage of is that each node along a message's route passes the message up to a registered application. The application can perform application-specific computations related to the message and then inform the underlying Pastry layer whether or not to continue routing the message onward to its intended final destination.

3.2 Basic Design

At its most fundamental level, Overlook employs a distributed hash table to diffuse load across a set of server machines that are interconnected by a Pastry overlay network. Table entries are the directories of the hierarchical name space. The hash key for a directory is obtained by applying a secure hash to its full pathname to obtain a 128-bit hash key.

Name lookup, directory enumeration, and name update requests get routed via Pastry to the server node where the relevant directory resides. Directory creation and deletion requests require that messages get routed to two different nodes: the node hosting the parent directory and the node that will (or is) hosting the directory to be created or deleted.

As mentioned in the Introduction, we avoid the issues of dealing with the client machines that may be connected via high-latency, low bandwidth network links or that may be unreliable or corrupt by only maintaining a Pastry overlay network among a set of managed server machines. We envision that these might be distributed among ISP data centers around the "edge" of the Internet or aggregated within one or a few "mega-service" data centers.

All Overlook server machines register themselves with DNS. Clients interact with Overlook by finding an Overlook server machine via DNS and then sending requests to the selected server. Servers thus contacted act as proxies for client requests, forwarding them into the Pastry overlay network and subsequently relaying response messages back to the clients.

Clients can either randomly select an Overlook server from those registered in DNS or they can attempt to select one that is "nearby" if they have a means of gauging network distances, for example as Akamai's software does. If a client happens to pick a server that has crashed since it registered itself with DNS, then the client simply picks another one.

Overlook assumes the existence of client-side caches but permits name entries to be cached for only short durations so that updates can become visible quickly. In particular, unless a name's creator specifies a longer caching timeout value, it will be cached for only one second before being discarded.

3.3 Handling Popular Directories

A distributed hash table can diffuse the load of managing multiple directories across multiple server machines. However, it cannot diffuse the load directed at individually popular directories. For this we need replication.

Caching popular directories and/or their entries can provide the desired load diffusion if updates are not required to be quickly visible. However, if they are, then caches will have to timeout their contents quickly. This is all right if the average period between updates is comparable to the cache timeout interval. However, if updates occur significantly less frequently than cache invalidations then it is better to "push" updates rather than "pull" them. Hence Overlook employs a replication scheme in which replicas are kept up-to-date by means of update notifications.

Directory replicas are placed along the most congested Pastry routing paths leading to the "root" node for a directory so they will be encountered automatically by name lookup and directory enumeration requests. To do this, each server keeps track of the request rate for each directory of which it hosts a copy—either as a root instance or as a replica. The request rate is furthermore tracked according to which nodes the requests came from on their most recent routing hop. The nature of the Pastry routing scheme is such that requests only arrive from about log *N* different forwarding nodes, hence the storage costs of tracking this information are minor.

When a server node detects that the total request rate for a directory it is hosting exceeds a particular threshold, it initiates the creation of a new replica of the directory. Overlook does this by selecting the incoming forwarding node that has the highest recorded request rate associated with it and sending that node a create-replica request message. Once the request has been accepted, the server node records the fact that it has created a new child.[1] This information will be used to implement the update propagation scheme described in Section 3.5.

One could imagine selecting a replica candidate node based on both the above-described request rate criteria as well as its current CPU load. We have not explored this alternative yet, in part because it requires maintaining distributed state information in an up-to-date manner, whereas the request rate information can be maintained locally.

As mentioned, directory replicas are encountered automatically by name lookup and directory enumeration requests. This is because the Pastry routing layer passes each message up to the application layer on each node that a message gets routed through. When a request message is received by a node that hosts an instance of the directory to which the message is directed, the node processes the request and informs the routing layer not to forward the message any further.

Under sustained high load conditions, a node eventually creates a directory replica along all the incoming forwarding paths to it (for a given directory). At that point, it services only requests that it receives directly from clients. For an extremely popular directory, eventually every server node in the system contains a replica of it and services requests to it only from directly connected clients.

Server nodes discard local directory replicas when the request rate to them goes below a lower-bound threshold. To discard a given directory replica, a server node must inform both the "parent" server node that requested the creation of the replica and the "child" server nodes on which the server has created additional replicas. The child nodes must be informed that their new parent is the parent of the departing replica; the parent node must be told to add all the departing replica's children to its own child replicas list for the given directory.

[1] If the selected node has failed then the next most-trafficked forwarding node is selected and so-on.

3.4 Dealing with Network Congestion

The replication scheme described in Section 3.3 works well if server load is the limiting resource in a system. Network congestion is a second limitation on overall system capacity. Congestion can occur in two places: on the network links interconnecting server nodes and on the links between clients and servers. The latter form of congestion is something that we cannot do much about. However, the former form of network congestion can be alleviated by suitable replication of directory instances.

To activate replication of a directory due to network congestion, we require a means of detecting network congestion. Because server nodes act as proxies for client requests, they can measure the round-trip times for requests that they forward into the overlay network. When round-trip times for requests to a given directory are observed to be larger than a given threshold, a server (in its capacity as client proxy) sends a request-replica message to the server node that replied to the request. That server responds with a create-replica message back to the requesting node. In this manner, replicas can be "pulled" to other nodes across congested network links, in addition to being "pushed" to other nodes from overloaded nodes.

If server nodes were to request replicas immediately in response to high observed round-trip times, many less-popular directories might end up getting replicated as well as the popular ones creating the network congestion. This is because network congestion affects *all* request traffic that is being routed through congested links. To avoid this unnecessary replication, servers only initiate replica requests for directories for which they themselves observe high client request rates.

3.5 Propagating Updates among Replicas

Replicated directory management implies that updates must be propagated to all replicas. Ideally, updates would be applied in a strongly-consistent manner, so that all clients would see identical views of the name space at any given time. However, the cost of maintaining strong-consistency semantics across multiple replicas would be prohibitive for the number of replicas that can occur in Overlook for popular directories. To support scalability, Overlook applies client update requests by default in a weakly-consistent manner.

Clients wishing strongly-consistent update semantics can request them when a directory is created, but doing so will disable replication for that directory. This is fine for directories whose creators know that they will only be accessed by limited numbers of clients. The scalability limit for such directories will be determined by the ability of a single server node to process requests. Modern machines are typically able to process a few thousand network requests per second. Recognizing that hosts must process both lookup and forwarding traffic, it should still be possible to achieve our goal of hundreds of updates per second. Thus, in practice, all but the most popular directories could probably be declared to be strongly-consistent without affecting actual client request latencies in any noticeable manner. Such directories would, however, have to forego the opportunity of ever becoming popular.

For weakly-consistent directories, all update requests get routed to the root node for a directory. The root node forwards each update to all its child replicas. These, in turn, forward each update to their child replicas, until eventually every update propagates down the entire replica tree.

Update messages are sent using a reliable request-response messaging protocol. As a consequence, when replica nodes reply to update messages the sending node will know when an update has been successfully received, in the absence of failures. Replicas reply to an update message after they have forwarded the update message to their own child replicas and have either heard back from those replicas or those replies have timed out. Thus, after the root node for a directory hears a reply from each of its child replicas or has timed out, it knows that the update has been successfully propagated to all reachable replicas of the directory in the system. If we assume that replicas can infer when they are unreachable and take themselves out of commission, the root node can at that point reply to an update request with an indication that the update has successfully propagated to all visible replicas of the directory.

In order for the replica nodes for a given directory to detect that they are unreachable, they keep track of the last time they have heard from the directory's parent node. If a node hasn't heard from its parent within a specified time interval, it assumes the parent is unreachable and stops answering lookup requests, effectively taking its replica off-line. When contact with the parent is reestablished, the node requests that the parent send it all updates it has missed. If too many updates have been missed then the parent simply sends down a copy of the entire directory. For this process to work correctly, at least once each time interval the root node for a directory must either forward an update or send out a "heartbeat" message.

The timeout value that a server node should use while waiting for replies from updates forwarded to child replicas must be a function of how deep the tree of replicas currently is below that node. The appropriate value can be determined by monitoring the round trip times required for successful propagation of updates. Thus, server nodes keep track of how long it takes for update—as well as heartbeat—replies to return from their children. These times are used to dynamically adjust the timeout value used for waiting for future replies.

If one—or worse yet, several—extremely popular directories must also be updated on a frequent basis, then the update and heartbeat message traffic in the system may become substantial. Two things can be done to reduce the traffic should such a situation occur: updates can be batched together and the frequency of heartbeat messages emitted can be reduced.

For a directory's root node to be able to decide whether update and heartbeat traffic should be throttled it needs to know how many replicas exist in the system for its directory. This can be achieved by using update and heartbeat reply messages to roll up a summary of how many replicas exist for a directory.

To reduce the frequency with which heartbeat messages are sent out requires that replica nodes be informed that they must increase their heartbeat timeout interval. This can be done by including a timeout interval in heartbeat messages that tells replicas the length of the next timeout interval they should use. The penalty for this increase in heartbeat timeout interval is longer timeouts waiting for replies to update messages. That, in turn, will result in clients having to wait longer to hear that an update has successfully propagated to all visible directory replicas within the system.

3.6 Fault Tolerance

If we wish to tolerate *f* server node failures then we must immediately make at least *f* replicas of a directory in addition to its root node instance. Since the Pastry overlay network routes a message to the live Pastry node whose nodeId is numerically closest to the key value that a message is being routed to, the best place to put the *f*+1 replicas for a directory is on the server nodes whose Pastry nodeIds are closest to the key value corresponding to the directory. With this placement, "fail-over" from a failed root node for a directory to the next closest directory replica (in Pastry nodeId space) is automatic. Hence routing a request message towards the root node for a directory is guaranteed to find the same root replica among the *f*+1 replicas regardless of which *f* of those replicas might fail.

To ensure that updates are not lost as a result of root node fail-over, the order in which update messages are propagated to replicas must be the order in which fail-over would choose a new root node. Also, fail-over candidate nodes must be cognizant of the child replicas that the original root node has created so that they can reestablish contact with them after a fail-over has occurred. Fail-over is detectable on a new root node by keeping track of relevant activity in the node's local overlay routing table.

4. Experimental Results

In this section we present results obtained from running simulations of the Overlook design. We start with an overview of the kinds of experiments we used to evaluate our design. We then present basic scalability results, flash crowd results, and results quantifying the impact of updates.

4.1 Experimental Overview

4.1.1 Experimental Setup

Our experiments used an enhanced version of the packet-level discrete event simulator described in [4]. As well as modelling propagation delay on physical links, we added modelling of link bandwidths as well as both network queuing delays and delays caused by server CPU time consumption. A simplifying assumption we retained was that link congestion always causes packets to be queued rather than dropped.

The simulations ran on network topologies generated using the Georgia Tech random graph generator [23]. We used a transit-stub model to generate the core of the network and attached a LAN with a star topology to each node in the core. There were an average of 100 nodes in each LAN and each node in a LAN was directly attached by a stub link to the core node (as was done in [4]). Name server nodes were assigned randomly to LAN nodes with uniform probability and Pastry was configured with a leaf set size of *l*=16, a neighborhood set size of 32, and b = 4.

Our experiments were conducted on a topology with 600 nodes in the core and 60,000 LAN nodes. Links were simulated with these bandwidths: Transit-Transit links were set to 2Gbits/s and LAN links to 100Mbits/s. Transit-Stub links were simulated with three different settings: 45Mbits/s, 100Mbits/s, and 1Gbits/s. Stub-Stub links (those corresponding to links within ISPs) were simulated with bandwidths of 1.5Mbits/s, 45Mbits/s, 100Mbits/s, and 1Gbits/s. In the rest of the paper we shall use the following acronyms: T1 = 1.5Mbits/s, T3 = 45Mbits/s, LAN = 100Mbits/s, and GIGA = 1Gbits/s.

We modeled the service load of processing an application-level message as taking either 0.5 or 1 millisecond of CPU time each.

4.1.2 Experiment Design

We conducted a series of experiments designed to determine the performance and scalability limits of our name service implementation. In these experiments, we simulated clients sending requests to servers, with the requests arriving at randomly chosen times according to an exponential distribution with a specified mean. We measured request service times from the point at which a server, acting as a proxy for a client, forwarded a request into the server overlay network until the point at which that server received a corresponding reply message from the server overlay network. Thus, our request service times are representative of the latencies that clients will experience when their lookup requests *miss* in their local client-side cache. Note, however, that our latencies do not include the message times required from a client to its proxy server and back again. In the case where a request could be serviced directly on the originating server node, we accounted the service time to be the time required to process a request at the server, without any additional network message transmission time.

To determine the capacity of the system, we ran simulations with increasing request rates until the measured response times started rising noticeably. We applied a threshold to the observed service times, calling runs with average service latencies of under 800ms "good" and those with average service latencies of over 800ms "bad". The value 800ms was chosen as the breakpoint because we observed that for experiments not experiencing significant network congestion or server queuing delays, round trips would typically average less than 700ms. For experiments experiencing significant network congestion and/or server queuing delays the observed service times would quickly cross this threshold.

We ran successive experiments, varying the average request arrival rate for each configuration until we determined the boundary between "good" and "bad" runs to within a 10% ratio. The resulting lower and upper bounds on request rates are what is shown in all the graphed figures of this paper.

Parameter	***Meaning***
Stub-BW	Bandwidth of Stub-Stub links
S	Server CPU time per message
N	Number of overlay network nodes
D	Number of name service directories
R	Total number of requests per second
CPU-Rep	True if CPU load-based replication on
Latency-Rep	True if latency-based replication on
Updates	True if name update traffic present

Table 1: Experiment Parameters

The parameters to our experiments are shown in Table 1. Note that *N*, the number of overlay network nodes, represents the number of machines that are members of the overlay network eligible for containing directory information. For small values of *D*, though, not all of these nodes may actually contain directory entries. However, if replication is enabled (either with *CPU-Rep* or *Latency-Rep* or both), even though a particular server may not host the primary copy of a directory it may contain a replica of it.

If the *Updates* parameter is true, then for every 100 name lookup requests, a single update request will be made to change the value of a name, causing update traffic to all server nodes holding replicas of that name.

Finally, note that a parameter one might expect—the number of clients—is absent. Our model has clients making requests to servers that are members of the overlay network, but the clients themselves are not members of the overlay. In our experiments, we assume that client requests are uniformly distributed among the servers. Therefore, for simulation purposes, we have each request originate directly from the server where the client would have made the request. Increasing the number of clients can then instead be modelled by making a corresponding increase to the total request rate *R*.

4.1.3 Experiment Details

Experiments are run in two phases. The first, the warmup phase, lasts for five simulated seconds and linearly ramps up the request rate over this period from zero to the desired value of *R* by the end of the warmup period. This phase is intended to inject enough work into the system to bring it closer to a steady state than it would be in if we started taking measurements with no pending traffic.

During the second, the data collection phase, we begin recording data and run the actual experiment to determine the lookup request latency for the specified set of parameters. The second phase runs for at least 30 seconds of simulated time. After this, several heuristic termination criteria are applied with the goal of running the experiment long enough to determine its steady-state behavior.

We used the following termination criteria, which were determined experimentally from numerous simulation trial runs: If the average lookup latency is above one second and rising, then the run is declared "bad". If the average latency is below 800ms and falling, then it is declared "good". If the averages over a simulated ten second period have remained within 10% of one another, we assume the run has reached a steady state and the run is declared "good" or "bad" by comparing its average to the 800ms threshold. Finally, if the overall average drops below 50ms, the run is declared "good". If none of these criteria apply, the run is continued by simulating another 50,000 requests and then reevaluating the termination criteria.

4.1.4 Example Experiment

This section explains an example experiment to give the reader a feel for the kinds of data to be presented in the remainder of the section.

Consider an experiment with values *Stub-BW*=T1, *D*=1, *N*=100, *S*=1ms, *CPU-Rep*=False, *Latency-Rep*=False, *Updates*=False. This models traffic going to a single name service directory that is hosted on a single node among the 100 nodes in the overlay network, at a rate *R*, which we will vary. Replicas of the directory will not be made on other nodes because neither kind of replication is enabled. Traffic consists solely of lookup requests—no updates are being sent. Finally, the network being simulated uses T1 lines for Stub-Stub (ISP network) connections.

We simulated these conditions for varying values of *R*, looking for the point at which too large a value of *R* causes the average request latency to cross the threshold of 800ms. Table 2 shows results for a series of experiments with these parameters.

R **(Requests/s)**	**Requests Simulated**	**Max Latency (ms)**	**Avg. Latency (ms)**	**Latency Std. Dev. (ms)**	**Status**
500	15,995	1,072	646	234	Good
1,000	19,140	13,988	7,164	3,842	Bad
667	28,538	6,106	3,184	1,537	Bad
571	31,932	1,167	668	236	Good
615	30,876	2,753	1,514	561	Bad

Table 2: Results for *Stub-BW*=T1, *D*=1, *N*=100, *CPU-Rep*=False, *Latency-Rep*=False, *Updates*=False

In this series, when the system is not overloaded (the *R*=500 and 571 "Good" runs), the average request latency is under 700ms and the maximum latency is a bit over one second. These times are due to the costs of transmitting requests and replies through the overlay network, the physical network underlying it, and invoking the name service application code on each overlay network forwarding hop node. Conversely, when the network becomes overloaded (the "Bad" runs) the average and maximum latency values shoot up due to network congestion.

In this case, the inflection point between "good" and "bad" runs, representing roughly the maximum effective request rate that can be supported by the system, lies between 571 and 615 total requests per second. Such sets of bounds on the maximum supportable request rate are the subject of all the graphed data in subsequent sections.

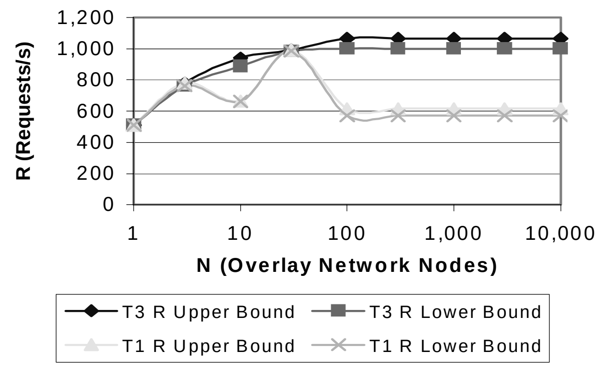

Figure 1: Performance of a Single Directory.
D=1, S=1ms, no replication or updates, Stub-BW={T1,T3}.

4.2 Single Directory Calibration

Figure 1 shows the results of experiments to find the load at which a single unreplicated directory saturates. We placed a directory on a randomly chosen LAN node and accessed it from varying numbers of servers, searching for the request rate at which the lookup latency crossed the 800ms threshold. The results were different for T1 and T3 stub-stub links.

For T3 links, the directory could handle almost 1000 requests per second. At this point the server becomes CPU-limited, as each request takes 1ms of CPU time.

The T1 case is more interesting. For large numbers of servers network effects limit the directory to handling about 600 requests per second. The 30 server point, where the directory could handle nearly 1000 requests per second, demonstrates that the T1 link connecting the server to the WAN is not necessarily the limiting factor, since at least for this particular placement of servers, it had sufficient bandwidth to allow the computation to become CPU-limited. But what the oscillations for small numbers of servers do illustrate is that for small systems, unfortunate placements of links or bandwidth settings will be visible, as each link will represent a significant part of the total overlay network. For large networks, we expect to see smoother data since the law of large numbers will come into play.

4.3 Scalability Calibration

To examine basic scalability, we ran tests in which *N* directories were placed randomly among the *N* servers of the system. Clients' requests were directed randomly among the directories, resulting in a uniform load against all directories, coming from all server machines in their capacity as proxies for client requests. Directories were not replicated. These results are illustrated in Figure 2 and summarized below:[2]

- For T1 stub network links the aggregate network capacity is a noticeable limiting factor. Adding additional servers to the system does not appreciably increase capacity beyond about 20,000 requests per second.
- Employing T3 stub network links increases system capacity significantly. In this case, aggregate network capacity limits system performance to a maximum of about 300,000 requests per second.
- Employing 100Mbits/s speeds for both Transit-Stub and Stub-Stub links increases system capacity even more. The upper bound seems to be around 600,000 requests per second. Increasing link speeds to 1Gbits/s causes network capacity to no longer be the primary bottleneck, at least for system sizes of 10,000 or less.
- Halving the CPU service time yields about twice the performance in all cases except when the aggregate network capacity is reached. Individual server capacity is extra important because completing a single lookup request involves application-level messages being queued up and then processed at multiple servers. Thus, the aggregate number of lookups that can be handled is only a fraction of the number one might expect from multiplying the number of machines by their raw service request capacity.

[2] 10,000 servers was the maximum sized system we could simulate due to computer memory limitations.

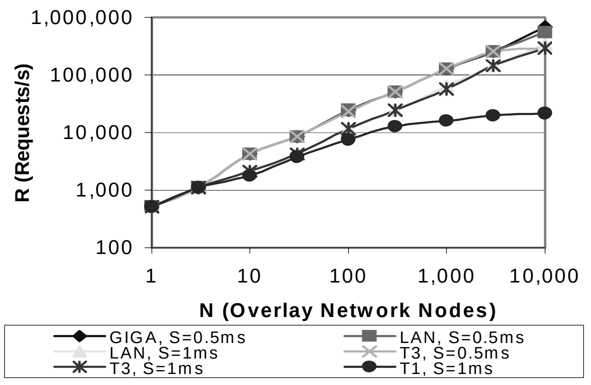

Figure 2: Scale Out Without Replication.
D=N, no replication or updates. Only lower bound curves are shown; upper bound curves are the same at this scale.
Stub-BW = {T1, T3, LAN, GIGA}. S = {0.5ms, 1ms}.

- Employing at least 100Mbits/s links everywhere and a CPU service time of 0.5 milliseconds resulted in a system that could process about 130,000 requests per second using 1000 servers and about 560,000 requests per second using 10,000 servers. Increasing the links speeds to 1Gbits/s does not significantly improve the performance for a system of 1000 servers, but it does improve the capacity of a 10,000 server system to about 690,000 requests per second.

4.4 Flash Crowd Tests

To test how our CPU-load-based replication scheme responds to flash crowds, we ran experiments in which all client traffic was directed at a single directory. When either CPU-load-based or latency-based replication is enabled, or both, the system is able to dynamically respond to hotspot overloads by creating additional replicas. If demand continues to increase, then eventually every server machine in the system will obtain a local replica of the hotspot directories. In the limit (assuming no updates to the hotspot directories), the aggregate hotspot load capacity of the system will equal the sum of the CPU load capacities of all the server machines—assuming that the Internet capacity is sufficient to allow clients to still reach the server machines to make requests of them.

Indeed, in the experiments with replication enabled but no update traffic present, the lookup latencies eventually dropped to just over 1ms—the local limit imposed by the CPU. This is unsurprising because without updates, no network traffic is needed to maintain the effectively read-only state.

4.5 Update Propagation Results

Finally, we ran experiments to determine the scalability of Overlook when the workload includes update traffic—meaning that the replicas are no longer read-only. These were like the flash crowd tests above, except that 1% of the requests update the value associated with a name instead of reading it.

Figure 3 shows the results of these tests for T1 stub links, combined with the T1 scaling data with no replication, as taken from Figure 1 for comparison purposes. The system capacity scales nearly linearly until 1000 servers are reached, at which point the system can service a load of between a half

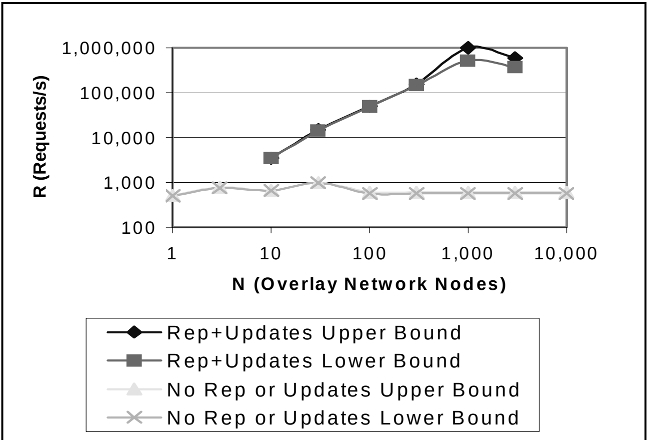

Figure 3: Non-replicated versus replicated scaling limits with 1% update traffic. D=1, S=1ms, Stub-BW = T1, with and without replication.

million and a million total requests per second. For larger numbers of servers, the serviceable load begins to drop.

This behavior can be explained as follows: Under sustained high load each server eventually hosts a replica of the directory being queried and updated. Consequently, each update will need to be propagated to every server node in the system, implying a quadratic dependency on system size for update costs. For system sizes up to about 1000 nodes the overhead of updates is overshadowed by the fact that all lookup requests are effectively handled locally. Beyond replication factors of about 1000 the overhead of updates begins to noticeably dominate the overall responsiveness of the system. Thus, an important control that must be exerted by directory root nodes is to monitor the number of replicas existing for a directory and manage the rate of batched updates and heartbeat messages sent out accordingly.

Whereas the overhead incurred for updates has a quadratic dependency on the number of replicas for a directory, the latencies experienced by updating clients has a mostly logarithmic dependency. Under non-overload conditions, the time required to propagate an update to all the replicas of a directory is determined by the time required to send update messages down and back up the directory's replica tree.

For a system with 10,000 server nodes, the depth of the tree will be at most 4, given the uniform manner in which Pastry routing paths are defined. Thus, update latencies for even the most popular directories are observed to be a few seconds once replication has removed most lookup traffic and assuming that the update (and heartbeat) rate is throttled to avoid overloading the network with update traffic.

However, under circumstances when the entire system is at or near its capacity—for example, in our experiments where we try to find the maximum effective system request rates—the latencies experienced by update clients can be substantially higher. We conclude that directory root nodes must monitor the update latencies they observe and be aggressive about throttling update and heartbeat rates whenever observed latencies exceed their normal-case values.

5. Related Work

5.1 Traditional Name Services

Name services are widely recognized as being one of the key building blocks for distributed applications. They have a substantial history of use in real distributed systems. The Grapevine system [1] was the first replicated distributed name service in regular use over a geographically distributed internet. It was designed to scale to a size of about 30 name server machines.

The Internet Domain Name System (DNS) was introduced in November 1983 [13, 14] to replace host tables and is the primary scalable name service in actual use today. Lampson [12] also proposed a design for a general-purpose global-scale hierarchical name service.

An important factor common to both DNS and Lampson's design is that they both rely on a set of root name servers that must be traversed to lookup hierarchical names. These designs try to mitigate the load on the root servers through caching. [10] quantifies the effectiveness of this caching for today's Internet. In contrast, Overlook explicitly avoids having to traverse root servers and servers for intermediate directory path components when using a name.

Like DNS and Topaz, our design replicates directories onto multiple servers both for availability and load spreading. However, while in these systems the amount of replication needed is determined by a system administrator and changes in the degree of replication are a relatively rare event, in Overlook dynamic replication in response to offered load and/or network congestion is a principal feature.

5.2 Overlay Networks and Applications

There are several active research projects exploring scalable routing via general-purpose overlay networks. These projects include Tapestry [24], Chord [22], Pastry [18], and CAN [15]. While the results presented in this paper were produced using Pastry for routing, we believe that the same techniques should be usable on any of the above overlay routing schemes with similar effectiveness.

Several applications have been proposed on top of general-purpose scalable overlay routing networks. These include the OceanStore storage system [11] on Tapestry, the CFS file system [7] on Chord, the PAST file system [17] on Pastry, and the Scribe event notification system [4, 19] on Pastry. In addition, Freenet [6] is an existing peer-to-peer system that provides file storage functionality similar to that of PAST and Chord, albeit with inexact semantics: file retrieval requests are not guaranteed to find the files they refer to. All of these systems utilize the technique of hashing a content identifier to determine which overlay node should hold the primary copy of the content, just as we do for name service directories. The storage systems also cache content at intermediate forwarding nodes to alleviate hotspots. However, these storage systems are primarily intended to provide archival storage for immutable data; updates to existing data objects is not their primary focus.

Overlook's design differs from these overlay storage applications by supporting directory enumeration and by employing replication instead of caching so that quickly visible updates to stored objects can be achieved. Our work is also the first to address the question of what effect replication has upon the time it takes to make updates globally visible.

5.3 Overlay Multicast

One sub-problem faced by Overlook was the need to propagate name updates to directory replicas. This can be viewed as an instance of the reliable multicast problem applied to directory replicas.

There are several proposals for doing multicast using general-purpose overlay networks. Among them are the CAN multicast design [16] and Bayeux [25]. Other systems providing overlay multicast include Overcast [9], Inktomi [8], End System Multicast [5], and the MBONE [21].

6. Future Work

Much remains to be done to fully understand how a large peer-to-peer system such as Overlook will behave in real life. We list a few of the most important topics needing additional exploration below.

Given the sensitivity that our simulations exhibited to network aspects such as bandwidth, the most important future work to do is to explore how a system such as Overlook behaves under a variety of different network topologies. Part of this should be an exploration of the potential benefits to be gained from modifying a given network topology so that it is more suitable for a system such as Overlook. For example, consider a "mega-service"-style name service, implemented by means of several thousand server machines residing in a single (or a few) data center(s), interconnected primarily by means of high-speed LAN or Gigabit connections. The uniformity and high capacity of the network in such a setting might provide substantially different performance results than the Internet topology that we have so far been exploring.

Another important area to explore is how systems such as Overlook behave under various failure scenarios, as well as under a variety of more heterogeneous workloads than have so far been simulated.

We are also in the process of implementing Overlook on a testbed of several hundred machines in order to understand more detailed aspects of a real system. This will give us the ability to carefully validate various aspects of our simulations.

7. Summary and Conclusions

We have presented the design of Overlook, a scalable name service that supports flash crowds and quickly visible updates. The motivation for this work was management of the event topic name space for an Internet-scale event notification service, but we believe the design to be generally useful.

Our simulation experiments lead us to believe that Overlook can be scaled to support large loads. For example, when run on a network topology that provided at least LAN-level bandwidths on all network links and message processing service times of 0.5 milliseconds, a system of about 10,000 server nodes seems able to handle request loads on the order of 560,000 requests per second. However, the aggregate CPU overhead of processing multiple application-level forwarding hops per lookup request does mean that the incremental contribution of each new server is much less than one might expect.

Our dynamic replication scheme seems able to divert as many servers to the job of fielding requests for a popular directory as is necessary, up to the limit of all servers in the system. Despite this ability to handle flash crowds via dynamic replication, Overlook's design is able to support the updating of directory entries in a manner that makes updates globally visible within seconds in most cases.

Overlook's design is based on a scalable overlay routing network and exploits several interesting features of such a network to achieve its goals. Key among those were the natural support for distributed hash tables, the ability to dynamically, and more importantly, transparently replicate directories without having to effect global state updates, and the automatic "fail-over" behavior of the routing substrate in response to failed nodes.

While the results presented in this paper were produced using the Pastry overlay design, we believe that the same techniques should be usable on other similar overlay routing schemes with similar effectiveness.

Other projects have proposed a variety of different applications that exploit the same features of an overlay network as we have to achieve different, but still similar goals. One way to view our design work is as an extension of prior work that further illustrates the versatility of overlay networks such as Pastry. These networks seem to offer a very versatile "toolkit" of capabilities for building a number of different behaviors into large distributed systems. In our case, we explored how to handle flash crowds while still quickly propagating updates to all replicas of a data object.

Another significant contribution of our work is a first, preliminary exploration of the effects of network bandwidth and CPU load on peer-to-peer systems under high load conditions. Our experimental results imply that peer-to-peer systems may be quite sensitive to both the detailed aspects of the interconnection networks they employ as well as the CPU power of the machines used. In particular, we observed that insufficiently provisioned network bandwidths or CPU resources can cause a system to fail to live up to its scaling potential. Even within a purely server-to-server setting, Overlook was able to scale to truly large sizes only when all network links among our servers were at least LAN-speed links.

In consequence, we chose to avoid a pure peer-to-peer design for our system because of a concern about how low bandwidth network links and intermittently connected, weak client nodes might impede the scalability of our design. We speculate that highly scalable peer-to-peer systems may only be feasible in server-to-server settings, such as "edge of the Internet" data centers, rather than in settings that include client machines sitting behind DSL, or worse yet, 28K modem lines.

This sensitivity to network congestion also forced us to design a replication scheme that would "pull" replicas across congested links as well as the more traditional method of "pushing" new replicas out from overloaded server nodes.

In conclusion, we believe that peer-to-peer overlay networks offer a very promising way of building scalable distributed systems, such as the Overlook name service, that offer a variety of interesting capabilities such as efficient support for flash crowds and quickly visible updates. However, we also speculate that these peer-to-peer systems will end up being successful primarily when deployed as server-to-server systems rather than client-to-client systems because of their sensitivity to the characteristics of the underlying transport networks and host machines on which they rely.

Acknowledgments

The authors wish to thank Antony Rowstron, Miguel Castro, and Anne-Marie Kermarrec for allowing us to use their network simulator and for their ongoing discussions on how to best model the behaviors we report in this paper. We also wish to thank Patricia Jones for her expert editing.

References

[1] Michael D. Schroeder, Andrew D. Birrell, and Roger M. Needham,: Experience with Grapevine: The Growth of a Distributed System. *ACM Transactions on Computer Systems*, vol. 2, no. 1, pp. 3-23, February 1984.

[3] Luis Felipe Cabrera, Michael B. Jones, and Marvin Theimer. Herald: Achieving a Global Event Notification Service. In *Proceedings of the Eighth Workshop on Hot Topics in Operating Systems* (HotOS-VIII), Elmau, Germany. IEEE Computer Society, May 2001.

[4] Miguel Castro, Peter Druschel, Anne-Marie Kermarrec, and Antony Rowstron. *SCRIBE: A large-scale and decentralized publish-subscribe infrastructure*. Submitted for publication, September 2001. http://research.microsoft.com/~antr/scribe/.

[5] Yang-hua Chu, Sanjay G. Rao and Hui Zhang. A Case for End System Multicast. In *Proceedings of ACM SIGMETRICS*, Santa Clara, CA, pp. 1-12, June 2000.

[6] I. Clarke, O. Sandberg, B. Wiley, and T.W. Hong. Freenet: A Distributed Anonymous Information Storage and Retrieval System. In *Designing Privacy Enhancing Technologies: International Workshop on Design Issues in Anonymity and Unobservability*, LNCS 2009, e. by H. Federrath. Spring: New York (2001).

[7] Frank Dabek, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica, Wide-area cooperative storage with CFS. In *Proceedings of 18th ACM Symposium on Operating Systems Principles*, Lake Louise, AB, October 2001.

[8] *The Inktomi Overlay Solution for Streaming Media Broadcasts*. Technical Report, Inktomi, 2000. http://www.inktomi.com/products/media/docs/whtpapr.pdf.

[9] John Jannotti, David K. Gifford, Kirk L. Johnson, M. Frans Kaashoek, and James W. O'Toole, Jr. Overcast: Reliable Multicasting with an Overlay Network. In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation*, San Diego, CA, pp. 197-212. USENIX Association, October 2000.

[10] Jaeyeon Jung, Emil Sit, Hari Balakrishnan, Robert Morris. DNS Performance and the Effectiveness of Caching. In *Proceedings of ACM SIGCOMM Internet Measurement Workshop*, San Francisco, CA, November 2001.

[11] John Kubiatowicz, David Bindel, Yan Chen, Steven Czerwinski, Patrick Eaton, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon, Westley Weimer, Chris Wells, and Ben Zhao. OceanStore: An Architecture for Global-Scale Persistent Storage. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, Cambridge, MA, November 2000.

[12] Butler Lampson. Designing a Global Name Service. In *Proceedings of Fourth ACM Symposium on Principles of Distributed Computing*, Minaki, ON, pp. 1-10, 1986.

[13] P. Mockapetris. *Domain Names - Concepts and Facilities*. RFC 882, November 1983. http://www.ietf.org/rfc/rfc882.txt.

[14] P. Mockapetris. *Domain Names - Implementation and Specification*. RFC 883, November 1983. http://www.ietf.org/rfc/rfc883.txt.

[15] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A Scalable Content-Addressable Network. In *Proceedings of ACM SIGCOMM*, San Diego, CA, pp. 161-172. August 2001.

[16] Sylvia Ratnasamy, Mark Handley, Richard Karp, and Scott Shenker. Application-level Multicast using Content-Addressable Networks. In *Proceedings of Third International Workshop on Networked Group Communication*, UCL, London, UK, November 2001.

[17] Antony Rowstron and Peter Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *Proceedings of 18th ACM Symposium on Operating Systems Principles*, Lake Louise, AB, October 2001.

[18] Antony Rowstron and Peter Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proceedings of IFIP/ACM Middleware 2001*, Heidelberg, Germany, November 2001.

[19] Antony Rowstron, Anne-Marie Kermarrec, Miguel Castro, and Peter Druschel. SCRIBE: The design of a large-scale event notification infrastructure. In *Proceedings of Third International Workshop on Networked Group Communication*, UCL, London, UK, November 2001.

[20] Stefan Saroiu, P. Krishna Gummadi, Steven D. Gribble. A Measurement Study of Peer-to-Peer File Sharing Systems. In *Proceedings of Multimedia Computing and Networking 2002*, San Jose, CA, USA, January 2002.

[21] Kevin Savetz, Neil Randall, and Yves Lepage. *MBONE: Multicasting Tomorrow's Internet*. IDG, 1996. http://www.savetz.com/mbone/.

[22] Ion Stoica, Robert Morris, David Karger, Frans Kaashoek, and Hari Balakrishnan. Chord: A Scalable Peer-To-Peer Lookup Service for Internet Applications. In *Proceedings of ACM SIGCOMM*, San Diego, CA, pp. 149-160. August 2001.

[23] Ellen W. Zegura, Kenneth L. Calvert, and Samrat Bhattacharjee. How to Model an Internetwork. In *Proceedings of IEEE Infocom '96*, San Francisco, CA, April 1996.

[24] Ben Y. Zhao, John D. Kubiatowicz, and Anthony D. Joseph. *Tapestry: An Infrastructure for Fault-tolerant Wide-area Location and Routing*. U. C. Berkeley Technical Report UCB/CSD-01-1141, April, 2001.

[25] Shelley Q. Zhuang, Ben Y. Zhao, Anthony D. Joseph, Randy Katz, and John Kubiatowicz. Bayeux: An Architecture for Scalable and Fault-tolerant Wide-area Data Dissemination. In *Proceedings of Eleventh International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV 2001)*, Port Jefferson, NY, June 2001.