Namecast

A Reliable, Flexible, Scalable DNS Hosting System

Andrew Schran

Jennifer Rexford

Michael Freedman

April 15, 2009

1 Introduction

The Domain Name System (DNS) is what translates domain names, like `www.cs.princeton.edu`, to IP addresses, like `128.112.136.35`, on the Internet (see section 2.1). If you are running a service on the Internet, DNS is critical: even if your service is working properly, without functioning DNS nobody will be able to access it, since they will not be able to find it.

DNS hosting services (see section 2.2) allow you to offload the responsbility for storing and serving DNS records to a third party. This similar to how, for example, third party web hosting allows you to have a web site without having to run your own server[1]. Such systems can provide improved reliability and cost savings over hosting your own DNS records due to economies of scale. Existing DNS hosting systems are typically limited in the features they support, or they are huge, expensive, closed services provided mainly to enormous corporate clients the likes of Google or Microsoft. Namecast is an open-source DNS hosting system with a number of unique advantages over many other such systems (see section 3).

Namecast is especially reliable because is designed to run in multiple locations over IP anycast (see section 2.3), so it can easily withstand the failure of one or several individual nodes without service interruption. IP anycast automatically routes packets to the closest working Namecast node and redirects them if it fails, all transparently to the user. Namecast is especially flexible because it supports authoritative DNS hosting of many kinds of records for any domain you own, and allows updates to be done in software instead of only by hand over a web-based interface. Namecast is especially scalable because it uses distributed storage systems like CRAQ (see section 2.4) that are specifically designed to store more data and serve more requests with the addition of new nodes.

As a result of these features, Namecast has low administrative overhead. The automatic failover and recovery mechanisms employed by the watchdog process running on each node (see section 4.2.4) enable Namecast to automatically fix problems in many cases. Namecast's scalability also allows administrators to add additional capacity to the system with minimal effort. Low administrative overhead is crucial, because Namecast is designed to be run as a free service; such a

[1] Indeed, most web hosting services will also handle DNS hosting for your domain if you so desire.

service will not have many people on hand to manually fix any problems that crop up.

Namecast is designed with a number of separate components that together make up one node (see section 4.2). Depending on performance requirements, these components can run on one or more separate physical machines. Each Namecast node communicates with all the others over the Internet in order to coordinate the system as a whole (see section 4.3). General Internet users querying for DNS records hosted by Namecast simply interact with the system as they would any DNS server.

Users of Namecast for DNS hosting interact with the system through an update server (see section 4.1) that speaks a UDP-based protocol called the Namecast Update Protocol (see section 4.4). Security is maintained through the use of Digital Signature Algorithm (DSA) with key-based accounts. Each Namecast DNS hosting account has a DSA public key associated with it, and every update request must be signed with the corresponding private key. This ensures that the request came from the account's owner. (In other words, a DSA public key acts as the "username" for an account, and the signature as the "password.") With key-based accounts, no registration is necessary for anyone to start using Namecast for DNS hosting: as soon as the first update request is received for a particular DSA key, an account for that key is automatically established.

In order to evaluate and optimize the performance of the Namecast system, we performed microbenchmarks on some of its components (see section 5.1) and experimental simulations to test the failover behavior of IP anycast and BGP (see section 5.2). Namecast is a fully functioning system, but there is nonetheless much room for future improvements, some possibilities for which are discussed at the end of this report (see section 6).

2 Background

As an understanding of how the Namecast system works is dependent on knowledge of some underlying technologies, in the following sections we will describe in general terms how these technologies work and their relevance to Namecast.

2.1 DNS

The domain name system (DNS) acts essentially as the phone book of the Internet, converting textual domain names–for example, `www.cs.princeton.edu`–into IP (Internet protocol) addresses–for example, `128.112.136.35`–that can be used to send messages from one host on the Internet to another [7]. This is useful for a number of reasons:

▷ First, it is much easier to remember words than (seemingly random) sequences of numbers. With the help of DNS, you can access a remote system on the Internet by name rather than by address.

▷ Second, DNS makes changing your IP address much easier. If you need to change the IP address of your service, it can be accomplished simply by updating a few DNS entries; users of your service would not need to change their behavior in any way. You may need to move to a new IP address if, for example, you change your web host or ISP.

▷ Third, you can use DNS to map one name to multiple IP addresses. This is useful for load balancing purposes. For example, when a DNS query comes in for `myservice.com`, you might configure your DNS server to respond with the IP address of the least busy server at that moment. Or, you could respond with the IP address of the server geographically closest to the querier. (There are any number of metrics you might use to have DNS direct traffic optimally for your service.)

Reliable DNS service is critical because DNS usually acts as a gateway to all other services offered on the Internet. Even if our web server at `128.112.136.35` is working properly, for example, if you are unable to learn that `www.cs.princeton.edu` can be found at `128.112.136.35`, then you will not be able to access our web site.

The specific details of how DNS works are given in RFC 1035 [13]. In DNS, each domain name has a set of resource records (abbreviated RR) that contain information about that domain. The resource records can have one of several types, and each holds data specific to its type. For example, an `A` (address) record contains an IP address for the domain, `MX` (mail exchange) records

contain the list of mail servers for the domain, and so on. Each resource record also has a TTL (time to live), which indicates how long that record should be cached by the systems that request it. One DNS server (or set of servers) is responsible for being the authoritative server for a domain. This means that it knows about every resource record which pertains to that domain. The authoritative server for a domain identifies itself by serving an SOA (start of authority) record that contains some general information about the domain, such as a default TTL for its records and an email address that can be used as a technical contact for the domain.

In most cases, a DNS query consists of a single UDP packet, and a single UDP packet is sent in response containing all records pertaining to the query. The query contains, among other things, the domain name and record type for which information is sought.[2] For example, if you were trying to access the web site hosted at `www.example.com`, you might execute a DNS query requesting the `A` records for `www.example.com` in order to learn to which IP address you should send your HTTP request. You may not always get back exactly the record types you request. For example, if `www.example.com` were actually an alias for `webserver.example.com`, then instead of receiving an `A` record for `www.example.com` (since such a record does not exist) you would get a `CNAME` record indicating you should restate your query for `webserver.example.com`. If the DNS server wanted to be especially helpful, it would also include all the `A` records for `webserver.example.com` in its reply without requiring you to ask again.

2.2 DNS hosting

Using a DNS hosting service simply means outsourcing the responsibility of answering DNS queries for your domain names to a third party. The benefits of outsourcing DNS are similar to those of outsourcing other IT-type services, such as email or web hosting. It allows you to avoid dealing with the minutiae of setting up and running DNS software and the physical servers and equipment. Further, a cost savings (similar to that of shared web hosting) can be realized by pool-

[2] It is possible to request all of the DNS records for the given domain by setting the query record type to `ANY`. This is not always honored by the remote DNS server, however.

ing the DNS hosting needs of several small-scale users on one server, since none of them alone may require its full capacity. Finally, the larger scale of a DNS hosting service could make setting up a more reliable, redundant system across different locations viable, when it may not be cost effective for a single user to do this.

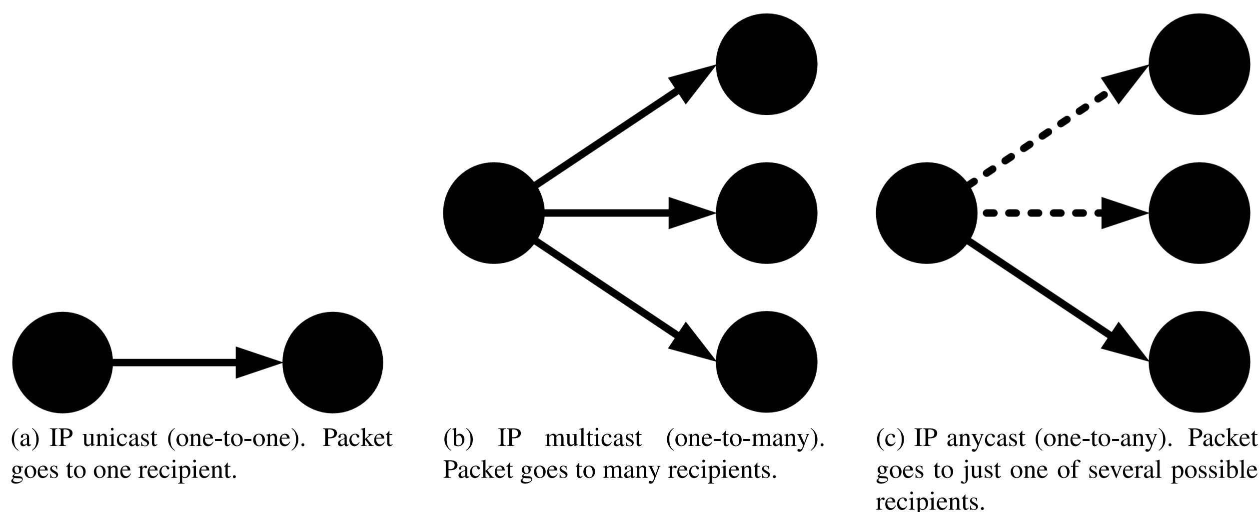
A number of DNS hosting services, both consumer-grade and "business"-grade, already exist. A primary use of the consumer-grade DNS hosting services is to permit easy access to computers on residential Internet connections, most of which provide dynamic (changing) rather than static IP addresses. Hosting a web server on your home computer, for example, becomes difficult when its IP address changes frequently. However, by installing software on the computer to update your DNS host with your new IP address every time it changes, you only need to remember the domain name they gave you to access your web server. No-IP [1] and DynDNS [2] are two examples of such services.

Namecast is designed to provide largely the same kind of service as business-grade DNS hosting services, such as Dynect and Akamai. These services can serve many types of DNS records (in addition to the basic `A` records supported by simpler services) and provide APIs to allow users direct access to their records [3]. They provide automatic load distribution and failover through the use of IP anycast, placing DNS servers in multiple geographic locations (more on IP anycast in section 2.3). Akamai in particular hosts DNS for tremendously large clients, such as Google and Microsoft [4].

2.3 IP anycast

While there is always exactly one sender for an IP packet (or, to be more precise, a single source IP address), its destination can take one of three forms: unicast, multicast, or anycast. A unicast packet is sent to a single recipient with a unique IP address[3]. A response to an incoming ping, for example, would take the form of a unicast packet. Unicast traffic is, therefore, one-to-one: sent

[3] Except where NAT is in use, in which case the packet is sent to the IP address of the NAT device rather than the true destination's unique IP address.



(a) IP unicast (one-to-one). Packet goes to one recipient.

(b) IP multicast (one-to-many). Packet goes to many recipients.

(c) IP anycast (one-to-any). Packet goes to just one of several possible recipients.

Figure 1: Three types of IP packet destinations

from one source to one destination (see Figure 1a).

A multicast packet is sent to multiple recipients. The original packet from the sender is duplicated as necessary by intermediary IP routers, so that the packet arrives at all of its destinations. Multicast traffic is, therefore, one-to-many: sent from one source to many destinations (see Figure 1b). A broadcast packet is a special case of multicast packets, intended for any and all recipients. (That is to say, the the destination of a broadcast packet is “everyone.”) Multicast and broadcast packets are sent to special IP addresses dedicated for this purpose, since there is no provision in IPv4 for having a list of multiple unique destination addresses for one packet [6].

IP anycast packets are, as the name suggests, one-to-any: sent to any (and only) one of multiple possible recipients (see Figure 1c). Again due to the fact that an IPv4 packet can have only one destination address, each of these recipients shares the same IP address, called the “anycast address.” When a packet is sent to the address, the location where the packet actually ends up is determined by the routing decisions made by the IP routers between the packet’s source and its destination. These routing decisions are governed by whatever routing protocol is in use (for the public Internet, this means BGP–the Border Gateway Protocol).

IP anycast is a useful trick because for services that use it, it makes scalability and failover automatic. It is very easy to scale a system that operates over IP anycast by simply adding additional

nodes that announce the same IP address, as long as the backend components of the system (for example, storage) can operate in a distributed manner. If one of the nodes fails for some reason, it can be just as easily removed by withdrawing itself from the list of possible destinations for its anycast address, and the routing protocol in place will automatically figure out where packets should be sent instead. When the new routes are in place after a node is withdrawn, packets sent from a user of the system will find their way to the next nearest node.

IP anycast cannot be used to provide these benefits to just any system, however, due to some limitations of this addressing format. There is no way for the sender or recipient of an anycast packet to know in advance at which receiving location it will arrive, nor can they ensure that different packets from the same source sent to the same anycast address will always end up in the same place. Packets may switch destinations at any time, even in the middle of a stream. This means that to take advantage of the automatic scalability and failover that IP anycast enables, systems running over IP anycast must meet one of two conditions. Either:

- ▷ Protocols have to be designed such that all atomic communications can be contained in a single packet (that is, if a command or request cannot arrive in two pieces at two different spots, then it must fit in one packet); or

- ▷ There must be some kind of backend communication between different nodes of the system so that incoming streams of packets split between two or more locations can be pieced back together.

These requirements mean that TCP and most protocols based on it will not work reliably over IP anycast. DNS is an ideal application for IP anycast since it works over UDP with single-packet queries, and in fact many of the root name servers on the public Internet are replicated through the use of IP anycast [5]. Even though it is sometimes said that there are "thirteen root servers" for DNS because there are thirteen IP addresses at which DNS root servers can be accessed [8], there are in fact many more physical servers than this.

2.4 CRAQ

CRAQ (Chain Replication with Apportioned Queries) is a "distributed object-storage system" developed by Jeff Terrace and Michael Freedman at Princeton University [16]. The Namecast system provides support for the use of CRAQ as its storage layer, where all information about accounts and hosted DNS records is kept. CRAQ is uniquely suited for use in the Namecast system for a number of reasons, discussed in detail in section 3. In short, CRAQ provides a distributed storage model that is resilient and consistent despite a frequently-changing set of available nodes on which to store data. CRAQ's performance also scales well with the addition of more nodes. Details about how CRAQ works are available in Terrace and Freedman's paper, "Object Storage on CRAQ" [16].

3 The Namecast service & system

Namecast was conceived as a project with a dual nature: both a running service which one could use to provide DNS hosting for his or her domains, and an open software system which anyone could use to set up his or her own DNS hosting service. This is analogous to the Emulab network test bed, which is both a service run by the University of Utah which is used by researchers to run experiments on virtual network topologies, and a software system which anyone can download to set up a new Emulab installation of his or her own. Similarly, anyone with the necessary resources (hardware, IP addresses, etc.) could set up a Namecast-compatible DNS hosting service of his or her own.

To set up Namecast as a running service on the public Internet, however, requires the cooperation of upstream ISPs due to its use of IP anycast. In order for Namecast to be accessed at anycast addresses and for a node to be able to withdraw the route to itself when something goes awry, the ISP's router must be willing to speak BGP to the Namecast node. Setting up Namecast as a service is therefore a logistical challenge which we have unfortunately been unable to overcome so far, since doing so requires finding an ISP willing to put in the extra effort of dealing with its

unusual BGP requirements[4]. Things are progressing on this front, however. Our colleague Nick Feamster at Georgia Tech is working with some commercial operators on setting up BGP sessions for use with academic projects like this one; sites are now set up in Seattle, WA; Ashburn, VA; and Atlanta, GA. We expect to be ready to deploy Namecast on the public Internet this summer.

3.1 Why a new DNS hosting service?

Despite the wide availability of DNS hosting for users of all kinds, there are a number of drawbacks to these existing services that we hoped to address in creating Namecast. Free DNS hosting services like DynDNS and No-IP are alright for limited uses, such as to provide a domain name for a web server on your home computer, but they lack flexibility. Usually only a small number of DNS record types are supported, there are limits on the number of records that can be hosted, and most problematically, DNS records must be updated by hand over a web interface. This means that these DNS hosting services often cannot be used for load balancing, hosting DNS for custom domain names, and other more advanced uses. Namecast, however, does provide authoritative DNS hosting for custom domains and provides a command line update utility so that changes to hosted DNS records can be done programmatically rather than by hand.

Commercial DNS hosting services like Dynect and Akamai do provide all these features and probably many more, but they are extremely expensive. The properties of these commercial systems and the way they are set up is also secret; in creating Namecast we hoped to learn in the process more about what the challenges are of setting up a large-scale DNS hosting system. Thinking about how to build an anycast DNS hosting system, for example, uncovered the problem of how to best configure multiple IP anycast addresses for the lowest failover delay, on which our experiments are discussed in section 5.2. It is our hope that other researchers can also find uses for an open source DNS hosting system in their own work, which until now has not existed.

[4] In an attempt to alleviate the difficulty of deploying services that use IP anycast, Hitesh Ballani and Paul Francis at Cornell University proposed a system called the Proxy IP Anycast Service (PIAS) [25, 26] that makes it possible for end-users to set up anycast-based services without requiring each one to get cooperation from ISPs. This system is not yet available for general use at the time of this writing, however.

3.2 Design goals & strategies

The three primary design goals for the Namecast system are reliability, flexibility, and scalability. The first two of these relate two how users interact with the DNS hosting system, while the last mostly relates to how the administrator(s) interact with the system. In the following sections we will discuss the reasons for these goals as well as some of the strategies employed to achieve them.

3.2.1 Reliability

Reliability is the most important property for Namecast to have, since (simply put) if Namecast were not reliable then nobody would use it. The design goal of reliability encompasses two different characteristics of the Namecast system.

▷ First, it should be very unlikely that Namecast will lose any DNS records that are hosted with it.

Namecast makes use of replicated data storage to achieve this. For both the MySQL and CRAQ storage layers, every stored DNS record is located on more than one Namecast node. This means that even if some of the nodes were to be wiped completely, no data would be lost. For the MySQL storage layer, every node holds a complete copy of all DNS records hosted by Namecast, so literally every node would have to be lost in order for any hosted record to be lost. For the CRAQ storage layer, the replication factor (chain length) is configurable, and is by default three. This means that all three of the specific nodes holding a DNS record would have to be lost in order for that record to be lost.

▷ Second, Namecast should have very high availability; even if some Namecast nodes are down the system overall should still be accessible and respond to record updates and queries.

Namecast achieves this through the use of anycast IP addresses for end users to access the system and again through the use of replicated data storage, both of which are necessary for Namecast to work with downed nodes. Anycast IP addresses ensure that users can always find a working Namecast node at the same address, since if a node goes down then the route to it will be withdrawn.

After BGP stabilizes, then, packets originally routed to the failed node are sent to a different, working node (assuming one still exists with that anycast address). Namecast node integration through IP anycast is described in section 4.3.

Replicated storage permits each node to have access to all hosted data, even if some nodes have gone down. For the CRAQ storage layer, data hosted by one machine is automatically replicated on another if the first goes down in order to maintain a constant level of replication. As long as all of the nodes hosting a particular datum do not fail before another can be "recruited," the system as a whole will never lose access to anything. For the MySQL storage layer, each node retains a complete copy of all hosted data, so DNS queries can be answered no matter what happens to the other nodes. However, MySQL uses a master-slave replication setup, where all write requests must be sent to the MySQL master. This means that the master is a single point of failure: if it were to fail, then no DNS record update requests could be handled until it was fixed. For the purposes of reliability, then, CRAQ is a better choice of storage layer in that it has no single point of failure for reads *or* writes.

3.2.2 Flexibility

A flexible DNS hosting system is one that can do anything a user could do if she set up her own DNS server. A number of aspects of the Namecast system help to make it almost as flexible as a self-run DNS server.

▷ Namecast allows the use of a simple command line utility for updating hosted DNS records.

Free DNS hosting services commonly use a web interface to which users must log in in order to update DNS records. Typically this means that all DNS records hosted with these services must be updated by hand (unless the user were to write a complicated script in order to interact with the web interface, which would need to be updated every time the site was changed). On the other hand, Namecast's simple command line client for DNS record updates enables users to manually *or* automatically update their records, since calls to this client can easily be added to custom scripts.

An argument for web-based interfaces could be made on the basis of usability: users uncomfortable with using command line utilities may have an easier time of updating their hosted DNS records with a graphical interface. However, a command line interface seemed more important to write first, since one can do things with it (scripting, for example) that could not be done easily with a web-based interface. A web-based interface or graphical client layered on top of the existing command line update utility, nonetheless, would not be an unwelcome future addition to the Namecast client toolset.

Some free DNS hosting services do provide software utilities to update hosted records in addition to web-based interfaces. However, most of these utilities perform only specific kinds of updates. Some of these utilities will, for example, monitor the external IP address of the machine on which they are running and update the `A` record for the user's domain name in case the IP address changes. Most cannot be used for arbitrary updates, however. Namecast's update client, on the other hand, can be used to make any change at all to hosted DNS records.

▷ Namecast supports most of the commonly used DNS resource record types.

Namecast can presently be used to host `A`, `AAAA`, `CNAME`, `MX`, `NS`, and `TXT` records. It also automatically generates `SOA` records as necessary. For most purposes, these are all the record types that are needed. A DNS query for all records for `princeton.edu`, for example, returns no other record types except for those listed above.

While Namecast supports the most commonly used DNS record types, future support for more kinds of records would increase the flexibility of the service. In particular, with increasing concerns over the security of DNS, support for storing and serving DNSSEC-related records would be a good start.

3.2.3 Scalability

Scalability makes the Namecast system easy to administer for large deployments, since as the need for capacity increases a scalable system can easily grow to meet that demand. Namecast is designed to be easily scalable in three different areas:

▷ Query capacity: the rate at which Namecast can handle DNS queries

▷ Update capacity: the rate at which Namecast can handle updates to hosted DNS records

▷ Storage capacity: the number of total records that can be hosted on Namecast (in terms of space).

It is easy to scale Namecast in all of these dimensions due to the system's use of a distributed storage layer and IP anycast. That these design features also contribute to Namecast's reliability is no coincidence: reliability and scalability both stem from the capability of the Namecast system to withstand nodes' coming in and out at random. IP anycast and distributed storage are chiefly responsible for this capability.

The use of IP anycast makes Namecast scalable because even with nodes appearing and disappearing, users–both users of Namecast DNS hosting and the ones that are sending DNS queries for hosted domains–do not have to keep track of an ever-changing set of unicast IP addresses. With an unchanging anycast IP address, there is only ever one destination to which Namecast queries must be sent, even if the actual node to which these queries are routed can change at any time. The Namecast system itself handles any problems that may arise from these changes in destination nodes, so that the actual node used is transparent to the user. (This is possible largely because all communications with the Namecast system can be conducted solely over UDP.) Further, the number of requests directed to any particular node decreases as more nodes are added to one anycast group, since packets sent to an IP anycast address are routed to only one destination. This helps make query and update capacity scalable, since if each node can only handle requests at a fixed rate, then IP anycast can be counted on to distribute these requests across the available nodes as more are added.

Scaling the Namecast system by adding new nodes works with varying effectiveness depending on the choice of storage layer, MySQL or CRAQ. Because each node for the MySQL storage layer carries a complete copy of all hosted DNS records, the query capacity of a MySQL Namecast service increases linearly with the number of nodes; database reads do not require any interaction

with other nodes. However, for the same reason, the storage capacity does not scale at all with the addition of new nodes: each node must carry every hosted record, so adding a new one does not in any way increase the maximum number of hosted records for the entire system. (To increase storage capacity, you would need to add additional storage to every single node.) Because MySQL uses a master-slave setup for database replication, update capacity also does not scale with the addition of new MySQL nodes: no matter how many nodes there are, all updates (which require writing to the database) must be directed to the single MySQL master.

When using the CRAQ storage layer, on the other hand, all three of these capacities scale up with the addition of new nodes to the system. Data stored with CRAQ have a constant replication factor no matter how many nodes there are, unlike MySQL, where the replication factor is equal to the number of nodes. This means that as more CRAQ nodes are added, each individual node is required to keep track of fewer records than before–CRAQ automatically offloads some existing records to a new node when one comes online. For this reason, query and update capacities scale up with the addition of new CRAQ nodes because with fewer records per node there will be fewer queries or updates directed at any one of them. Storage capacity also scales up with the addition of new CRAQ nodes as long as the replication factor is kept constant, since the more CRAQ nodes there are, the smaller the set of keys each is responsible for.

4 Architecture of Namecast

The Namecast system provides two user-facing services. The first is an update service, which is used to manage DNS records hosted by Namecast. How to establish and maintain DNS hosting with this service is described in section 4.1. The second is DNS service, by which general Internet users obtain the DNS records hosted by Namecast. The use of this service is completely transparent to end users: once the owner of a domain sets up Namecast DNS hosting, DNS will automatically direct them to a Namecast DNS server to get these records. The way the Namecast system is designed on the backend to ensure reliable delivery of these services is described in sections 4.2,

4.3, and 4.4.

4.1 Using Namecast for DNS hosting

Every user of Namecast is identified not by a username and password, but by a DSA public/private key pair he or she holds. This means that no registration is necessary for anyone to start using Namecast. Each user controls all of the domain names under a suffix he or she owns, which by default is `[keyhash].namecast.org` (`keyhash` is the SHA-1 hash of the user's public DSA key) Any time the user adds a record, it will be for a subdomain of this suffix. For example, you could add records for `www.[keyhash].namecast.org` or `mail.[keyhash].namecast.org`. If you have only a small number of subdomains and they will remain relatively fixed, this is very simple to set up as follows. If you own `example.com`, you could add a `CNAME` record on your registrar's web site to point `www.example.com` to `www.[keyhash].namecast.org`, `ftp.example.com` to `ftp.[keyhash].namecast.org`, and so on, and all DNS hosting from then on will be handled by Namecast.

Namecast can also provide authoritative hosting for any domain you own. With authoritative hosting you can easily add and remove subdomains without having to go through your registrar's web site to mess with the `CNAME` records, since DNS queries for an authoritatively hosted domain are sent directly to the Namecast DNS servers rather than redirected to Namecast from the registrar's DNS servers. This also has reliability benefits, since with authoritative hosting you do not need to depend on your registrar's DNS servers to be running in addition to the Namecast DNS servers; the extra layer of indirection is removed.

To prevent just anyone from coming along and seizing control of authoritative Namecast hosting for a domain they do not own, however, first your domain ownership must be validated. Say you own `example.com` and control key `k`. To validate you would add a `CNAME` record from `validate-[hash of k].example.com` to `namecast.org` on your registrar's web site, or wherever else your DNS records are currently being hosted. After this is done, you must send a validation request to Namecast. All of your DNS records hosted with Namecast would then be for

the suffix `example.com` instead of `[hash of k].namecast.org`. So, for example, if you already had DNS records at Namecast pointing `www.[hash of k].namecast.org` and `ftp.[hash of k].namecast.org` to `10.0.0.1`, these records would be changed after validation to point `www.example.com` and `ftp.example.com` to the same IP address.

To manage DSA keys (which are synonymous with user accounts in Namecast) and the corresponding DNS records hosted for each key, a Java client for the Namecast Update Protocol is provided. This client is the main conduit for users of Namecast DNS hosting to interact with the service, and will:

- ▷ Generate new DSA keys for new users of Namecast or those with multiple accounts.
- ▷ Provide the SHA-1 hash of a given DSA key. This is useful for those who are not using Namecast to authoritatively host their own domain name and therefore whose key hash is part of their absolute domain name.
- ▷ Update DNS records hosted by Namecast as specified in command line arguments, querying the update server as necessary for information (e.g. sequence numbers) required to perform the given updates.
- ▷ Update multiple DNS records atomically (either all of the requested changes are made at once, or none are made) through the use of a specially formatted update file.

DNS records hosted by Namecast can therefore be updated automatically according to any criteria the user requires, by incorporating calls to this Java client in other scripts or programs.

4.2 Components of one Namecast node

There are five main components of a single Namecast node. Some of these components are "off-the-shelf," open-source software while others we wrote ourselves. The five components are are the storage layer (MySQL or CRAQ), the DNS server (PowerDNS), the Namecast update server (written by me), the watchdog software (also written by me), and the BGP software (Quagga). Each of these components integrates with one or more of the others as shown in Figure 2.

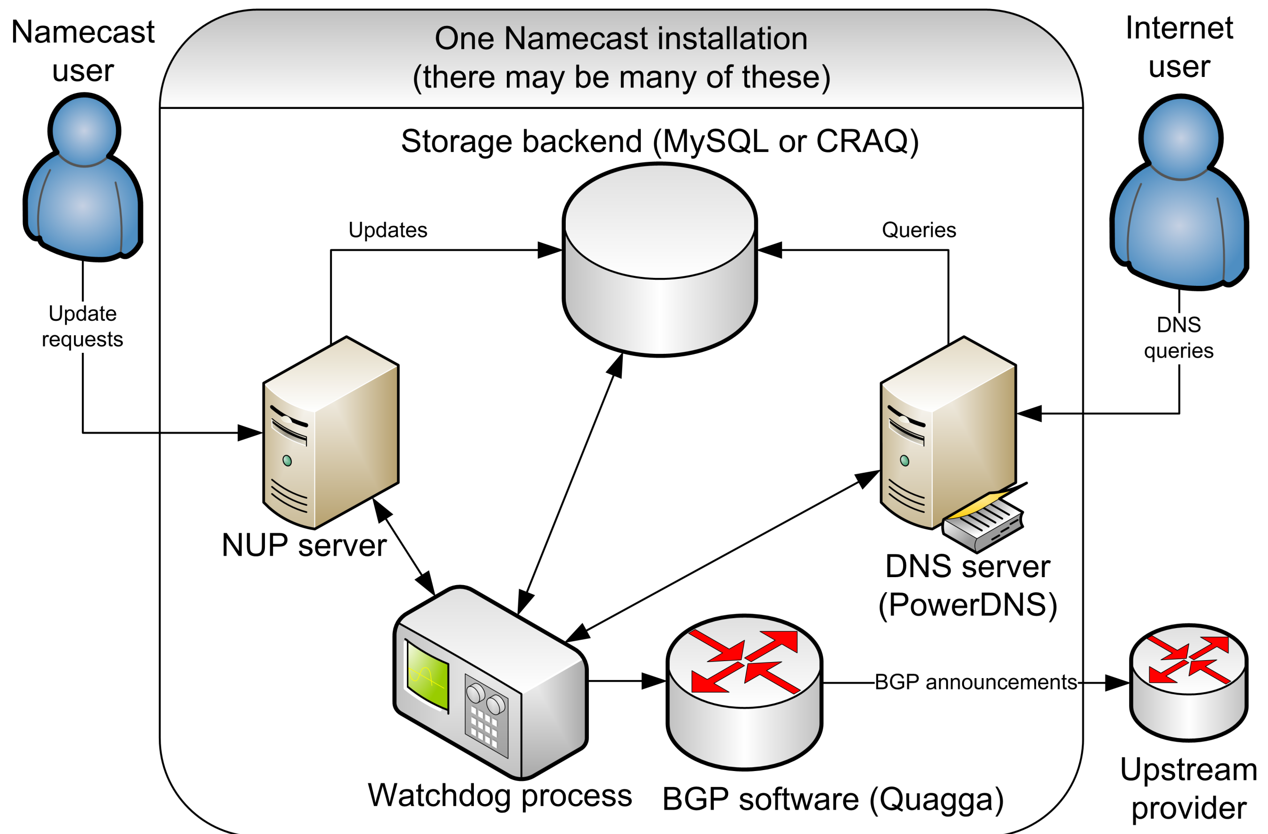


Figure 2: Architecture of one Namecast node

4.2.1 Storage layer

The initial implementation of the storage layer for Namecast used the MySQL database server. The advantage of this choice was MySQL's ubiquity: PowerDNS and Java (in which we wrote the update server code) both have great support for MySQL since so many people use it. However, the way MySQL handles replication is not ideal for the purposes of this system. MySQL requires either a master-slave configuration or a circular multi-master configuration. In the former case, all update requests would have to be sent to the one master location rather than to any Namecast location, negating the benefits of IP anycast for update requests. Further, the master MySQL server would be a single point of failure: if it failed, no more updates could take place across the entire system. In the latter case, if one server in the replication circle were to fail, then the chain of replication would be broken and consistency would be lost. (DNS queries could still be handled by other Namecast nodes even if the master MySQL node failed, however.)

Later on, we also added support to Namecast for a CRAQ storage layer. While the CRAQ code is still experimental, there are a number of significant advantages to the use of CRAQ over MySQL for Namecast. First of all, CRAQ does not use a master-slave configuration to achieve data replication. This means that the single point of failure for writes that exists when using the MySQL storage layer is eliminated for CRAQ: any one CRAQ node can go down and the rest of the system will continue to function for both reads *and* writes. Second, CRAQ has a configurable replication factor; every Namecast node does not need to have a copy of every single hosted DNS record as it would with MySQL replication[5]. This means that total storage space scales in addition to DNS query and update capacity with the addition of new nodes.

CRAQ is nonetheless less efficient than MySQL in some ways. First, all communication with the CRAQ storage layer has to be done over TCP through a client that speaks an ASCII-based protocol. This incurs some overhead for both update server writes and DNS server reads that the MySQL storage layer does not, since it has native drivers for Java and PowerDNS. Second, CRAQ is only a simple key-value store and not a relational database. This means that (at least for our present, naive implementation of the CRAQ-based storage layer) when queries come in for specific DNS record types, the Namecast node has to search linearly through all of the records for the given domain name to find the requested ones rather than executing one simple SQL query. Due to the tradeoffs inherent in either choice of storage layer, we left in support for both. For future development of Namecast, however, we believe the CRAQ storage layer is more promising. The lack of a native CRAQ driver for PowerDNS and Java can be corrected with some additional work, but the deficiencies with MySQL replication cannot. Further, CRAQ is already faster than MySQL for handling DNS queries (see section 5.1.1).

4.2.2 DNS server

The DNS server for Namecast is the open-source PowerDNS [9]. PowerDNS has proven to be a good choice for this purpose for a number of reasons. First, PowerDNS totally separates the front-

[5] If a DNS query arrives at a Namecast node that does not have the necessary records stored locally, then that node will retrieve these records from another one that does have them in order to answer the query.

facing components of the DNS server responsible for processing and responding to DNS queries from the backend storage layer. Unlike the `bind` DNS server, which requires the use of a specially-formatted "zone file" to specify DNS records to be served, PowerDNS can use any number of data sources: MySQL and other databases, `bind`-style zone files, and even arbitrary data stores through the use of a special query protocol over a Unix pipe. This flexibility made it possible to update records on the fly without having to constantly generate new zone files, which would have been prohibitively computationally expensive. It also made it possible to easily integrate CRAQ in addition to MySQL as a supported storage layer for Namecast.

Second, PowerDNS has excellent performance, including built-in caching that works with any storage layer–even arbitrary ones running over the pipe backend. This made it suitable for use in Namecast, which is designed to scale to handle large amounts of traffic. The fact that caching works on arbitrary backends is especially useful for improving performance with the CRAQ storage layer, since it does not have a native PowerDNS backend. (More on PowerDNS performance and caching in section 5.1.1.)

4.2.3 Update server

The update server is responsible for all changes to DNS records for users of the Namecast service. It uses a custom UDP-based protocol called, simply enough, the Namecast Update Protocol (NUP). This server listens at the same anycast IP addresses as the DNS server for update requests from the Namecast update client (or any other client conforming to the protocol specification in section 4.4). This means that updates can be sent to any Namecast node, and can therefore continue even if some of the nodes were to fail as long as the storage layer supports this.

There are two versions of the update server: one for the MySQL storage layer and one for the CRAQ storage layer. The MySQL update server uses the MySQL Connector/J JDBC driver [10] to connect to a local MySQL database server. For valid, authenticated update requests, it adds or changes rows in the `records` table in the database according to the schema required by PowerDNS's MySQL backend (called `gmysql`) [11] by executing the appropriate SQL queries.

The CRAQ update server speaks to a CRAQ client over a TCP socket using an ASCII client protocol, documented on the CRAQ Wiki page [12]. Instead of searching for and updating records using SQL, since CRAQ is a simple key/value storage system and not a relational database, the CRAQ update server stores data in the form of large serialized Java objects each containing all the information for one Namecast account.

The update server uses two security mechanisms to ensure that update requests are only executed if they come from the actual owner of the hosted DNS records in question[6]. First, all update requests must contain a DSA signature that can only be generated by the holder of the private key matching the public key associated with the account. This means that an adversary could not arbitrarily generate a valid update request. Second, all update requests must contain a sequence number that is included in this signature and greater than any sequence number seen before. This means that an adversary could not record update requests as they are transmitted and then later replay them to cause the update server to execute the same request again.

The update server is multithreaded: each incoming update request is immediately passed off to a new handler thread. This means that requests can be processed in parallel, which is especially useful when running the server on a multicore machine. The performance characteristics of the update server are discussed in more detail in section 5.1.2 about the microbenchmarks performed on the server.

4.2.4 Watchdog

The watchdog process is used to monitor all of the other components on its local node–DNS server, update server, and MySQL server. It periodically checks to make sure all of the processes for these components are running. It also sends frequent test queries to the DNS and MySQL servers, and test updates to the update server, to make sure that they are responsive. (Even if the process for a

[6] To be more precise, updates are only executed if the source of the request can prove that it holds the DSA key pair associated with the records to be updated by generating a valid DSA signature for its request. This means that updates will be accepted from anyone who can generate this signature, which may not always be the "owner." It could also be, for example, an authorized agent of the owner, a thief who has stolen his DSA key pair, or someone with a large supercomputer or clever algorithm who has managed to crack the DSA key or algorithm (perhaps the NSA?).

component is running, the test queries are necessary to make sure that it works properly, since not all error conditions will necessarily result in process termination.)

If any of the components terminates or stops responding, the watchdog immediately instructs the BGP software to withdraw the anycast route to its node. (Ideally if the BGP software itself fails, the router on the other end of the BGP session will notice and withdraw the route itself.) It then attempts to restart the component to bring it back and tests again to ensure everything else is still working. To prevent route flapping, wherein the route to a node is rapidly toggled if more than one of its components fails[7], the watchdog reannounces the anycast route only when all components have been verified again as working. If automatic recovery fails, the node stays offline until someone can come around to fix it.

The watchdog itself is simple enough (about a page of Python code) that it is unlikely the watchdog will fail but BGP will continue to work. If the BGP software itself fails, then eventually the router on the other end of the BGP session will notice this when its keepalive message is not returned and will withdraw the anycast route on our node's behalf. However, one still could conceive of a situation in which the watchdog may crash without the anycast route to the node being withdrawn. Reliability could be improved further, then, through the addition of cross-node monitoring, wherein each Namecast node keeps watch over some set of the other nodes and can withdraw the anycast route to a failed node remotely if a problem is found[8].

4.2.5 BGP software

The Quagga open source router [17] is used in Namecast for its implementation of the BGP protocol. This software is responsible for announcing the anycast route(s) for the node on which it is running to the upstream ISP and the Internet at large. As long as Quagga is running, the anycast route to its Namecast node is maintained. If the watchdog process detects any problems with the

[7]More than one component can fail due to dependencies. For example, if the storage layer goes down then the update and DNS servers (which depend on it) may also need to be restarted after the storage layer is repaired in order to get things running again.

[8]Google's senior VP of operations Urs Hölzle notes that Google uses a similar technique to detect outages at their datacenters [23].

node, it terminates Quagga. This results in the removal of the anycast route to the node; as the resulting BGP update propagates through the Internet, routes originally pointing toward this node will be changed to send packets to a working Namecast node (assuming one still exists).

There are certain error conditions (maybe an unexpected link failure, for example) that could potentially cause a Namecast node to fail without an automatic route withdrawal by Quagga. For these cases, BGP has a built-in keepalive timer; the upstream router with which Quagga has a connection periodically pings it, and if no response is received then all routes it has announced are withdrawn. To achieve the lowest possible failover time in this (hopefully) rare circumstance, the lower the keepalive timer can be made the better. This requires the upstream ISP's cooperation in setting the timer to a low value. The watchdog process attempts to have Quagga explicitly withdraw anycast routes to the node in case of trouble to avoid relying on the keepalive timer, which in most cases can be expected to act relatively slowly.

4.3 Node integration with IP anycast

The integration over the Internet of multiple Namecast nodes installed in different geographic locations is illustrated in figure 3. Users–meaning both users of Namecast DNS hosting and general Internet users querying Namecast for DNS records–access Namecast over the Internet solely through its public anycast IP address(es), which do not change with the addition or removal of individual nodes. When a user sends a packet to Namecast (indicated by the solid red arrow in the figure), it will end up at any one of the Namecast nodes with that destination address (indicated by the dotted red arrows).

Namecast nodes coordinate backend components through unique unicast IP addresses of which each node has its own. This way, nodes can keep track of which one is responsible for which stored record (with the CRAQ storage layer) or retrieve database updates from the master node (with the MySQL storage layer). The unique unicast address possesed by each node makes it possible for backend coordination to continue even if, for example, the DNS server on one node fails, which would necessitate withdrawing this node from anycast service to users. It also allows

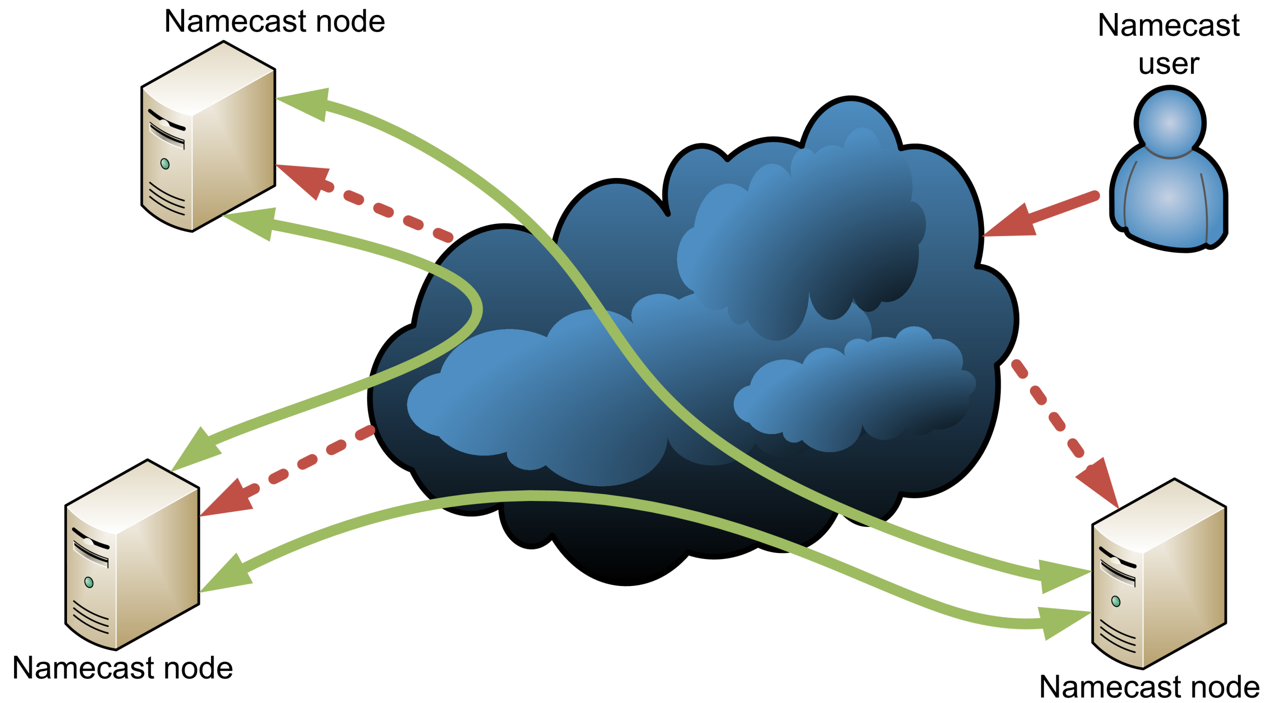


Figure 3: Integration of geographically disparate Namecast nodes over the Internet. Users access the nodes by sending anycast packets which can arrive at any one of the available nodes (red arrows). Nodes communicate between themselves through a set of static unicast addresses (green arrows).

administrators to remotely access and (attempt to) fix a damaged node, even when it is no longer accessible through the public anycast address.

It is worth noting that a Namecast node does not need to consist of only one computer, though this is how they are depicted in figure 3; a node could also consist a group of servers, each running some (or one) of the components of a Namecast node (enumerated in section 4.2). A node could even be an entire datacenter. Because access to the entire group of machines is severed whenever the watchdog detects a problem, it may not be a good idea to set up a datacenter for each node, since every problem would place a lot of otherwise-functioning machines out of service. However, running a small group of machines at each node to increase capacity is certainly a feasible strategy. For example, the DNS server could run on one machine, the update server on a second, the storage layer on a third, and so on.

4.4 Namecast Update Protocol

The Namecast Update Protocol (NUP) is the protocol spoken by the update server in order to make changes to DNS records hosted with Namecast. The protocol is binary-based rather than text-based (as HTTP is, for example), since it is not intended to be human readable. The Java reference implementation of the protocol for both client and server is given in the `UpdateClient` and `UpdateServer` classes written for this project. Two versions of the `UpdateServer` exist, for compatibility with both MySQL and CRAQ storage layers, but the protocol details are unchanged between the two.

4.4.1 Why a custom protocol?

A custom protocol for updating DNS records hosted by Namecast is needed because updates are sent to an anycast IP address, and as such, no existing protocol based on TCP (such as HTTP or SOAP) could be used. Just as anycast allows us to provide geographically distributed DNS service by using route withdrawals to provide failover, it can similarly eliminate the need for a single point of failure with update requests. However, since multiple packets sent to an IP anycast address could be delivered to different servers, connection-oriented protocols could not be expected to work reliably. The Namecast Update Protocol avoids this problem by keeping everything in a single packet. NUP was therefore designed from scratch with a few key goals in mind:

- ▷ **Integrity.** The Namecast update server lives at the same anycast address as the DNS server. This means that to maintain integrity when the destination of an update request may change at any time, all update requests must fit inside a single packet. To achieve this: (a) authentication must be stateless–that is, all information needed to authenticate an update request must be contained in a single packet, and (b) all the changes that a user might wish to perform atomically (either complete the entire request or none of it) must reasonably fit in a single packet.
- ▷ **Extensibility.** It should be possible to change the protocol without breaking existing scripts

and client versions. This is accomplished through the use of a protocol version number that is incremented in case of any changes. (Older clients can request a fallback to previous protocol versions by replying to a request with the appropriate “protocol version unsupported” error message.)

▷ **Security.** The security goals of the Namecast Update Protocol are twofold. First, no one should be able to make arbitrary changes to the Namecast-hosted DNS records for a domain name suffix that he does not own. This is achieved by the use of the Digital Signature Algorithm (DSA). Second, no one should be able to commit a replay attack in which he can resubmit a previously-seen DNS record update. This is achieved by the use of update sequence numbers.

4.4.2 Protocol overview

A Namecast update request works like this: each packet sent by the user, called an “update packet” contains a list of desired changes to her hosted DNS records, called “request elements,” as well as some authentication information. The Namecast Update Server that receives the update packet attempts to execute all of the request elements atomically–if all of them cannot be executed (perhaps due to a syntax error or authentication problem) then none of the requested changes are made. In either case, the server sends back a “reply packet” to either confirm the successful completion of the update or explain why it could not be executed.

The user authenticates update requests by including a public DSA (digital signature algorithm) key at the top of each request and a DSA signature at the bottom that matches the given key. The public DSA key is effectively the “username” for the account; after a specific public key is seen for the first time, no other public key can be used to change or delete records created with it. The effectiveness of this authentication mechanism relies on two properties of DSA: first, that it is practically impossible to generate a signature matching a given DSA public key without the associated private key but easy to verify it; and second, that it is practically impossible to figure out the private key for a given public key. Short of somehow acquiring the original keypair associated

with a set of hosted DNS records (collectively called an "account" on Namecast), no adversary could generate new update requests that would be accepted by the update server.

The technical details of the Namecast Update Protocol, including the packet fields and formats for update requests and replies, are listed in appendix A.

5 Evaluation

In addition to setting up a small-scale installation of the Namecast system on Emulab [14] in order to make sure all of the components work properly together, we performed two experiments to evaluate and optimize the Namecast system. The first was a set of microbenchmarks to determine the rate at which the various components of Namecast can handle users' requests, and the second was a set of tests performed on a simulated network topology to determine how to best allocate Namecast nodes to a set of anycast addresses for the lowest failover time.

5.1 Namecast component microbenchmarks

In order to get a sense of the kind of load a single Namecast node can handle, we ran two microbenchmarks on the Emulab network testbed [14]: one measuring the maximum number of DNS queries per second, and one measuring the maximum number of DNS record updates per second. These benchmarks will also prove useful for future work as a performance baseline against which to measure new optimizations. The benchmarks were performed on the center machine (`n1`) in the network topology illustrated in figure 4. The four querying machines (`u1` through `u4`) simultaneously sent requests to the benchmark machine as fast as possible, maxing out its CPU. (We verified this manually by logging into `n1` and checking CPU usage with the `top` utility.) Four querying machines were necessary to max out the CPU of the benchmark machine due to the asymmetric nature of NUP requests: it is more computationally difficult to generate these requests than it is to process them. The benchmark result was computed as average number of queries answered per second across all of the senders.

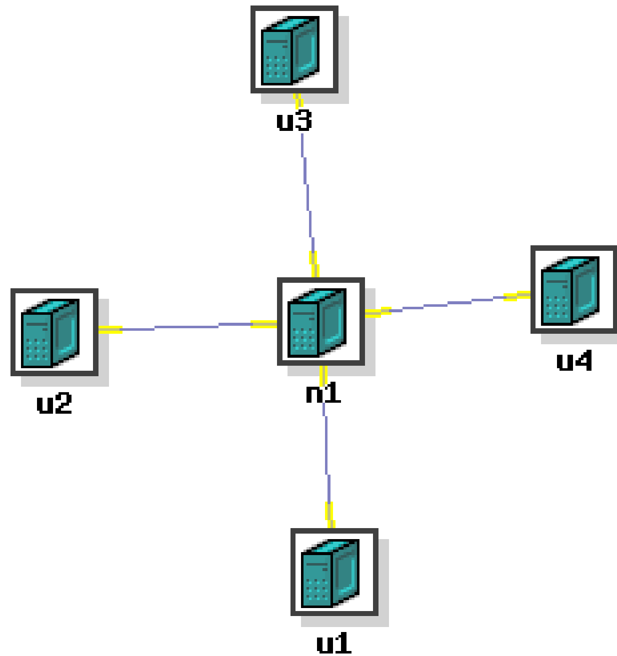
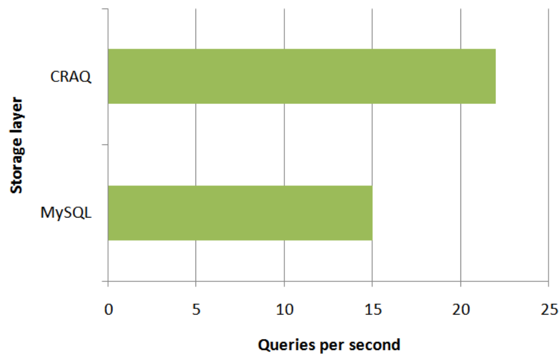


Figure 4: Emulab network topology for Namecast component microbenchmarks

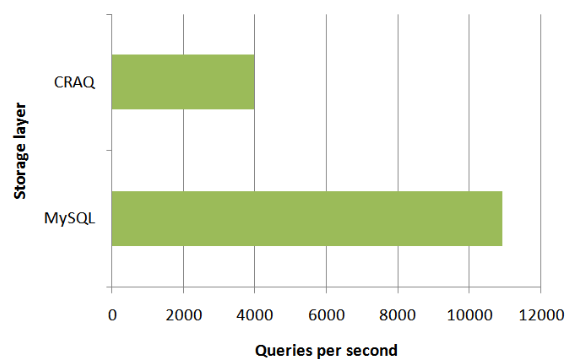
To ensure consistent results across tests, a `pc850` on Emulab [21] was always used as the benchmark machine. The Emulab `pc850` has an 850 MHz Pentium III Coppermine CPU and 512 MB of RAM, so it is significantly underpowered compared to the kind of machine you might expect to actually host Namecast on. This resulted in benchmark numbers that are lower than one might otherwise expect. However, the `pc850`'s low speed made it easier to get consistent benchmark results, since by benchmarking on a slow machine we could ensure that the service was CPU-limited rather than bandwidth-limited.

5.1.1 DNS server

DNS server benchmarks were performed both with caching turned off and with caching turned on at default timeouts. PowerDNS caching works on a number of different levels [19]. First, it caches entire DNS packets that are generated to respond to incoming requests. If an identical



(a) Results with PowerDNS caching disabled



(b) Results with PowerDNS caching enabled with default timeout values

Figure 5: Namecast DNS server microbenchmark results

request is received while the cached packet is still available, it is simply resent without any further processing or the need to query the storage layer. Second, it caches responses from the storage layer for specific domain/type queries. Third, it caches negative responses from the storage layer: if the storage layer informs PowerDNS that a certain domain has no records of a certain type, PowerDNS will not query it again for records of that type for that domain until the cached negative response expires.

To run the benchmark, we used an open-source DNS benchmarking utility called `querysim`, made by the National Institute of Standards and Technology [18]. We configured the benchmark utility to send a rapid stream of DNS queries for a given record to the target DNS server, measuring the average queries per second over a ten minute period. `querysim` used a pool of twenty threads to send requests, waiting for a reply from one request to its corresponding thread before sending another so as to avoid a livelock on the benchmark machine. Nonetheless, the rate at which `querysim` sent DNS queries to the benchmark machine was sufficient to max out its CPU.

With the caching off (figure 5a), every single DNS query received caused a second query to the storage layer for PowerDNS to retrieve any relevant DNS records. PowerDNS then synthesized and sent and appropriate reply packet based on the answers it received from the storage layer. Despite CRAQ's experimental nature, it outperformed MySQL by a fair margin in this benchmark.

Interestingly, with PowerDNS caching turned on (figure 5b), MySQL performed better than

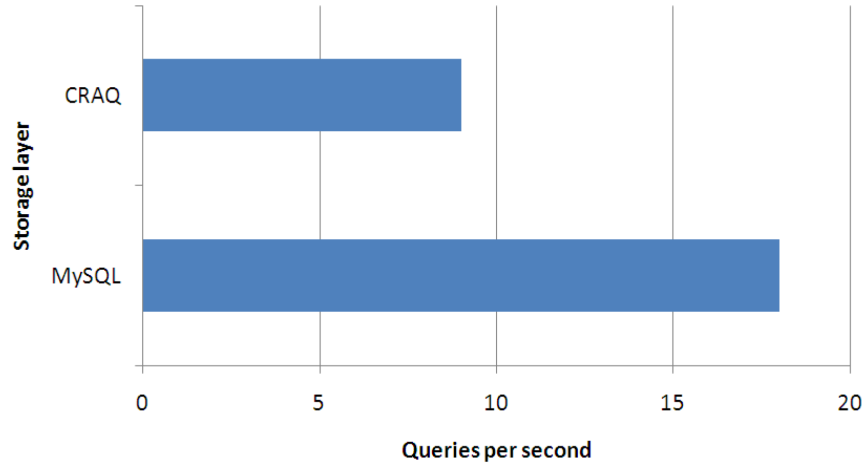


Figure 6: Namecast update server microbenchmark results

CRAQ. While this may seem counterintuitive at first, the result can be explained by the greater latency of CRAQ. While the CRAQ storage layer has higher throughput than the MySQL storage layer (as demonstrated by the non-caching benchmark), it takes CRAQ longer to respond to any one particular request. Since this benchmark tested only how fast PowerDNS could serve the same DNS record repeatedly, every time the cache expires PowerDNS stops responding to requests until the storage layer responds to a query for that record. Since all threads must wait at this point for the response for this particular record, it does not matter that CRAQ can handle more queries per second overall; the waiting time is equal to the response latency for one query and not to the CPU time CRAQ actually uses in serving that request. Although CRAQ was slower than MySQL with caching on, its higher throughput still makes it the better choice for DNS server speed with the varied set of queries you might expect in a regular Namecast installation.

5.1.2 NUP server

To benchmark the update server, we wrote a short program that works similarly to the utility used for DNS benchmarks. Each machine connected to the benchmark machine sent repeated, simultaneous requests to the update server to add records for eight different accounts. (The update server was thus handling requests for thirty-two different accounts across all the machines.) Requests for each account were sent serially, waiting for a reply to one request before sending another in order

to avoid a livelock during the benchmark.

The performance of the CRAQ storage layer for this benchmark was significantly worse than that of the MySQL storage layer (see figure 6)–CRAQ was about half as fast. This is largely due to the way the CRAQ storage layer is integrated with the update server rather and not indicative of performance issues with CRAQ itself. The low update server performance with the CRAQ storage layer is also not a drastic problem, since one would expect DNS queries to make up a much larger portion of the load on a Namecast node than update requests (DNS records are typically read far more often than they are changed). However, there is nonetheless a great deal of room for improvement here. There are a number of explanations for the low performance of the CRAQ storage layer, and these point to ways in which performance here could be increased:

1. CRAQ requires every query to be performed over a TCP socket with an ASCII-based protocol, and the objects stored in the CRAQ storage layer are serialized Java objects. This means that each time a read or write is performed, the update server must serialize or deserialize a Java object, which takes a significant amount of CPU time. Finding ways to organize the stored data without using serialized Java objects may speed things up.

2. The update server currently must read and write some of these Java objects more than once to process an update request; caching these reads and writes would also probably help performance.

3. The version of CRAQ used for these benchmarks requires three chain nodes to be running (the chain length is fixed at three). For the purposes of the benchmark, we ran all three of the nodes on the benchmark machine, which means that for every write performed the data needed to be sent to all three separate CRAQ processes. Running the benchmark with a version of CRAQ supporting a chain length of one or moving the other chain nodes to separate machines (which would be more realistic but harder to measure) could also improve the results here.

5.2 IP anycast failover experiments

Namecast relies on BGP to automatically route around nodes in case they should fail, redirecting packets to those that are still functioning. However, it is not immediately obvious how to best set up these nodes in order to achieve the fastest possible failover. Indeed, in a paper on measurement-based IP anycast deployment, Ballani, Francis, and Ratnasamy note that "IP [a]nycast, if deployed in an ad-hoc manner, does not provide fast failover to clients" [24].

It is possible to have any number of anycast IP addresses, each advertised by a different subset of the total set of Namecast nodes. If there were only one anycast address for every node, then users would be completely reliant on BGP in order to redirect their requests to a working node if one should fail, as they would only have the option to send requests to one destination. Having more than one anycast address could fix this problem: if BGP were slow to route around a failed node at one anycast address, the user (or his or her local DNS server) could force the use of a different, hopefully-functioning node by sending her request to a different address.

However, having too many anycast addresses could also impede failover performance by slowing BGP convergence. With lots of anycast addresses (assuming disjoint sets of nodes for each address), there will only be a few distant nodes to which packets for some address can be routed. The greater distance between these nodes could increase the time BGP would require to reconverge after a node failure.

5.2.1 Experiment design

To determine the best number of anycast addresses in which to group available Namecast nodes, we ran experiments using virtual nodes connected by a virtual network topology (for an example, see figure 7). The "best" number of anycast addresses is here defined to mean that which results in the lowest failover delay[9]. These topologies were automatically and randomly generated, all taking the form of an incomplete binary tree. The nodes running Namecast make up the leaves of the tree,

[9] A paper by Avramopoulos and Suchara [28] also examines the effects of varied anycast group sizes on the security of anycast; the effectiveness of prefix hijacking attacks, where an adversary attempts to steal traffic originally destined for your servers, also varies with anycast group size.

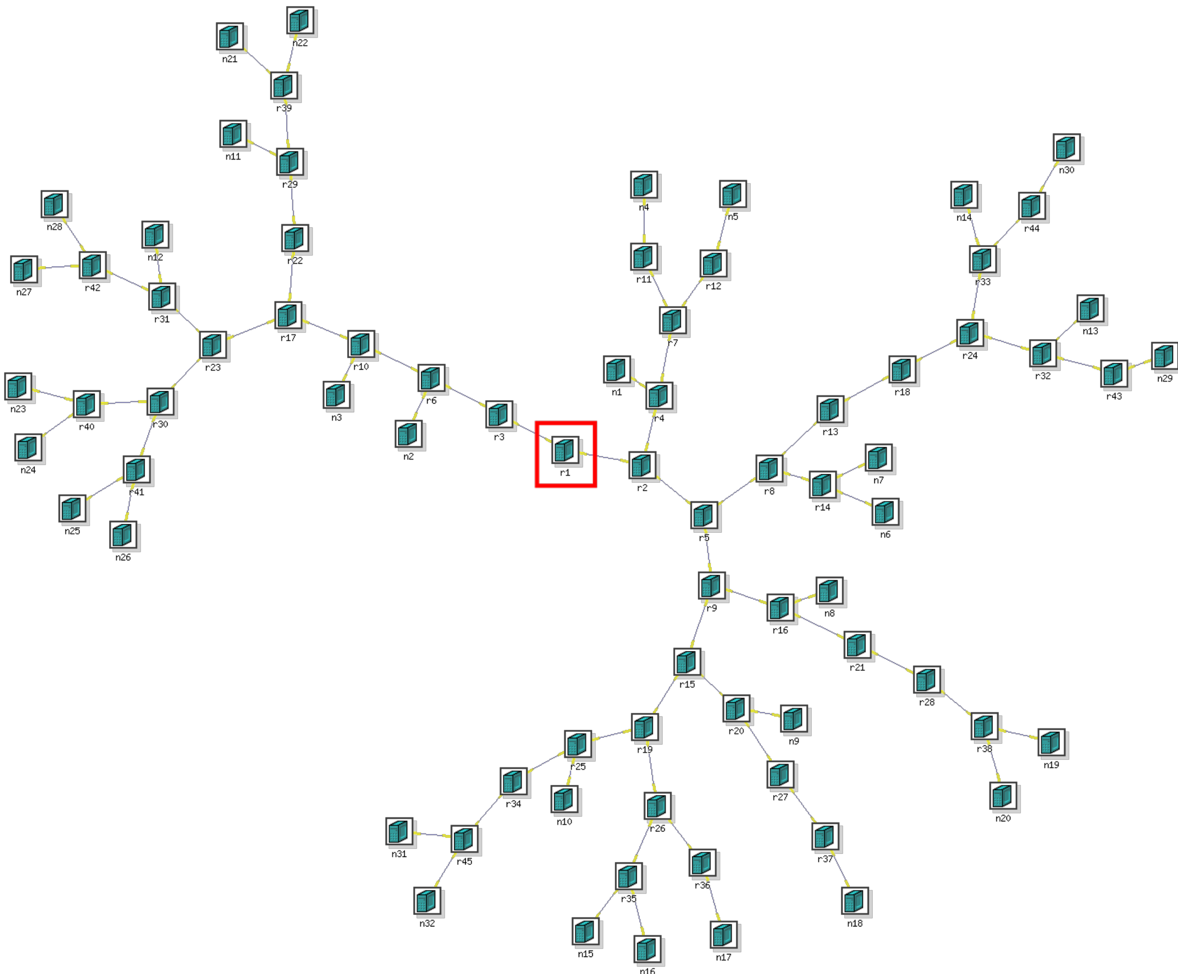


Figure 7: Visualization of a randomly-generated incomplete binary tree test topology. Namecast nodes are at the leaves of the tree. The root node from which requests originate is outlined in red.

and its head is the point of origin from which the failover speed was measured. This topology was chosen because it roughly corresponds to a picture of the Internet from a user's perspective; the possible paths taken by one user's packets (coming from the head of the tree) expand exponentially as the number of hops increases and the path terminates wherever the packet's destination lies (at the leaves of the tree, where the Namecast nodes live). A 50ms delay was added on each virtual link to better imitate latencies seen on the Internet and to lessen the effect of random variations in the results.

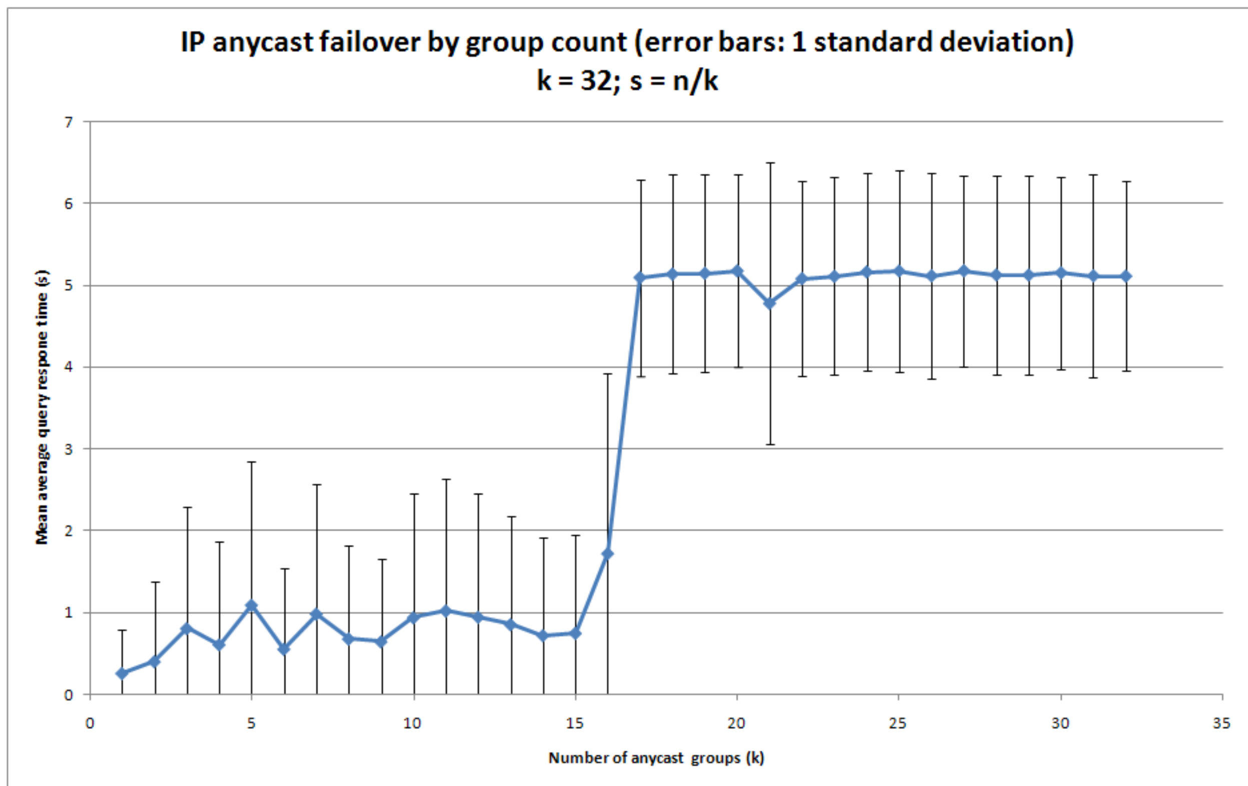
For each experimental topology, the number of servers n was kept at a constant value of 32

while the number of anycast groups k among which the servers were distributed was varied from 1 to 32. For each value of k, the Namecast node to which queries from the head of the tree were delivered for the first anycast group was failed and a DNS query was sent from the head of the tree to this anycast address. (The order in which these two events occurred was varied by ± 1 second in order to ensure that results were not affected by any time-dependent behavior in the BGP convergence process.) If no response to the query was received after five seconds, the next anycast group (or the same anycast group a second time for $k = 1$) was queried instead. The five second failover time was chosen because this matches the behavior of BIND, the most popular DNS server software [20]. The total time it took to receive a response to the DNS query after the first one was sent was measured sixteen times for each value of k on each test topology.

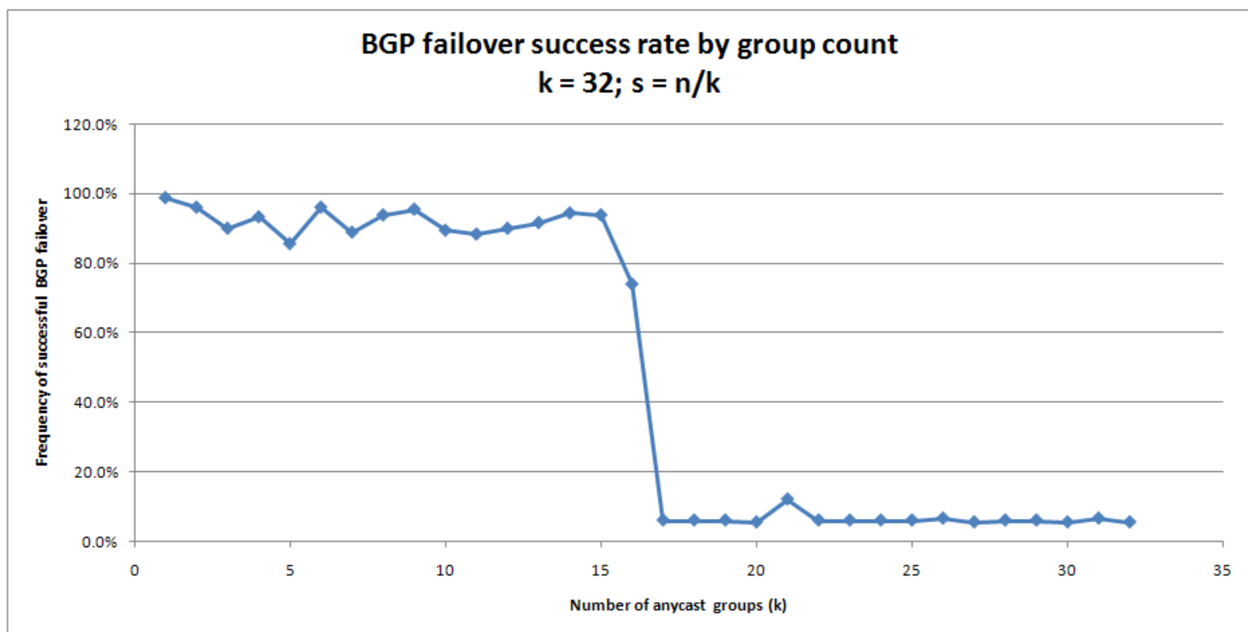
5.2.2 Results

For the collected experimental data we have plotted the average time it takes to receive a response from the time of the initial DNS query (including any time waiting for the retry timer to expire) in figure 8a. The results clearly show that response time trends upward, for two reasons: first, with more anycast groups, there are fewer nodes in any one group. This means that it is more likely that the chosen node for any one anycast address will be farther away from the root of the tree than if there were more nodes to choose from. The drastic jump in average response time around $k = 16$ can be explained by the fact that for values of $k > 16$, the anycast group size drops to one (effectively becoming a unicast address). When there is only one possible destination for a given IP address and that node fails, there is no way for BGP to route around the failed node and so the querier must always wait for the retry timer to expire.

Second, the rate at which BGP failover works fast enough to obviate the need for a retransmission of the initial DNS query decreases as the number of anycast groups increases (as shown in figure 8b). This itself can be explained by the increased distance between nodes for smaller anycast groups; as the distance between nodes drops, BGP messages must travel farther in order for reconvergence to finish and it therefore can take too long to be of any help in avoiding the need



(a) Average time to receive a response for DNS query based on the number of anycast groups among which the available Namecast nodes are divided



(b) Proportion of test runs in which BGP successfully routed around the failed node before the initial request timed out

Figure 8: IP anycast experiment results

for retransmission.

The results of individual tests used to produce figure 8a clustered mostly around two values: one group around 500 ms, and another group around 5,500 ms. This can be explained by the design of the experiment. Either BGP failover works, in which case the failed node's replacement receives and responds to the query in a short time; or it does not, in which case DNS failover triggers after 5 seconds to retransmit the request to a different group. Retransmission to a different group will always work (except where $k = 1$), since our groups are disjoint and we only fail a node in the first group. This wide interval between the two possible values for any one run explains the large error bars in this figure. It is interesting to note that where BGP failover worked, it worked quickly: there was no case in which, for example, BGP eventually routed the first request to a working node but it took more than a second to do so.

The results show that setting $k = 1$ (a single anycast group for all nodes) offers the best performance in the average case because BGP can route around failures faster when there are more nodes in a group. As the number of nodes in one anycast group increases, the average distance between them decreases; therefore, so does the number of routing changes BGP must make to redirect traffic to the nearest working node. When $k = 1$, BGP will be most likely to reconverge in time for the original request to be directed to a functioning node.

Despite the better average case performance where $k = 1$, the use of a single anycast group makes users completely reliant on BGP to handle failed nodes. This means that in the worst case, where BGP may for whatever reason take an unusually long time to route around a failed node or fail to do so entirely, users could potentially experience minutes or even hours of downtime. For this reason, $k = 2$ would be a better choice for the number of anycast groups in the Namecast system. Adding a second, disjoint anycast group to which queries can be sent would seem to provide the best balance between rapid BGP failover in the common case and giving the user an option to failover manually in case something goes wrong with BGP[10].

[10] Bill Woodcock of the Packet Clearing House also suggested $k = 2$ as the optimal number of anycast groups in a talk given August 2002, though Woodcock's suggestion was for two overlapping rather than disjoint groups [27].

6 Future work

While the Namecast system in its present state is fully functional, there are nonetheless plenty of areas in which there exists room for future improvement.

▷ The most important uncompleted piece of Namecast is making it publicly available. This has not yet been done because for Namecast's failover mechanisms to work requires the cooperation of upstream ISPs where the nodes are set up. The difficulties of acquiring this cooperation are discussed in more detail in section 3.

▷ Even if a full-scale publicly available Namecast service cannot be set up in the short term, testing the system with a few "real users" (as opposed to the solely experimental tests run so far) would help to determine which areas of the system are most in need of improvement in addition to uncovering any lingering bugs. This could be considered a precursor to a fully public Namecast service.

▷ The design features of the CRAQ storage layer make it a superior choice to MySQL for use in the Namecast system. However, both due to the experimental nature of the current CRAQ implementation and moreso due to the inefficient implementation of the CRAQ-based update server, performance with the CRAQ storage layer is lacking. This is especially true for handling update requests. A number of modifications to the integration of CRAQ with Namecast could be made in order to increase this performance, such as those discussed in section 5.1.2 on the microbenchmarks for the update server.

▷ A number of improvements to increase the flexibility (see section 3.2.2) of the Namecast system:

- Namecast already supports the most commonly-used types of DNS records. However, there are many more types of DNS records supported by PowerDNS for which there is not yet support in the update server (of which a complete list is available online [22]).

With recently growing concerns over the security of DNS, support for DNSSEC-related records would be especially welcome.[11]

- Namecast supports domain names containing numbers, hyphens, and letters from the standard ASCII character set. Some international users may find support for non-English characters in domain names useful. This should be a relatively simple addition given Java's native support for unicode characters.

- While Namecast can store and serve a static set of DNS records, it cannot do some of the fancier DNS tricks. It would be useful if users of Namecast could, for example, have it direct users to different IP addresses for the same domain name based on where the querier is geographically or which of the web servers for that domain is least busy at the moment. This would be probably be a significant undertaking, though, as it would require a new version of the Namecast Update Protocol as well as changes to the storage layer schema.

▷ Adding support to the Namecast Update Protocol for users to change the public key associated with their account would help improve security, as it would make it possible for users to switch to a new DSA keypair for authentication in case the original is compromised or suspected to be compromised.

▷ Adding cross-node monitoring to withdraw anycast routes to remote nodes in case the watchdog itself fails would improve reliability by lowering failover delay in this extreme case.

▷ The anycast experiments were performed solely for disjoint anycast groups (mathematically, where group size $s = k/n$). It would be worth exploring whether overlapping anycast groups could provide superior failover performance to disjoint groups.

[11] As of this writing, PowerDNS cannot perform DNSSEC processing but can store and serve DNSSEC-related records.

Acknowledgments

Thanks to Jeff Terrace for developing CRAQ, which works so well for Namecast it seems as though it was custom-designed for us, and for assisting us with implementing the CRAQ storage layer. Thanks to Eric Keller for his help with troubleshooting Quagga BGP configuration files (which are surprisingly inscrutable) and setting up our virtual OpenVZ nodes for the anycast failover experiments. Thanks to the Flux Research Group at the University of Utah for maintaining Emulab, which made developing and debugging a system as intricate as Namecast orders of magnitude easier.

References

[1] Vitalwerks Internet Solutions, LLC. 21 Mar 2009 <http://www.no-ip.com>.

[2] DynDNS. Dynamic Network Services, Inc.. 21 Mar 2009 <http://www.dyndns.com>.

[3] "Dynect Platform | For Developers." Dynect. Dynamic Network Services, Inc.. 21 Mar 2009 <http://www.dynect.com/technology/developers.html>.

[4] Cullen, Drew. "Florida 'botmaster' charged with Akamai DDOS attack." 24 Oct 2006. The Register. 21 Mar 2009 <http://www.theregister.co.uk/2006/10/24/akamai_ddos_attack_man_charged>

[5] Hardie, T. "Distributing Authoritative Name Servers via Shared Unicast Addresses." RFC 3258. Apr 2002. Network Working Group. 22 Mar 2009 <http://www.ietf.org/rfc/rfc3258.txt>.

[6] Albanna, Z., Almeroth, K., Meyer, D., and Schipper, M. "IANA Guidelines for IPv4 Multicast Address Assignments." RFC 3171. Aug 2001. Network Working Group. 22 Mar 2009 <http://www.ietf.org/rfc/rfc3171.txt>.

[7] Mockapetris, P. “Domain Names - Concepts and Facilities.” RFC 1034. Nov 1987. Network Working Group. 22 Mar 2009 <http://www.ietf.org/rfc/rfc1034.txt>.

[8] “DNS Root Zone – Hints File.” 12 Dec 2008. Internet Assigned Numbers Authority. 22 Mar 2009 <http://www.internic.net/zones/named.root>.

[9] PowerDNS. PowerDNS BV. 28 Mar 2009 <http://www.powerdns.com/en/documentation.aspx>.

[10] “MySQL Connector/J.” 2008. Sun Microsystems, Inc. 28 Mar 2009. <http://www.mysql.com/products/connector/j/>.

[11] “Basic setup: configuring database connectivity.” PowerDNS manual. 27 Jan 2009. PowerDNS BV. 28 Mar 2009. <http://doc.powerdns.com/configuring-db-connection.html>.

[12] “ASCII Client Protocol.” CRAQ. 26 Mar 2009. SNS Group. 28 Mar 2009. <https://wiki.sns.cs.princeton.edu/index.php/CRAQ#ASCII_Client_Protocol>.

[13] Mockapetris, P. “Domain Names - Implementation and Specification.” RFC 1035. Nov 1987. Network Working Group. 31 Mar 2009. <http://tools.ietf.org/html/rfc1035>.

[14] “Emulab – Network Emulation Testbed.” 31 Mar 2009. Flux Research Group, School of Computing, University of Utah. 31 Mar 2009. <http://emulab.net/>.

[15] “IPv6 Frequently Asked Questions.” 13 Feb 2009. Internet Society. 2 Apr 2009. <http://wiki.chapters.isoc.org/tiki-index.php?page=IPv6+FAQ>.

[16] Terrace, Jeff and Freedman, Michael J. “Object Storage on CRAQ: High-throughput chain replication for read-mostly workloads.” Proc. USENIX Annual Technical Conference. San Diego, CA: June 2009.

[17] “Quagga Software Routing Suite.” 05 Apr 2009. <http://www.quagga.net/>.

[18] "DIY Performance Testing." National Institute of Standards and Technology. 30 Mar 2009. <http://www-x.antd.nist.gov/dnssec/diy_test.html>.

[19] "Performance related settings." PowerDNS manual. 27 Jan 2009. PowerDNS BV. 05 Apr 2009. <http://doc.powerdns.com/performance-settings.html>.

[20] Kumar, A., Postel, J., Neuman, C., Danzig, P., and Miller, S. "Common DNS Implementation Errors and Suggested Fixes." RFC 1536. Oct 1993. Network Working Group. 05 Apr 2009. <http://www.ietf.org/rfc/rfc1536.txt>.

[21] "Emulab.Net – Node Type Information." 05 Apr 2009. Flux Research Group, School of Computing, University of Utah. 05 Apr 2009. <http://www.emulab.net/shownodetype.php3?node_type=pc850>.

[22] "Supported record types and their storage." PowerDNS manual. 27 Jan 2009. PowerDNS BV. 05 Apr 2009. <http://doc.powerdns.com/types.html>.

[23] Hölzle, Urs. Interview with Rich Miller. "How Google Routes Around Outages." 25 Mar 2009. Data Center Knowledge. 09 Apr 2009. <http://www.datacenterknowledge.com/archives/2009/03/25/how-google-routes-around-outages/>.

[24] Ballani, Hitesh, Francis, Paul, and Ratnasamy, Sylvia. "A Measurement-based Deployment Proposal for IP Anycast." ACM SIGCOMM Internet Measurement Conference. IMC 2006, Rio de Janeiro, Brazil: October 2006. <http://www.cs.cornell.edu/~hitesh/pubs/imc06-anymeasure.pdf>.

[25] Ballani, Hitesh, and Francis, Paul. "PIAS: Proxy IP Anycast Service." Department of Computer Science, Cornell University. 11 Apr 2009. <http://pias.gforge.cis.cornell.edu/>.

[26] Ballani, Hitesh, and Francis, Paul. "Towards a Deployable IP Anycast Service." 1st USENIX Workshop on Real, Large Distributed Systems. WORLDS 2004, San Francisco, CA: December 2004. <http://www.cs.cornell.edu/~hitesh/pubs/worlds04-pias.pdf>.

[27] Woodcock, Bill. "Best Practices in IPv4 Anycast Routing." Talk given Aug 2002. Version 0.9. <http://www.pch.net/resources/papers/ipv4-anycast/ipv4-anycast.ppt>.

[28] Avramopoulos, Ioannis and Suchara, Martin. "Protecting DNS from Routing Attacks: A Comparison of Two Alternative Anycast Implementations." IEEE Security & Privacy, Sep/Oct 2009. <http://www.cs.princeton.edu/~msuchara/ProtectingDNS.pdf>.

Appendix

A Namecast Update Protocol specification

Update packet format

Bytes	Contents
8	"`Namecast`" (ASCII string)
2	Version number (16-bit integer)
2	Public key length in bytes (16-bit integer)
Variable (about 450)	X.509-encoded DSA public key
8	Sequence number (64-bit integer)
2	Number of request elements (16-bit integer)
Variable	Request elements
2	DSA signature length in bytes (16-bit integer)
Variable (about 46)	DSA signature (with SHA-1) of all the above

"`Namecast`" string is used to identify this packet as conforming to some version of the Namecast Update Protocol. Every packet must start with this ASCII string verbatim.

Version number is a signed 16-bit integer indicating which version of the Namecast Update Protocol this packet conforms to. As of this writing, this field should only ever be set to zero. For future updates to the Namecast Update Protocol, this version number will change.

Public key length is a signed 16-bit integer indicating the length of the following DSA public key.

DSA public key is an X.509-encoded Digital Signature Algorithm (DSA) public key that identifies the account for which this update should be made. If this public key has never been encountered before, a new account will be made automatically. This key will be used to

verify the DSA signature for the update packet it arrives in. The SHA-1 hash of the key is also the user's account name–all domain names given in request elements for this packet will by default be subdomains of `<DSA public key hash>.namecast.org`.

Sequence number is a signed 64-bit integer containing the global sequence number for this update packet, incremented by one for each update performed for the given public key. The sequence number is only incremented in the case of a successful or partially-successful update (reply packet status codes 0 and 101). If a public key has never been used before, this number starts at zero. In the event of an overflow, the sequence number rolls back to zero. However, there really should never be an overflow. The Namecast update server can be queried for the next expected sequence number by sending an update packet with a sequence number and request element count of zero. The next expected sequence number is also sent with every reply packet. Update packets contain sequence numbers in order to prevent replay attacks in which someone attempts to re-send intercepted, old packets.

Request element count is a 16-bit signed integer that indicates the number of RR (resource record) change requests contained in this packet.

Request elements are contained in sequential order in the packet, formatted as specified below.

Signature length is a 16-bit signed integer that indicates the length of the following DSA signature.

DSA signature is used to authenticate the packet's sender. To generate this value, the entire contents of the packet up to this point should be hashed with the SHA-1 algorithm and the result signed using the private key corresponding to the DSA public key given above. (This corresponds to the "`SHA1withDSA`" algorithm in Java.)

Request element format

Bytes	Contents
2	Opcode (0 for add; 1 for delete; 2 for validate)
2	Domain name size in bytes (16-bit integer)
Variable	Domain name (subdomain for RR, or the domain name to be validated)
2	RR type size in bytes (16-bit integer)
Variable	RR type
4	RRDATA size in bytes (32-bit integer)
Variable	RRDATA
4	TTL (32-bit integer)

Opcode specifies in 8-bit binary format what operation should be performed.

▷ An opcode of 0 (**add**) indicates that the specified RR should be added with the new RRDATA given. This will create a duplicate record if one already exists.

▷ An opcode of 1 (**delete**) indicates that:

- If only a subdomain name is given, all RRs for that subdomain should be deleted. In this case, the RR type field will be a single null byte and the RRDATA size field will be zero.
- If a subdomain name and RR type are given, all RRs of that type for that subdomain should be deleted. In this case, the RRDATA size field will be zero.
- If a subdomain name, RR type, and RRDATA are given, then the RR containing that specific RRDATA should be deleted.

▷ An opcode of 2 (**validate**) indicates that Namecast should become an authoritative DNS server for the all RRs hosted under this key. If validation succeeds, the DNS suffix for all records for this key will be changed to the given one.

- In order to validate a domain you own, you must add a CNAME record pointing from `validate-<key hash>.your.domain.` to `namecast.org`. For example, if your key hashes to `xyz` and you want Namecast to host DNS for `example.com`, you would add a CNAME record from `validate-xyz.example.com` to `namecast.org` on your registrar's web site and then send a validation request for `example.com`.
- The RRTYPE field should be zero length.
- The RRDATA field should contain the technical contact email address for the domain (e.g. `hostmaster@example.com`).
- The TTL field should be set to the desired TTL of the SOA record for your domain.

Domain name size is a 16-bit signed integer indicating the size of the following subdomain name.

Domain name is an ASCII string containing the name of the subdomain for which this RR change should be performed, or the full domain name to be validated. In the former case, it should omit the implied portion of the domain name (that is, the "`.<public key hash>.namecast.org.`" or other validated name).

RR type size is a 16-bit signed integer indicating the size of the following RR type.

RR type is a null-terminated ASCII string containing the type of RR for which this change should be performed. Supported record types and the required data formats are:

▷ `A` – formatted in dotted decimal notation (for example, `10.1.2.3`)

▷ `AAAA` – formatted as an IPv6 address (for example, `2001:db8:85a3::8a2e:370:7334` [15])

▷ `CNAME` – formatted as a standard domain name, with no terminating dot (for example, `www.example.com`)

▷ `MX` – integer indicating priority for the mail server, one space, then a standard domain name with no terminating dot (for example, `10 mail.example.com`).

▷ `NS` – formatted as a standard domain name, with no terminating dot (for example, `ns1.example.com`)

▷ `TXT` – text records are stored as given, and can contain supplemental information about the domain (e.g. contact information) or data of other kinds

▷ `SOA` – generated automatically on domain ownership validation

RRDATA size is a 32-bit signed integer indicating the size of the RRDATA section for this request element.

RRDATA contains the data to be used for this RR change, formatted as indicated by the RR type.

TTL is a 32-bit signed integer indicating the TTL to be used for this RR.

Reply packet format

Bytes	Contents
8	"`Namecast`" (ASCII string)
2	Version number (16-bit integer)
2	DSA public key length in bytes (16-bit integer)
About 450	X.509-encoded DSA public key of the update packet
8	Sequence number of the update packet
8	Next expected sequence number
2	Number of request element codes (16-bit integer)
2	Status code for update packet
Variable	Two-byte status code for each request element

"`Namecast`" string is used to identify this packet as conforming to some version of the Namecast Update Protocol. Every packet must start with this ASCII string verbatim.

Version number is a 16-bit signed integer indicating which version of the Namecast protocol this packet conforms to. As of this writing, these bytes should only ever be set to zero.

DSA key length is a 16-bit signed integer indicating the length of the following DSA public key.

DSA public key copies the public key sent with the corresponding update packet.

Sequence number contains the sequence number of the update packet to which this reply corresponds. Together, the DSA public key and sequence number of the original update packet can be used to match any reply to its corresponding request.

Next sequence number contains the sequence number that should be used in the next update packet received for this public key.

Number of request elements indicates how many request elements were present in the update packet to which this reply corresponds.

Packet status byte can contain one of the following values. All other values are reserved.

Value	Meaning
0	Success
100	Failure
102	Total failure due to invalid RE
103	Failed because request contained no REs
200	Invalid DSA signature
300	Wrong sequence number
400	Unsupported protocol version
401	Protocol version obsolete
500	Malformed update packet
501	Update packet too short
502	Did not find expected number of REs

Element status byte can contain one of the following values. All other values are reserved. There

is one element status byte for each request element in the update packet to which this reply corresponds.

Value	Meaning
0	Success
100	Invalid opcode
200	Invalid subdomain name
201	Subdomain name too long
202	Subdomain name contains invalid characters
300	Invalid RR type
301	RR type unsupported
400	Invalid RRDATA
401	Expected RRDATA, but there was none
500	Invalid TTL
600	Could not process due to other errors
601	Could not process due to previous errors in request packet
602	Could not process due to previous errors in previous request elements