DNS and SMTP for Internet Technology Integrators

Concepts, Tools and Tricks

Prepared by Brent Emerson (Electric Embers Cooperative) for the Tech Underground

31 May 2007

Introduction

In the course of maintaining and troubleshooting different LAN and WAN networks, we get used to thinking of the Domain Name System and email as things about the Internet that Just Work as long as we point our computers at the servers recommended by our ISP—we assume that the right answers will come back from the vague cloud that is DNS, and email will just get to the right place, somehow. And most of the time, this is enough. But understanding exactly which agents are involved in each DNS lookup or the delivery of each email message and how they all work together can be very useful—when you're troubleshooting an email delivery problem, explaining DNS propagation to a client, or need to stand up to a clueless technician at a registrar or web host. With enough understanding of the agents and their interactions, the vague cloud turns into an simple machine with easy-to-understand, logical and discrete parts and pathways.

This guide attempts to explain these parts and pathways to empower tech workers everywhere. The discussion sometimes starts from scratch to build a strong vocabulary and conceptual baseline before moving on to the more interesting tools and tricks. In some places my terminology is a bit unorthodox; this is designed to correct the vagaries of more standard usage when I think more precise distinctions are in order. Some details have been glossed over or simplified when possible to make the explanations shorter without interfering with important concepts or practices. This document can be used as a quick reference, a tutorial, or a guide to group discussion. Enjoy!

DNS

Why we have the Domain Name System

The Internet Protocol (IP) on which the Internet is based provides uniform naming of all the computers connected to it and assures that those computers can all communicate with each other. But IP addresses are 32-bit numbers (128-bit under IPv6), which makes it very hard to remember more than a few. An Internet without human-readable names would be like the telephone system with really long phone numbers, hundreds of people to talk to each day, and no phone book!

One possibility for using names on the Internet is probably the simplest you can imagine: each host or site would maintain an independent hosts file mapping names to IP addresses (like the hosts files found in some operating systems). Each system administrator would make up their own nicknames for other machines with which they need to communicate, ensuring local consistency and making sure no one has to remember IP addresses. This could work very well on a small scale, but would quickly grow unmanageable as the number of hosts to be named increased. Also, names would be inconsistent across the network, and the scheme would be inefficient and redundant, as the same work of naming hosts would be duplicated across many sites. Clearly this method wouldn't work on even a very small Internet.

Another option would be to try to keep one hosts file for the whole network. In the 1970s and early 1980s, the Internet (then ARPAnet) used just such a central hosts.txt file, maintained at Stanford, updated by email and distributed by FTP. With one central registry translating names to addresses, this scheme solved many of the problems of using independent host files. Names were consistent across the network, and the work of maintaining the name database was done just once rather than being duplicated all over the net. But as the number of hosts increased, the work of allocating new names for the whole Internet became too much for one agency to handle. The central hosts.txt file got larger and larger, yet sites had to keep updating their copy more and more frequently to keep it from getting too out of date.

The **Domain Name System (DNS)** was invented to retain the consistency and non-duplication of effort of the hosts.txt system, but to distribute the work of maintaining names to sites all over the Internet. To allow the system to be consistent while being distributed, a hierarchical structure was created, where many different servers are delegated authority over discrete portions of the namespace, called **domains**. These servers can further sub-delegate their portions to other servers, creating a tree of arbitrary depth.

DNS Agents & Pathways: Queries between Resolvers, Helpers and Delegates

The Clients: Resolvers

In DNS, the client/server model doesn't always make perfect sense, because many servers will also operate as clients. However, there are agents that function exclusively as clients: **resolvers**. DNS resolvers are very simple little programs that are built into libraries, applications and operating systems. Their job is to look up DNS information for all the other programs on the system (e.g., web browsers, email programs) by asking a local DNS server. Configuring a DNS resolver is very simple: just point it at a helper server, and it works. This is often as far as we get with DNS.

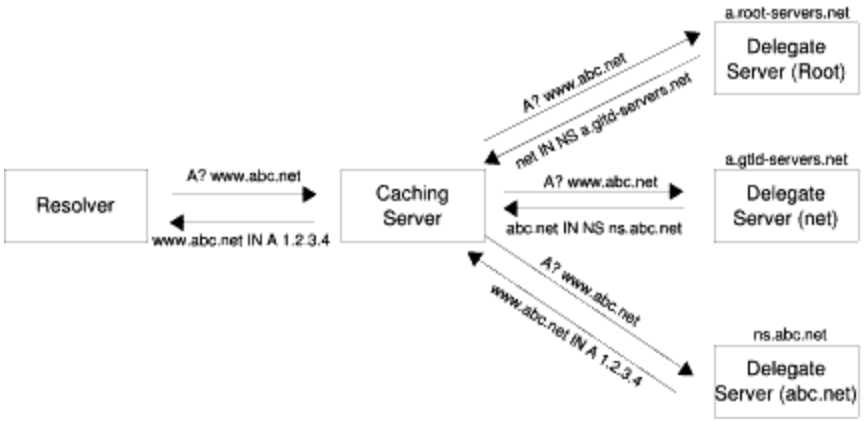
The Servers: Helpers and Delegates

There are two main roles for DNS servers: helper and delegate. A **helper server** is the one you or your ISP runs to know DNS information about the entire Internet, but just for your (or their network's) use. **Delegate servers** are generally operated by hosting companies and have the opposite role: they know only about a particular domain, but they're used by the entire Internet. The pathway of a typical DNS query goes like this: a resolver queries a helper server, which gets the information requested from the correct delegate server and returns it to the resolver.

In this simple pathway you can see the distributed nature of DNS: the resolver only has to know the IP address of one helper server, the helper server just has to know which delegate server to ask, and the delegate just has to know information about its own domain.

So what makes a DNS server a helper or a delegate? For a helper, the answer is simple: a DNS server is a helper just because your ISP told you to use it as your nameserver. This information can come to you in a letter and be configured manually into your resolver by a human, or it can come via DHCP and be auto-configured. A delegate gets its role via a more complicated interaction involving more people. First, somebody registers a domain name (say, abc.net) with a domain **registrar** and becomes the owner of the domain. Once the domain is registered, it's the registrar's job to inform the delegate servers for the next level up (the **superdomain**, .net in our example) of which nameservers should be delegated authority for this domain. Those servers delegate authority for the domain by serving its NS records, and from then on helper servers all over the world know which delegates to ask when they need to know information about the domain. Remember, delegate servers make a hierarchical tree. The top servers are the **root** servers, and they have the simplest and most important job in DNS: serving NS records which delegate authority for all the **top level domains** (TLDs) like .net, .org, and .eu. The servers they delegate then delegate authority for all the **subdomains** of the TLDs: abc.net, oxhouse.org, and france.eu. This process of delegation continues, distributing authority for ever longer (and lower-level) subdomains on the DNS tree.

So those are the DNS server roles, and a brief sketch of how the DNS is structured. But how do the servers actually behave in fulfulling their roles? A DNS server has two possible behaviors: caching/recursing or authoritative/referring. Depending on the query, a caching/recursing server either already knows the answer or will go find it for you:



We call this a **recursed** response because the helper server looked up the answer for you, checking with all the necessary delegate servers (technically, it actually would only be "recursive" if some of the delegate servers had to go on their own hunting expeditions, but that's the word that got picked.) Most likely, though, it already knows the answer and can simply tell you without looking it up. When a caching server looks something up in DNS, it caches the answer it gets back for an an amount of time set by the delegate server—after this **time to live (TTL)** interval has expired, subsequent queries are again checked on the delegate server. This reduces DNS traffic, but gives rise to the phenomenon of DNS propagation: if a change is made on a delegate server, caching servers all over the Internet may wait up to one TTL interval before expiring their old data, re-checking and caching the new record. This means that resolvers which query those caching servers won't find out about the change until up to one TTL after the change was made. Most responses resolvers get are like this—we call them **cached** responses.

Authoritative/referring servers have much simpler behavior[1]. They'll tell you only what they already know (they won't ever do the extra work of returning a recursive response), and they only know what they know because a human manually configured them that way. If they know what you asked, they'll tell you —this is called an **authoritative** response. If not, they'll refer you to the server that does—a **referral**. In the diagram above, the caching server got 3 referrals before finally getting the authoritative response it was looking for.

You may already have noticed that caching/recursing servers would make perfect helper servers, and authoritative/referring servers would be great for the job of delegate. This is how DNS usually works:

Role	Behavior	Responses
Helper	Caching (/Recursing)	Cached
		Recursed
Delegate	Authoritative (/Referring)	Authoritative
		Referral

1 At least, this is how it looks from the outside. From an administrator's perspective, there are a few different types of authoritative servers, because humans usually only directly configure some of them (**master** servers) and configure others (**slave** servers) to automatically mirror the DNS records in the domains they know about. When DNS records are updated on a master server, it sends a special NOTIFY signal to its slaves, telling them that they should check the master server and update their corresponding records (if the master's data is newer) by doing a **zone transfer**. These auto-updates used to be directed from the slave side, and each domain still features domain-specific **refresh** (time between scheduled rechecks of the master), **retry** (time between rechecks after a failed refresh), and **expire** (time after which the slave should delete its data for this domain and stop responding to queries) intervals coded in the SOA records for each domain. The domain's serial number, also found in the SOA record, is used by slaves to compare their version of the domain against the master's—slaves only update if the master has a newer zone file.

But "caching" and "authoritative" are *behaviors*, and "helper" and "delegate" are *roles*—this is worth keeping in mind. Again, "helper" simply means that a resolver is expecting the server to return arbitrary DNS records; "caching" means it actually will. "Authoritative" means the server *thinks* it knows the right answers for a domain, while "delegate" means it has actually been given that authority by a server higher up in the tree. Caching servers can be improperly delegated authority for domains they don't behave authoritatively for; resolvers can mistakenly try using an authoritative server as a helper. When roles and behaviors are confused, miscommunication reigns; when they're mismatched, DNS breaks.

The Pathways: Queries & Responses over TCP/IP

Because DNS queries underlie many of the interactions taking place on the Internet, there are a lot of them flying around. In order for queries to be made and returned quickly and with minimal overhead for resolvers and servers, the protocol was designed to work over UDP (rather than TCP) on port 53. A query is made from a high port on the client to port 53 on the server, and the server responds over UDP from port 53 to the original high port (except when the response is too long to fit into one UDP packet, in which case the query/response pair is repeated over TCP). Zone transfers also occur over TCP.
Here are some example queries and responses:

Type	Query	Response
Recursed	`A www.abc.net?`	`www.abc.net 3600 IN A 1.2.3.4`
Cached	`A www.abc.net?`	`www.abc.net 3590 IN A 1.2.3.4`
Authoritative	`A www.abc.net?`	`www.abc.net 3600 IN A 1.2.3.4`
Referral	`A www.abc.net?`	`abc.net 84600 IN NS ns.abc.net`

In the recursed query, we ask our helper server for the address of "www.abc.net" and get back the answer (1.2.3.4) and a TTL of 3600 (3600 seconds, orone hour). Repeating this query on the same helper server, we get the same answer as before, but the TTL is reduced. (When the query was first recursed and returned, the helper server cached the record with the TTL set by the authoritative server it queried; now, ten seconds later, it's telling us how much longer it will cache this record until it expires.) If your DNS query tool shows how long the query took, you should also notice that the cached query returns much more quickly than the initial recursed query.

Querying an authoritative server directly, the response is identical to that given by our helper server (it should be, since the helper queried this very authoritative server for us). But on repeating the query, the TTL doesn't decline—this server isn't telling us how long it *will* cache a record, it's telling us how long we and others *should* cache the record. The referral query (made of a .net delegate) is an NS record telling us that we should query "ns.abc.net" if we want information about the abc.net domain.

DNS Records

There are many DNS record types defined in the original specification and subsequent extensions, but only a few are frequently-used and thus important to understand:

Function	Type	Description
Addresses	**A, AAAA**	A (Address) records associate a name with an IP(v4) address: `www.abc.net IN A 1.2.3.4` There may be more than one A record associated with each name—the server will usually cycle through different orderings (**round-robin**) of the A records when it serves then, allowing for a simple kind of load-balancing via DNS alone. AAAA records are used to point a name to an IPv6 address.

Email	**MX**	MX (Mail Exchanger) records associate names with the servers that are designated to receive email for them, with priority levels. For instance:
		
		`abc.net IN MX 10 mail.abc.net`
		`abc.net IN MX 20 backup.abc.net`
		
		These records would direct an MTA with a message for anything@abc.net to deliver the message over SMTP to the server "mail.abc.net" (since it has the lowest priority). If that primary server is unavailable, the MTA will try to deliver to "backup.abc.net". There may be multiple MX records for a single name, with different priorities (to implement a tiered logic) or with the same priority (another kind of load-balancing).

Authority	**NS**	NS (Name Server) records are used to delegate authority for subdomains. Each domain typically hosts its own DNS records:
		
		`abc.net IN NS ns.abc.net`
		`abc.net IN NS ns2.abc.net`
		
		as well as any delegations:
		
		`east.abc.net IN NS ns-east.abc.net`
		`east.abc.net IN NS ns2-east.abc.net`
		
		There may be many NS records for each name—in fact, they're almost always provided in twos or threes to make sure that at least one delegate server is available to helper servers at all times.

Reverse	**PTR**	PTR (Pointer) records are used for **reverse DNS**. Instead of pointing a name to an IP address, they point an IP address to a name—that way, you can look up the IP address and get an idea of what its canonical or primary name is. (There may be many names with A records pointing to this IP address, meaning that it has many names, but there can be only one canonical name.) Reverse DNS is accomplished via a special domain (in-addr.arpa) which is subdelegated to the ISPs who have authority over the various portions of IP address space on the Internet. A PTR record for 1.2.3.4 would look like: `4.3.2.1.in-addr.arpa IN PTR www.abc.net` The order of the IP address is reversed because DNS names become more specific to the left while IP addresses become more specific to the right. There is usually only one PTR record for each IP address.
Other	**SOA**	There is typically one SOA (Start of Authority) record for each domain, and it indicates that the host serving it is authoritative. The SOA record encodes information about the whole domain: a nameserver, an email address for the responsible party, the zone's serial number, and the refresh/retry/expire/TTL intervals.
Other	**TXT**	TXT records are used to serve arbitrary strings of text. They're frequently used to serve additional information or metadata without creating a whole new DNS record type. Some examples: SPF records (which encode legitimate email senders and policies for a domain); geography (records which encode the locations of hosts); DNS Service Discovery (for zero configuration networking); DomainKeys (public keys used to verify digital signatures which authenticate message senders); licensing records.
Other	**CNAME**	CNAME (Canonical Name) is an aggressive record for aliasing. It basically means that the name on the left should be considered an alias for the name on the right, with respect to all types of DNS records. This means that all the records for the name on the right can be substituted by applications for the name on the left: A records (which is usually what people want, because they're just trying to alias the address), MX records (which is usually an unexpected consequence), even NS and other records. This can get very confusing and dangerous, especially when email is involved. It's almost always clearer to use duplicative A records (when you need to alias an address) and more effective to delegate a subdomain via NS records (if you need to give authority for a name to someone else).

Tools

Because DNS was invented when the Internet was young and UNIX-like systems dominated, the best tools are still on the command line in the UNIX-like world. So when playing with DNS, it's best to either use a computer with a UNIX-like operating system (Mac OS X, Linux, *BSD) or get a shell account on such a system. Otherwise you'll have to use Windows or web versions, which are less powerful.

To learn whether a server is supposed to be a helper server, you just have to call your ISP and ask. To learn about delegates, you can use **whois**. Whois will tell you, for any domain: who owns it, which registrar registered it, details of the registration (creation, expiration, last update) and which nameservers *should* be delegated authority for it.

```
mycomputer > whois abc.net

Registrant:
   J. Morgan
   789 College Avenue
   Berkeley, CA 94678
   USA

   Registered through: AwesomeDomains
   Domain Name: ABC.NET
      Created on: 11-Jun-95
      Expires on: 10-Jun-09
      Last Updated on: 17-Apr-04

   Administrative Contact:
      Morgan, J.  julia@abc.net
      789 College Avenue
      Berkeley, CA 94678
      USA
      5101234567      Fax -- 5101234567

   Technical Contact:
      Morgan, J.  julia@abc.net
      789 College Avenue
      Berkeley, CA 94678
      USA
      5101234567      Fax -- 5101234567

   Domain servers in listed order:
      NS.ABC.NET
      NS2.ABC.NET
```

Much more interesting is verifying information about behaviors. Instead of simply doing a straightforward DNS query and depending on your local nameserver to get things right, you'll need to be able to do manual queries where you control exactly what and who you're asking. A good tool for this is **dig**. We might start by querying the root servers as a more accurate authority check:

```
mycomputer > dig @a.root-servers.net abc.net

;; QUESTION SECTION:
;abc.net.                       IN      A

;; AUTHORITY SECTION:
net.                    172800  IN      NS      B.GTLD-SERVERS.net.
net.                    172800  IN      NS      C.GTLD-SERVERS.net.
net.                    172800  IN      NS      D.GTLD-SERVERS.net.

;; ADDITIONAL SECTION:

B.GTLD-SERVERS.net.     172800  IN      A       192.33.14.30
C.GTLD-SERVERS.net.     172800  IN      A       192.26.92.30
D.GTLD-SERVERS.net.     172800  IN      A       192.31.80.30
```

This response from dig (truncated for clarity) reminds us that the servers at gtld-servers.net are delegates for .net, and helpfully gives us their IP addresses. Stepping down:

```
mycomputer > dig @b.gtld-servers.net. abc.net

;; QUESTION SECTION:
;abc.net.                       IN      A

;; AUTHORITY SECTION:
abc.net.                172800  IN      NS      ns.abc.net.
abc.net.                172800  IN      NS      ns2.abc.net.
```

This matches what we saw from whois, so all is well at this level. We may want to query this domain's delegates to see what the currently correct data is for a particular record. Besides specifying the record and the server you want to query in dig, you can also specify the record type:

```
mycomputer > dig @ns.abc.net mx abc.net

;; QUESTION SECTION:
;abc.net.                       IN      MX

;; ANSWER SECTION:
abc.net.                14400   IN      MX      10 mail.abc.net.
abc.net.                14400   IN      MX      20 backup.abc.net.

;; AUTHORITY SECTION:
abc.net.                14400   IN      NS      ns.abc.net.
abc.net.                14400   IN      NS      ns2.abc.net.
```

If dig isn't quite low-level enough for your problem, you may want to try **tcpdump** or another packet sniffer. It's often quite easy for them to focus on port 53 and decode DNS queries, so you can see exactly who is querying who and what responses come back, to generate a complete picture of what's happening in DNS and where failures might be occurring.

Tricks & Gotchas

Generally, if there's a DNS problem, the best way to find the misbehavior is to work down the chain—start at the root, get delegation information, query the next level down, get delegation information ... until everything checks out or something fails. After determining the correct information, you might next want to query your local helper server to see what's in its cache.

Checking registrars with whois will tell you which servers *should* be delegated authority, but registrars make mistakes like everyone else, and occasionally domain information won't make it from their systems to the root servers. Checking the root servers tells you which servers *are* delegated authority for a domain. The root servers and whois almost always agree, but when they don't things get very confusing, and keeping this distinction straight can help resolve things smoothly, especially when you're on the phone with the registrar's first-level tech support.

Similarly, sometimes the newest DNS records or changes will have made it to one delegate server, but not the others. The result is fragmented DNS where some people get one record and other people another, depending on the delegate server their information comes from. You can query the server independently to find out which ones are misbehaving and inform the host. (Often the problem here is that the change was made on the master server, but hasn't propagated to the slaves, probably because the serial number wasn't updated on the domain's zone file.)

Remember the role/behavior distinction: a server can behave authoritatively even if it hasn't been delegated authority. This frequently causes problems when a domain moves from one host to another, and the first hosts's DNS servers still behave authoritatively for the domain (with the old, now wrong DNS information). Anybody who's using the first host's servers as helpers will get the wrong information.

You can look up the PTR record for an IP address to get its reverse hostname, which will often help you determine who's responsible for it. Better yet, use the -a flag to whois to query the ARIN registry of US IP addresses (or other flags for addresses managed by international registries).

When working with zone files, @ on the left side means the domain name itself. Empty whitespace on the left means whatever name is on the line above. If you want a name to be interpreted as absolute (i.e., not have the domain name appended), terminate it with a dot. And when you make a change, always update the zone's serial number.

When a query or response is too long to fit into one UDP packet it will go over TCP instead, so TCP/53 should be open in firewalls wherever UDP/53 is required.

SMTP

Why we have SMTP

The **Simple Mail Transfer Protocol** (SMTP) was invented in 1982 for sending short text messages between diverse systems connected to the ARPAnet. The network was very different then (much smaller, more personal, more cooperative) and the underlying assumption in designing a network-wide email protocol was that all that was needed was a relatively simple way for one host to reliably send a message to another. The main challenge to overcome was the slow and unreliable network links in use at the time. So adding a layer of reliability was paramount: the protocol was designed to make sure that messages reached their destinations, even if they had to be queued and tried via several different network paths. Security was hardly a consideration; everyone was trusted, and everyone was helpful. In fact, most sites ran **open relays** which accepted messages destined for anyone and tried to forward them on closer to their recipient.

Fast-forward 25 years, and the world of email has changed. A tool designed to relay messages between a few thousand researchers in academia and the military has grown into a communications backbone for billions of people and organizations. As the medium has grown, so have opportunities to benefit from abusing it. Now pressed into the service of transporting a sea of spam, email-borne viruses and other malware, SMTP displays its original design goals—Reliable? Yes. Secure? No.

SMTP Agents & Pathways: Messages relayed between MUAs, MSAs, MTAs, and MDAs

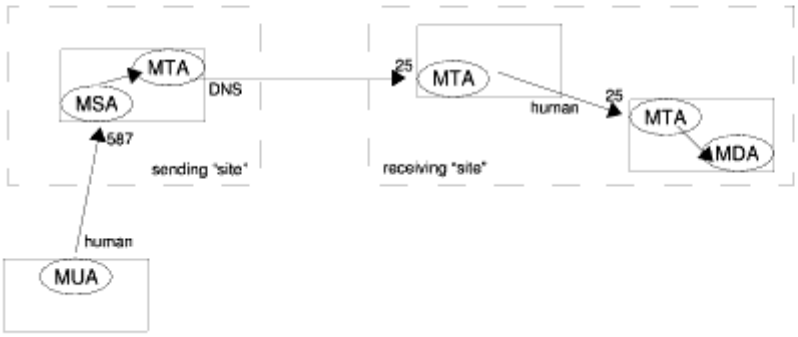
The Clients: Mail User Agents

As in the world of DNS, the client/server model isn't perfect because most SMTP servers also operate as SMTP clients (in that they connect to other SMTP servers). But again one role is exclusively client-like: the **Mail User Agent** (MUA). An MUA is a program for reading and composing email messages: desktop programs, server-based programs, and webmail programs are all examples. The part of their job relevant to SMTP is creating messages and sending them using SMTP, which they do based on how they've been configured by a human. Usually they send all their mail to one (outgoing) server, which has either been provided by their ISP (in which case they probably are unauthenticated and connecting to port 25) or by their email host (in which case they probably must be authenticated, and connect to port 587).

The Servers: Mail Transfer Agents

The **Mail Transfer Agent** (MTA) is the core agent in SMTP. Its role is to receive messages and relay them on. So where does an MTA send its messages? To **the next hop**. This can be anywhere, depending

on how the server is configured (and on how the larger email system which includes it is designed). Some servers simply send all the mail they receive on to the same one place, so the next hop is always the same server. Others might relay mail for different domains to different servers, so the next hop depends on the recipient domain. Sometimes there is no next hop, because the message has reached its destination and should be delivered locally rather than relayed to another server. To distinguish all these situations, the SMTP world developed the concept of a **site**: a collection of interconnected servers which are all under the control of one administrator or organization. It's useful to consider whether a particular hop is occuring within sites, or between sites. For mail that's going between sites on the Internet (usually the most interesting case) an MTA's default behavior is to check DNS for MX records for the recipient domain and deliver the message to the servers found there, in order of their priority. For mail that's going within sites, the MTA has usually been human-configured with next hops for particular domains. The combination of these two behaviors gives rise to the following typical delivery scenario:[2]



So, typically, a human being composes a message using an MUA, which relays the message to its human-configured next hop on port 587 to an MSA, which relays the message to its local MTA, which relays the message to its DNS-configured next hop on port 25 to another site's MTA, which relays the message within its site to its human-configured next hop MTA on port 25, which calls the local MDA to deliver the message. At that point SMTP is through with the message—whether and how it gets transmitted to the recipient is a matter for other protocols like POP or IMAP.

2 You'll note there are two extra agents shown above. The **Mail Submission Agent** (MSA) is really just a special kind of MTA used only when messages are first injected into an SMTP delivery chain—it accepts messages only from MUAs, not MTAs. The **Mail Delivery Agent** (MDA) is a totally different kind of program which doesn't interact with the Internet at all—its job is to receive messages from a local MTA and write them to disk in the proper format and the proper place, depending on the recipient.

The Pathways: How an MTA relays a message over TCP/IP

SMTP is a human-readable protocol that involves exchanges of text over TCP/IP. The typical TCP port for SMTP communications is 25[3]. An MTA (client) initiating an SMTP transaction first connects to the next hop (server) over TCP on port 25. A typical conversation goes like this (client's statements in bold):

	Server answers the connection and begins the SMTP conversation with a status code of 220 (Service Ready) with additional information about its hostname (mail.abc.net), the protocol it supports (ESTMP), and its mail software (Postfix).
`220 mail.abc.net ESMTP Postfix`	
`HELO mycomputer`	Client announces its hostname (**HELO** command).
`250 mail.abc.net`	Server repeats hostname with status OK.
`MAIL From:<george@xyz.net>`	Client announces the envelope sender (**MAIL** command).
`250 Ok`	Server accepts this sender address with status OK.
`RCPT To:<julia@abc.net>`	Client announces the envelope recipient (**RCPT** command).
`250 Ok`	Server accepts this recipient address with status OK.
`DATA`	Client announces it's ready to send message (**DATA** command).
`354 End data with <CR><LF>.<CR><LF>`	Server says 354 (Start mail input) with period terminator.
`From: nonsense123@fake.net` **`To: nonsense456@fake.net`** **`Subject: testing`** **`This is the body.`** **`.`**	Client sends the data portion of message (headers and body) ending with a single period alone on a line.
`250 Ok: queued as D2FE3E0435`	Server accepts message with status OK and its SMTP ID.
`QUIT`	Client ends conversation (**QUIT** command).
`221 Bye`	Server confirms with status 221 (Closing Transmission Channel).

Because SMTP is so easy for humans to read and write, you can converse in it yourself. Most SMTP transactions have this HELO/MAIL/RCPT/DATA/QUIT structure. (Often the conversation will start with EHLO (Extended HELO) command instead if the client wants to test whether the server speaks ESTMP (Extended SMTP) and supports the service extensions to SMTP that entails. EHLO will return much more information about the server's capabilities, including authentication methods and message

3 Port 587 was added for submission in 1998 when it was noticed that SMTP, designed as a message transfer protocol, was also being used as a message submission protocol. Separating submission from transfer in the mail agent is efficient because the separate agents can implement separate policies that are relevant at different parts of the SMTP chain; separating the processes on different ports is also beneficial, as it allows ISPs to manage message transfer traffic separately from submission traffic (for instance, blocking outbound traffic on port 25 to hosts outside their network, thereby stopping lots of spambots and email-borne viruses dead in their tracks).

size limits.) The three-digit status codes returned by the server start with "2" for success, "4" for transient failure, and "5" for permanent failure—these can be helpful in distiguishing temporary message rejections (**deferrals**) from permanent ones (**bounces**).

Nowadays, it's important to remember that the path taken by an email message is not completely described by SMTP. Remember that SMTP was designed to reliably transport messages, and it does this very well. But in our now spam and virus-filled world, email providers have to do a lot more with your mail than simply deliver it reliably. They also have to make sure that you're not overwhelmed by junk mail. So they may apply many different policies and tests and systems and filters to your mail, none of which are part of SMTP (and which may, in fact, interfere with its reliability). A few examples:

- In the early days of SMTP, most servers were **open relays**, happy to receive any mail and forward it on. These days running an open relay is incredibly shameful; servers have to be carefully configured to accept only exactly the mail destined for their site;

- After messages are received by MTAs over SMTP, other software often takes over, **filtering messages** based on their routing, content patterns, reports by other people or sites, or statistical evaluation of the message contents;

- Some systems use the reliability features of SMTP against malware, "**greylisting**" by deferring the first message sent from each sender/receipient/MTA IP address triad (real mail tends to remain queued at the sending MTA, while spam and viruses tend to give up and never retry), "**tarpitting**" by stalling SMTP conversations when the message is suspected to be spam (aiming to tie up the sending processes used by malware to control its rapid spread);

- Other systems **extend SMTP** by adding layers that authenticate the different features of an email message which can otherwise be so easily forged: SASL to validate senders, SPF to validate envelope sender addresses, SenderID and DomainKeys to validate header sender addresses.

The SMTP Message

Email messages sent over SMTP are structured fairly simply, but there are a few layers of encapsulation that can be confusing. Understand them and you'll become a master of SMTP! In the widest view, an SMTP message consists of an **envelope** that "wraps" the message **data**. This data consists of a series of **headers** that wrap the **body** of the message. And the body itself frequently consists of a series of **MIME parts**, separated by boundaries. Let's talk about those three contexts now:

Message = Envelope(Data)

In the SMTP conversation above, the client had to identify the sender and the recipient of the message in the MAIL and RCPT commands. When this message is accepted and queued by the server, these two addresses are maintained separately from the data of the message, and they're repeated in the next SMTP transaction when the message is relayed on to another MTA. But note the nonsense From: and To: addresses we put in the data portion. The sender/recipient addresses in the MAIL and RCPT commands are not required to have any connection[4] whatsoever to the addresses that appear in the From: or To: headers in the message data—no MTA reads, writes, or pays any attention to the addresses found in the headers. To distinguish these special addresses, they're often called the **envelope sender** and **envelope recipient**; together, they constitute the message envelope. Everything else about the message is contained in the data portion sent after the DATA command.

Data = Headers(Body)

The data portion of the message consists of two parts. The headers contain what we normally think of as a whole bunch of metadata about the message, and then the body contains what we normally think of as the actual message.

Here is the message data, with full SMTP headers, that we created in the SMTP conversation above:

```
Received: from mail.abc.net (mail [1.2.3.5])
        by pop.abc.net (8.13.6/8.13.6) with ESMTP id l3EIUGkQ021247
        for <julia@abc.net>; Sat, 14 Apr 2007 11:30:16 -0700 (PDT)
Received: from mycomputer (adsl-71-142-79-226.dsl.att.net [71.142.79.226])
        by mail.abc.net (Postfix) with SMTP id D2FE3E0435
        for <julia@abc.net>; Sat, 14 Apr 2007 11:29:22 -0700 (PDT)
From: nonsense123@fake.net
To: nonsense456@fake.net
Subject: testing
Message-Id: <20070414182927.D2FE3E0435@mail.abc.net>
Date: Sat, 14 Apr 2007 11:29:22 -0700 (PDT)
X-Envelope-From: george@xyz.net

This is the body.
```

The body is easy to get out of the way first: it's simply the line “This is the body.” Everything else in this message is headers. As you can see, SMTP headers take the form of one word defining what's coming, then a colon, then a bunch of text. (The headers above are a very standard minimal bunch, except the X-

4 Of course, they often do have a connection, which is that the MUA which originally composed the message used the header From: and To: addresses to set the envelope sender and recipient when it first submitted the message to an MSA. But nothing requires them to match or even be related. The existence of a hidden envelope outside the message data means there could be several answers to the simple question of what addresses a message is “from” or “to”.

Envelope-From header. Creating **X-headers** by prepending "X-" to some descriptive text is a standard way to name nonstandard SMTP headers.)

Note that we only sent a few of these headers in the original message: the From, To, and Subject. The rest were added by the MTAs that processed the message. The first MTA added the Date: (stamped the moment it received the message) and Message-Id (a random string, here based on the date/time, SMTP id and hostname), because those are required by SMTP. The first MTA also added the first Received header, and then the second MTA added the second Received header. Somewhere along the way, one of the MTAs was nice enough to encode the envelope sender into an X-Envelope-From header (otherwise the envelope sender wouldn't appear anywhere in the message data).

Received headers are usually the most interesting and useful, so let's look at those a little more closely. First off, note that the Received headers appear in reverse order—they're added by each MTA to the beginning of the current message, so they pile up from bottom to top. Since each MTA adds one when it receives the message, Received headers provide a perfect way to inspect the chain of SMTP transactions through which a message was relayed. Starting from the first header (at the bottom):

```
Received: from mail.abc.net (mail [1.2.3.5])
        by pop.abc.net (8.13.6/8.13.6) with ESMTP id l3EIUGkQ021247
        for <julia@abc.net>; Sat, 14 Apr 2007 11:30:16 -0700 (PDT)
Received: from mycomputer (adsl-71-142-79-226.dsl.att.net [71.142.79.226])
        by mail.abc.net (Postfix) with SMTP id D2FE3E0435
        for <julia@abc.net>; Sat, 14 Apr 2007 11:29:22 -0700 (PDT)
```

The first line shows that the initial MUA (us) announced itself in HELO as "mycomputer"; to get a better idea of who it was talking to, the first MTA did a PTR lookup on the IP address of the MUA and received "adsl-71-142-79-226.dsl.att.net." Next comes the MTA's own name (mail.abc.net), the software it's running (Postfix), and the SMTP id it used to identify this message—you can see that this matches the one it mentioned when it accepted the message in our SMTP conversation. Finishing up, it logs the envelope recipient and the time the message was received. The next line has exactly the same structure, but tells the story of the next hop. We can see that although the first MTA calls itself "mail.abc.net" in HELO, the second MTA translates its IP address simply to "mail" (perhaps via a hosts table or internal DNS?). The second MTA calls itself "pop.abc.net", it's running Sendmail (8.13.6 is the version number), and logs the SMTP id, envelope recipient and date.

Comparing the two lines, we can tell a great deal about the message and its transit: pop.abc.net knows mail.abc.net simply as "mail", so they're probably within the same site; although the first hop was conducted over plain SMTP, the second hop used ESMTP;the message was relayed through Postfix and Sendmail; the message spent less than a minute on mail.abc.net before it was relayed to pop.abc.net; and both MTAs are configured in the same time zone.

Body = Boundary/MIME Part/Boundary/MIME Part/Boundary...

The simplest possible message body is just a bunch of plain text, as in our message above—before **MIME** (Multipurpose Internet Mail Extensions), this was all there was to email. Although many messages are still sent in this format, most message sent these days consist of several MIME parts. This structure allows for "attachments" (one part is the main message text, another part is an attached file), messages with multiple formats (e.g., one plaintext part plus one HTML part, so the recipient can choose which to view), and any other message that needs to be richer than simply one unified plain text section. Here's a very basic example:

```
Date: Sat, 14 Apr 2007 11:42:12 -0700 (PDT)
Message-Id: <200704141842.l3EIgC4D021767@mail.xyz.net>
From: George <george@xyz.net>
To: julia@abc.net
Subject: A simple multipart message
MIME-Version: 1.0
Content-Type: multipart/alternative; boundary="5494-19501-841=:9866"

This message is in MIME format. Since your mail reader does not understand
this format, some or all of this message may not be legible.

--5494-19501-841=:9866
Content-Type: text/plain; charset=us-ascii

This is the plain old text part.  You'll see just this line if your email client
is showing you the text part.

--5494-19501-841=:9866
Content-Type: text/html; charset=us-ascii

<html><head></head>
<body><b>This is the fancy HTML part.  You'll see just this line if your email
client is showing you the html part.</b></body>
</html>

--5494-19501-841=:9866--
```

The MIME-Version header signals to the client that the message should be interpreted with respect to the MIME standard, and the Content-Type header establishes the type of the outermost MIME container (which contains the other parts). The type here is "multipart/alternative", meaning that the parts should be viewed by a recipient as alternative versions of the same data (so only one needs to be displayed). Another common container type is "multipart/mixed", used to attach different MIME parts that are not alternatives for each other (attachments, for instance). The "boundary" section of this header establishes which (arbitrary) bit of text will be used in this message to separate the parts from each other (so it's important that this text not appear in any of the actual body parts).

The first paragraph of the body will only appear for clients who don't understand MIME (since it's not within any of the part boundaries), so it's a place to give a message to old email clients. The first boundary is followed by the Content-Type of the first part, and a plain text message follows. After another boundary, the Content-Type text/html is followed by an HTML message, which is closed with another boundary. (If the second part were a binary attachment, like a PDF file or an image, it would be encoded into text via base64 so that it could be sent over SMTP and decoded by the recipient.) So the structure here is:

```
multipart/alternative
      text/plain
      text/html
```

and a receiving client will choose to display either the text/plain or the text/html part. Note how the MIME message structure is like a tree, with abstract container parts which leaf out into concrete data parts. Containers can also contain other containers, creating arbitrarily complex structures like:

```
multipart/mixed
      multipart/alternative
            multipart/mixed
                  text/plain
                  image/jpeg
                  image/jpeg
            multipart/mixed
                  text/html
                  image/jpeg
                  image/jpeg
      text/plain
```

Tools

Working and troubleshooting with SMTP, you can choose to either watch what's currently happening (or recently happened) or make things happen yourself. To check and see how things are currently working (or were recently), MTA logs are extremely informative and useful. I won't include any here (since every MTA is different) but most will tell you—with extremely fine time resolution—which SMTP client connected to them (as identified by HELO, IP, and PTR), the envelope sender and recipient of the message the client tried to relay, its size, and what the MTA did with the message (defer, bounce, deliver, relay), all identified by the SMTP ID assigned by the MTA to the message. Because the MTA will also log the SMTP ID assigned by the next hop, you can use SMTP logs to track the historical progress of a message through multiple servers.

To make things happen instead of waiting, you can play SMTP client and connect to MTAs yourself. Using dig to determine MX records for a particular domain, you can then start an SMTP session with an MTA of your choice by using **telnet** to open a TCP connection on port 25 (or 587):

```
mycomputer > telnet mail.abc.net 25
Trying 1.2.3.5...
Connected to mail.
Escape character is '^]'.
220 mail.abc.net ESMTP Postfix
```

Send whatever commands you want (you'll probably find the EHLO/MAIL/RCPT/DATA sequence most useful) and terminate the session with QUIT.

Tricks & Gotchas

SMTP delivery problems are often a blame game between ISPs and email hosts. MTA logs can often make a confusing situation extremely straightforward by showing exactly how a message was (mis)delivered. (And end users love to know that their message is a discrete, concrete parcel that can be tracked in delivery like a FedEx package.)

Mail doesn't always work via MX records the way we expect it to: when an MTA is relaying a message to another site and can't find MX records for a domain, it will usually try to deliver to the A record for that domain as a last resort; spammers will often try to deliver to lower-priority MXs first, assuming that backup/secondary servers have weaker anti-spam protection.

To explore the "forgery" possibilities of email, send (via a manual SMTP session) a message with an arbitrary envelope sender and/or header sender. Try a message where the envelope recipient and header recipient don't agree. Remember this distinction! Logs and SMTP sessions are all about envelope senders/recipients, while end users often only see header senders/recipients.

You can also use manual SMTP sessions to validate the fitness of a particular sender or recipient or MTA. If you try sending a message to a particular server to test whether it will accept that recipient and you're rejected, you're probably either trying a bad recipient or the wrong server (though sometimes your message might be rejected on the basis of its content without a clear indication).

Received headers are indispensable for inspecting the routing history of a message or for determining where delays might have occurred (be careful with time zones). However, Received headers can be "forged" just like any other message part—this is a common trick spammers use to try to hide their footsteps. You can usually spot when the real Received headers start by working backwards from the final, trusted delivery and correlating the hostnames in adjacent headers. When the correlation breaks down, the rest is forgery.

Appendix: Group Role Playing Exercises

Modeling a DNS A record lookup

Roles(6): web browser, local helper server, root server, .org delegate, foo.org master delegate, foo.org slave delegate

Initial query: Web browser wants to load website (www.foo.org). Consults resolv.conf. Asks local helper server. Local helper server consults files, is not authoritative. Checks cache for www.foo.org A record. Not there. Checks cache for foo.org NS record. Not there. Checks cache for .org delegate. Not there. Asks root server: NS org?. Root server: these are the delegate servers for org. Caches record and TTL. Asks .org delegate: NS foo.org?. .Org server: these are the delegate servers for foo.org. Caches record and TTL. Asks foo.org delegate: A www.foo.org?. Foo.org server: This is the A record for www.foo.org. Caches record and TTL. Local helper server returns result to web browser. Web browser asks webserver for website.

Cached query: Web browser wants to load website (www.foo.org). Consults resolv.conf. Asks local helper server. Local helper server consults files, is not authoritative. Checks cache for www.foo.org A record. There. Local helper server returns result to web browser. Web browser asks webserver for website.

Record change: On the master server, move www.foo.org website. Without NOTIFY, slave and master now don't match until refresh interval elapses. With NOTIFY, master tells slave to get new zone, and they match. Web browser does lookup, gets cached response over and over - website is unreachable. When local helper server expires cached record from its cache (TTL expired), local helper server reloads using cached foo.org delegates - after that, record is up to date and website is reachable by browser.

Modeling the lifecycle of an email message

MUA1 creates and submits to MSA, sends to MTA1, between sites to MTA2, within site to MTA3, delivered by MDA, retrieved by MUA2. Draw a diagram of the servers and sites, and note the TCP ports and DNS queries involved.