The Case for Pushing DNS

Mark Handley, Adam Greenhalgh
University College London
{*m.handley, a.greenhalgh*}*@cs.ucl.ac.uk*

We present the case for using a peer-to-peer infrastructure to push DNS name server records to thousands of name servers world wide. We show that such an infrastructure increases the robustness of the DNS in an increasingly hostile Internet. We further show that the overheads of a peer-to-peer DNS infrastructure are both manageable and scalable.

1 INTRODUCTION

The Domain Name System (DNS, [5]) has long been a critical part of the Internet infrastructure. The successful Denial-of-Service (DoS) attacks against Microsoft's DNS servers in 2001 and the unsuccessful DoS attacks on the root name servers in 2002 have raised concerns about the vulnerability of the DNS. Operators responded by hardening the infrastructure, and using BGP anycast to replicate the root name servers, so such attacks would need to be larger today to be successful.

Most recent large DoS attacks appear to have been financially motivated, and so the root and top-level name servers have not been a primary target. However, it is hard to predict the motivation of future attackers, so there is still concern that a very large DoS attack on these name servers could cause serious disruption. Thus it is worth investigating alternative ways to harden the DNS infrastructure against attack. This is the goal of our work.

If we consider how DNS functions, it essentially comprises a single hierarchy that performs three functions:

- *Namespace hierarchy*: the name example.com is under com in the hierarchy.
- *Lookup hierarchy*: to query for example.com, we query ".", which gives us the name servers for .com. We then query the .com name servers, which give us the name servers for example.com and so on.
- *Trust hierarchy*: example.com trusts the name servers for .com to return the right answer.

The current DNS infrastructure uses the same hierarchy for all three of these. Most people would agree that we only want to have a single namespace hierarchy, as seen by Internet users, but it is not clear that we also want this to be the same hierarchy as used for lookup or trust. In particular, this places a great deal of trust in the motives and technical expertise of the operators of the root zone and the top level domains.

Our concern though is not primarily trust but robustness of mechanism. The problem we wish to tackle is the technical one of providing a robust DNS lookup infrastructure that is invulnerable to attack, although this is inevitably closely coupled with the issue of trust.

Other solutions recently proposed to this problem use a centralized scheme [2], or distribute the entire DNS via multicast to a large number of replication servers around the Internet [4].

Our solution, in outline, is to use a peer-to-peer infrastructure composed entirely of dedicated DNS servers located at thousands of ISPs throughout the world to distribute signed DNS name server records. This solution is pure *brute-force*. If everyone knows all the NS records, then no DoS attack can cause a *global* DNS failure. This may seem infeasible or uneconomic at first glance. We will show that it is not only technically feasible and extremely robust, but also that the costs are reasonable.

In this paper, we will first sketch out a design (Section 2), then discuss the trust and security issues in more detail (Section 3), and conclude with simulation and measurement data demonstrating that the proposed design is technically feasible and robust to attack (Section 4).

2 DESIGN

In this section we'll discuss the mechanisms we can use to push DNS data to thousands of name servers world wide, using a peer-to-peer infrastructure.

To simplify the explanation, we shall start by assuming that there is a single master DNS organization that has access to all the name server records for the root zone and all the top level domains (.com, .net, .co.uk, etc), and that this organization has published a public key which is known by all the DNS servers in our peer-to-peer infrastructure. Such a keying infrastructure is similar to that used for SSL [3], where all web browsers ship with a built-in set of public key certificates.

In Section 3 we'll revisit these assumptions, and discuss how multiple authorities can be accommodated to further increase robustness.

2.1 Simplest Design

In the simplest DNS push architecture, the master site takes all the DNS name server (NS) records for the root and top level domains, and creates a single file. It then signs this file using its private key, and distributes the signed file to a number of nodes of a peer-to-peer mesh comprised of DNS servers distributed world-wide.

When a peer-to-peer node receives this file, it first uses its in-built public key to check the signature on the file. If the signature is good, then the node caches the file. It uses the data from the file to answer DNS requests, and it also passes on the unmodified signed file to other peers. In such a way, the DNS file is eventually distributed to

all the nodes, and all these nodes can then use this data to answer local DNS requests.

This simple infrastructure has useful properties. No node in the network can corrupt or modify the data because each node will check the signature for itself. Indeed if a node receives bad data from a peer, then with high likelihood it knows that this peer is bad, and it can simply refuse to talk to the peer again. Thus for this application, many of the usual security problems associated with peer-to-peer networks simply do not apply. This is in contrast to multicast approaches[1], which provide robust delivery to all nodes, even if the data itself is bad.

How big is such a DNS file? The only information it makes sense to distribute for each domain are the NS records and their glue records, the SOA record, and the domain name itself. By only pushing NS records we reduce the data involved and also avoid the need to deal directly with DNS load balancing techniques. Our experiments show that a not-terribly-efficient ASCII representation for this data typically comprises less than 100 bytes per domain.

According to Verisign[6], there were 76.9 million domains registered in Q1 2005, including both generic TLDs (.com, etc) and country-code TLDs (.uk, etc). Thus the zone file for all these would comprise 7.5 GBytes. Storing this on disk is no problem, but few current machines could hold it all in main memory. Fortunately this isn't necessary, as the working set that any DNS server needs to cache in memory is much smaller than this.

Each peer node needs to receive this file once. Assuming each node transfers it on to three other nodes, then we could reach 20,000 DNS servers (roughly the number of active Autonomous Systems in the world) in less than 10 generations. If the master DNS site regenerates the master file on a daily basis, and wishes it to reach all servers worldwide in 24 hours, then each site needs to transfer the file on to three sites in 2.4 hours, which requires an outgoing data rate of 21 Mb/s. This is a high data rate for a small site, but not for a large ISP.

To permit efficient DNS lookups, we want to store the data uncompressed on disk, but it makes sense to compress it for transfer. Our measurements show that we can achieve a compression ratio of 3:1 on this data, reducing the outgoing data rate to about 7 Mb/s for the simplest design. This is feasible, although still not negligible.

2.2 Data Chunking

Such a naive design has its limitations. In particular, you must receive a very large file before checking the signature and forwarding. A better design is to split the original data into moderate sized chunks, each labeled with a timestamp[1], chunk number, and the total number of chunks. A reasonable chunk size might be 1 MByte: large enough that the overhead of the signature and metadata is negligible, but small enough to be able to check the signature before too much data has been received.

It is now easier to satisfy the requirement of sub-24-hour total distribution latency, as the first chunk can be re-sent to the next peer while the second chunk is being received. Although the total volume of data sent is unchanged, each node can now spread the transfer across almost the whole 24 hour period. We can reduce the fanout at each node to two, as the extra generations have less effect on overall latency, and so the outgoing compressed bitrate falls to about 470 Kb/s. Such data rates are negligible for almost all ISPs.

2.3 Data Routing and Robustness

To be resistant to DoS, we wish to hide the nodes used to inject data into the peer-to-peer mesh. This imposes constraints on how we can route data. It is important that the peer-to-peer network can still function if some of the peers are compromised and are actively malicious. As a node cannot pass on bad data, the main way to misbehave is to fail to pass on good data, thus acting as a sink for data that would normally be forwarded to other nodes.

Some peerings between nodes may be configured, whereas others will be learned via the peer-to-peer network. DNS servers are generally long-lived entities, so it makes sense for each peer to store a list of every node with which it has interacted. On reboot, a node establishes TCP connections to all of its configured peers and to a random subset of the learned peers. The goal is to establish many more connections than will actually forward the data. A reasonable target might be twenty peers.

When a node receives a chunk of data it checks the signature. If the signature is bad, it discards the chunk, and drops that peering. If the signature is good, the node sends an alert to each peer, indicating that it has this particular chunk. It then sends an offer message to at least one of its configured peers (and perhaps all, if configured to do so), and to at least one randomly chosen learned peer. When these neighbors receive the offer, if they do not have the chunk, they reply with a request message, and the data is then sent to them. If a peer does not receive any requests from the nodes it offered the chunk to, it then offers the chunk to each additional peer in turn. It stops offering when it has sent the data to at least two peers, or when all peers have the data.

If, after a node has forwarded a message to its two chosen peers, it does not receive an alert from another peer, then it can deduce that this peer has been bypassed by the normal flooding process. In such circumstances, another offer message may be sent to the peer missing the data, so as to work around problems elsewhere. As we will show later, this is much more effective than directly

[1]The timestamp may include an element of randomness to prevent the originating source being determined based upon the timestamp.

fanning out to three peers without waiting.

The result is semi-random flooding, where each chunk takes a different path through the peer-to-peer mesh. The randomness makes it impossible for an attacker to locate himself at a critical point in the network, and it makes it very hard to find the nodes originally injecting the data. Unless all its peers already have a chunk, each node fans it out to at least two peers. In almost all cases this is all that is required, but in a few cases the exponential increase fails to reach some nodes. Typically this is because there were few links into a small region, and the nodes feeding these links chose to offer elsewhere. This is solved by sending delayed offers, but under normal circumstances these are rare. So long as the good nodes form a well connected graph, then they will all receive the data. With a node degree of twenty, the probability of the network being disconnected is extremely small, even in the presence of a high fraction of compromised nodes.

2.4 Incremental Update

So far we have assumed that the data is pushed daily, but there is no need to do so if we can incrementally update the existing data. For example, the master site could regenerate the entire file and push this weekly, with deltas to this then being pushed hourly. This reduces the bandwidth needed, and provides a faster change notification mechanism. DNS name server records do not change all that frequently, so the data rate for these updates is low.

When a new node joins the mesh, it receives the signed weekly update chunks from its peers, who have this data stored, and it also receives all of the signed incremental change messages sent since the weekly update. We evaluate the costs of such incremental updates in section 4.2.

2.5 Bootstrap

All peer-to-peer systems need a bootstrap mechanism so peers can find each other. However, unlike most peer-to-peer networks, DNSpush nodes are long lived, so past peers are likely to be available in the future. Thus a DoS attack on the bootstrap mechanism will only affect new nodes with no configured peers. Hence the details of bootstrap are unimportant, and it is feasible to use conventional DNS for this mechanism.

Many nodes will not use the bootstrap mechanism, as they will have configured peers. Indeed NNTP entirely functions in this way. However, the learned peers add randomness to the mesh, which improves large-scale robustness, and give the peer-to-peer mesh small-world properties. For local robustness, using configured peerings between sites that trust each other provides both local robustness and increases the locality of the traffic, reducing the load on the backbone networks. Thus a mixture of both learned and configured peers gives both good small-scale and large-scale properties.

3 TRUST

In Section 2 we considered the case where a single authority signs the DNS data before injection. In this section we will revisit this assumption, but before doing so, we must consider where the data originates.

Every domain under a top-level domain registers two or more name servers. It is these registered name servers that are returned when the TLD server is queried. However, this information is often out-of-date with respect to the NS records given by the domain's own name servers. This can cause robustness and performance issues.

The data we push could be either the registered data or the authoritative data returned by each domain's own name servers. Regular DNS needs the former for the operation of the existing DNS hierarchical mechanisms, but a push mechanism has different constraints, and we believe it is better to push the more up-to-date data reported by each domain itself.

Given that this is not the same as the data given by the registries, the organization that injects the data into the peer-to-peer network does not need to be a registry. All that is needed to run such a site is a list of registered domain names - the data itself can be harvested using the existing DNS mechanisms, which would of course still function to serve sites not using DNSpush.

Returning to the issue of trust, it becomes clear that any site injecting data into the peer-to-peer mesh will need to actively harvest the DNS data. Thus there can be multiple such organizations operating simultaneously, which gives improved robustness. While we know of no case where a master SSL key has been compromised, it is unacceptable to design a mechanism that is fragile to such a single compromise.

With DNSpush, there can be multiple organizations that sign and inject data into the peer-to-peer mesh. Each peer-to-peer server can have multiple public key certificates embedded, one for each originating organization. When multiple organizations all inject their own copies of the data, each server passes on all data that is correctly signed and timestamped, but when it comes to using the data, it takes a majority vote for each domain. In such a way the compromise of a master key would not by itself be catastrophic.

Obviously we also need to design DNSpush in such a way that a signing key can be permanently revoked, and in such a way that a master key need never be kept online. Such issues can be dealt with in the conventional manner, by using the master key to create a certificate for a signing key, which itself is used to sign the data. Thus the signing key can be changed in the event of compromise, by simply flooding a message through the DNSpush mesh containing a key-change message for the compromised signing key, signed by the master key. Even

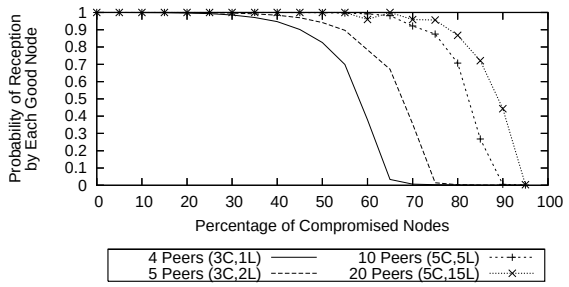

FIGURE 1—Effect of number of peers on probability of receiving a message in the presence of malicious nodes

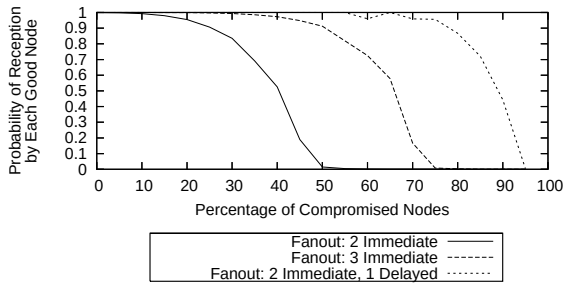

FIGURE 2—Effect of node fanout on probability of receiving a message in the presence of malicious nodes

master keys could be revoked (but not changed) in this way if the need ever arose, but distributing a new master certificate is not trivial. It would be better to predistribute spare master certificates which would not be valid until the previous master certificate for the same organization had been revoked.

The main cost of having multiple signatory authorities is that we need to transmit multiple copies of the data. However, if the authorities are willing to cooperate, the additional costs can be minimized by having some authorities send data that is merely a signed delta from the data distributed by another authority. In this way the additional robustness and decentralization of trust comes at very little additional bandwidth cost.

4 EVALUATION

Most peer-to-peer systems are vulnerable to attack from inside the peer-to-peer network. As we are proposing a distribution infrastructure serving a critical role and comprising tens of thousands of servers, we need to be sure that the our solution is not vulnerable to such attacks.

There is no possibility for a node to pass on bad data, as each node checks the signature and timestamp on the data before forwarding it. Thus the attacks we are concerned about are denial-of-service attacks, where an attacker attempts to prevent the peer-to-peer infrastructure from delivering up-to-date DNS data.

As it is unlikely that tens of thousands of servers can all be maintained perfectly, we must assume that some fraction of them will be compromised at any time.

To answer these questions we built a simple peer-to-peer simulator which, although it does not model network topology or latency issues, does illustrate the basic robustness properties. In our simulation, each peer connects to a number of other peers, which can be either "configured" peers or "learned" peers. Each node is given a random location on a square grid. Each node's configured peers are chosen randomly from the nodes in its local vicinity, with the learned peers chosen randomly from the set of all peers. This allows us to examine issues related to the tradeoff between robustness, hop count latency, and traffic localization.

We simulated varying numbers of malicious peer-to-peer nodes to understand the effect they had on data propagation. A malicious node in these simulations never forwards a message to any neighbor, and only sends an alert to the neighbor that sent it the message. In this way it serves as a data sink, where each malicious node can sink as many messages as it has neighbors. Given that each good node has limited fanout to avoid sending excessive data, the concern is whether the exponential increase in nodes which have received the data can exceed the effect of many data sinks. In these experiments, we initially inject the data to ten randomly chosen nodes, so it is unlikely all ten nodes we choose will be bad.

Figure 1 shows the effect of the number of peers on robustness. In these simulations there are 20,000 nodes, some fraction of which are malicious. On receipt of a message, each node forwards it to one configured peer and one learned peer and, after a short delay, forwards it to one additional peer if any of its peers still needs the message. The results are very good - when each node has 5 configured peers and 15 learned peers ("5C,15L"), 90% of the nodes can be malicious, and on average half the good nodes still receive the message. With fewer peers, the network is less robust, but even with only 5 peers per node, there is no appreciable degradation in robustness until more than 30% of nodes are malicious.

Figure 2 shows the effect of different fanout policies on robustness. As expected, a fanout of 3 is much more robust than a fanout of 2, but delaying the transmission of the third message improves results even further. The reason is fairly obvious - we don't waste the last transmission on nodes that would be reached through the normal exponential increase using the first two transmissions. Thus this last transmission does much more good, as it has a much higher probability of reaching a node that otherwise could not be reached.

These robustness results are almost identical for networks of 1000 and 100,000 nodes, giving confidence that they are relatively scale-invariant. There appears to be little point from a robustness point of view in increasing fanout much beyond three.

If we consider the possibility of a zero-day exploit in

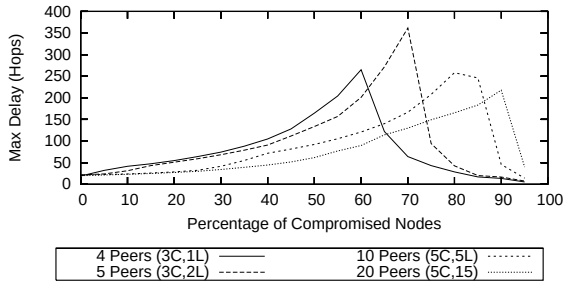

FIGURE 3—Effect of number of peers on maximum delay before receiving a message in the presence of malicious nodes

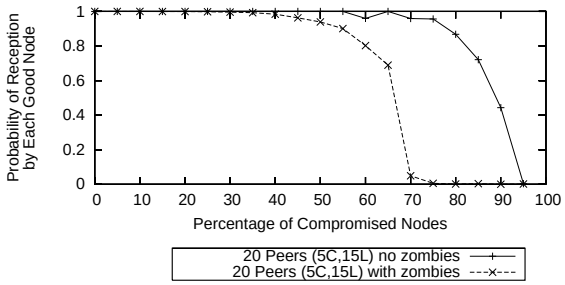

FIGURE 4—Effects of many zombies as learned peers

the peer-to-peer software itself, then these results seem to indicate that two independent software bases would be sufficient to provide a robust peer-to-peer distribution service, even if one software base was compromised. Of course this only holds if no single version of the software comprises more than about 70% of the installed base, and in practice more software diversity would be better.

The presence of malicious nodes also affects latency. Figure 3 shows the delay before the last good node in the mesh receives the message. In these simulations, time is not accurately modeled, so delay is in units of single-hop transmission times. As expected, the paths end up being significantly longer in the presence of malicious nodes, and the delay before each node forwards the third copy of the message also adds to latency. However, as the sender will be pushing more than 7,500 messages of 1 MByte each, even delays of 400 message transmission times are still small relative to the time to transmit all the messages in the first place. The sharp drop off to the right of the peak in each curve indicates that above this point very few nodes receive the data, so it doesn't take long to reach those few.

4.1 Alternative Attacks

The results above are based on a model of a malicious node that simply acts as a data sink. In the 20 peers case, such a malicious node will have to sink ten times the normal traffic load, which may in practice be self-limiting.

Another two avenues for attack are possible:

- Have each malicious p2p node establish learned peerings with as many other nodes as possible.
- Use a large botnet of zombie hosts to set up learned peerings with as many genuine nodes as possible.

In practice, the former will be even more self-limiting than the regular case, but the latter may be feasible. What effect does this have on the peer-to-peer network?

The worst case is a dual-pronged attack, with a significant fraction of the peer-to-peer nodes being compromised, and many zombies also joining the peer-to-peer network to establish learned peerings with the legitimate nodes and act as data sinks.

Figure 4 shows the effects of this attack. The original 20-node curve from figure 1 is shown, and compared against a simulation where zombie nodes comprise every single learned peering from the good nodes. The main thing that prevents this attack being successful is the existence of configured peers, and the nodes' preference for sending to configured peers first. Clearly the attack degrades performance, but it is not successful in disrupting the peer-to-peer network unless a majority of the peer-to-peer nodes are themselves compromised.

The simulation above shows a worst-case scenario. In practice any good implementation would have a preference for peering with nodes that it has known about for a long time, thus it is very unlikely that all of the learned peerings will reach malicious zombie nodes unless those zombies have been actively participating in the peer-to-peer mesh for a very long time.

4.2 Data Churn

In section 2.4 we raised the possibility of incremental updates as a way to reduce the cost of keeping the data up to date. An interesting question is whether each incremental update sent since the entire file was pushed should replace or supplement previous incremental updates.

To better understand how DNS data changes, we monitored 37,000 randomly chosen domains on a daily basis. Figure 5 shows the results. Each day about 0.5% of domains changed name servers, and about 0.1% of domains expired permanently. Extrapolating to the entire DNS, approximately 420,000 domains change and 100,000 domains expire each day. Versign's data indicates that for the last few years the number of registered domains grows by approximately 10 million domains each year, or a current growth rate of about 27,000 domains per day (over and above what is required to replace those that expire). The growth rate from Verisign's data appears to be linear. The churn doesn't appear to be a problem today, and it is unlikely to be a problem in future, as the exponential increase in CPU power and bandwidth out-pace the linear growth in registered domains.

If this data turns out to be a good predictor of future behavior, then compressed incremental updates would need to be about 760 KBytes/hour. If these updates were sent in a cumulative update, they would be about 120

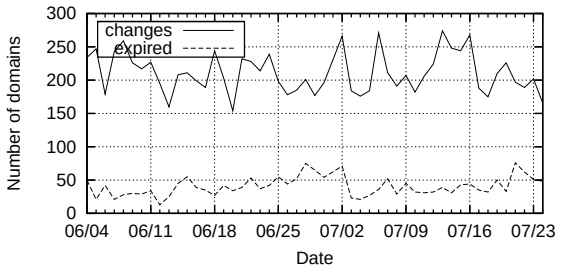

FIGURE 5—Domains changing name servers or returning no data, daily from random sample of 37,000 domains.

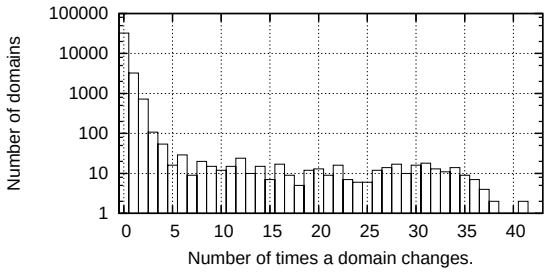

FIGURE 6—Frequency of NS record change amongst the 37,000 randomly sampled domains.

MBytes in size by the end of a week. If we wanted to use hourly updates, then to stream such a *cumulative* update in an hour to three outgoing peers would require about 850 Kb/s, which is probably manageable. Non-cumulative incremental updates would require an average of only 5 Kb/s, but would make it much more critical that every single cumulative update was received. The best solution may be somewhere in between, with multiple levels of incremental updates, similar to the system of dumplevels frequently used for disk backups.

In actual fact, these numbers may over-estimate the actual change rate. Figure 6 shows a histogram of how often a domain changes (in the same 54-day same period measured) against how many domains change that frequently. 87% of the domains don't change at all, and a further 9% change only once. Two-thirds of the changes come from domains that change many times, which is almost certainly dominated by some form of DNS round robin. If rate of change becomes an issue, we can always return the (relatively static) records from the TLD servers for these few domains, or simply accumulate the full set of servers by polling multiple times, rather than treating these as urgent changes.

5 CONCLUSIONS

In this short paper we have discussed the use of a simple peer-to-peer distribution mechanism to push the top levels of the DNS namespace hierarchy to large numbers of name servers world wide. The goal is to provide an extremely robust root to the DNS hierarchy, rendering it much more robust against denial-of-service attacks.

Results from simulation show that the mechanisms are indeed very robust to the attacks we have devised. It is of course possible that we have missed some critical flaw, and so these ideas obviously need more widespread evaluation before we can proceed with actual deployment. Our measurement results show that it is technically and (we believe) economically viable to push this data. As DNS data in the top levels of the hierarchy has been growing linearly for the last few years, it is likely that we have recently passed the point where Moores law now makes such a brute-force approach viable.

Many enhancements to the basic design are possible. For example, message-level FEC might be used to further improve robustness to message loss. As each message takes a different semi-random path through the mesh, this might enable messages that happen to take a bad path to be recovered from those that take a good path.

As we have described it, there is a single distribution channel for a single set of DNS data. However we can easily envisage the peer-to-peer mesh distributing multiple channels giving more specific data. For example, some subset of peer-to-peer nodes at sites that care about the ability to reach Windows Update might wish to subscribe to the microsoft.com DNS channel to receive not only NS records but also A records for Microsoft.

Finally there is some concern about whether the registries will permit distribution of lists of all the domains that have been registered. In practice this does not matter, as if a domain is not in the pushed data then the peer-to-peer DNS server will query it via the normal DNS hierarchy. Once one node discovers that a domain exists, a low rate feedback channel in the peer-to-peer mesh can return this information to the master sites, which can add the domain to the list of domains they monitor. Thus such a peer-to-peer network can be set up by any organization with sufficient network capacity to poll a large number of name servers. In practice though, the updates will be more timely if the registries play an active role in pushing DNS data.

REFERENCES

[1] S. Banerjee, S. Lee, B. Bhattacharjee, and A. Srinivasan. Resilient multicast using overlays. In *Proc. ACM SIGMETRICS*, 2003.

[2] Tim Deegan, Jon Crowcroft, and Andrew Warfi eld. The main name system: An exercise in centralized computing. *ACM SIGCOMM CCR*, 35(5):5–13, October 2005.

[3] Kipp Hickman. The SSL protocol. Technical report, Netscape Communications Corp., February 1995.

[4] Jussi Kangasharju and Keith W. Ross. A replicated architecture for the domain name system. In *Proc. INFOCOM 2000*, pages 660–669, 2000.

[5] P. Mockapetris. Domain names - concepts and facilities. RFC 1034, Internet Engineering Task Force, Nov 1987.

[6] Verisign. The domain name industry brief, May 2005.