The following paper was originally published in the
Proceedings of the USENIX Symposium on Internet Technologies and Systems
Monterey, California, December 1997

# A Highly Scalable Electronic Mail Service Using Open Systems

Nick Christenson, Tim Bosserman, David Beckemeyer
*EarthLink Network, Inc.*

# A Highly Scalable Electronic Mail Service Using Open Systems

Nick Christenson, Tim Bosserman, David Beckemeyer
*EarthLink Network, Inc.*
*Information Technology*
*Pasadena, CA 91107*
*npc@earthlink.net, tboss@earthlink.net, david@earthlink.net*

## Abstract

*Email is one of the most important of the Internet services. As a very large, fast growing, Internet Service Provider, EarthLink requires a robust and powerful email architecture that will support rapid expansion. This paper describes such an architecture, its motivations, its future, and the difficulties in implementing a service on this scale.*

## 1 Introduction

Electronic mail has a special standing in the ranks of Internet services. Of the direct services an ISP provides to its subscribers, email is certainly one of the most important. As a consequence, it requires special attention to keep email running as well as expected. Additionally, there are several issues that are far more problematic for email than for other services. Email typically requires more resources than any other service. This is because the storage needs, the processing power, and the bandwidth requirements are extreme. Furthermore, there are problems regarding authentication and provisioning that are often not required for other services.

Despite its criticality, little work has been made available publicly on robust, large scale electronic mail systems. The few references we have found, such as [Grubb96], have neither addressed what we consider to be the key problems, nor have they been able to scale to the capacity that we require. This isn't too surprising. Providing email service for hundreds of thousands or millions of users is a problem nobody had to solve before the advent of the national or international Internet or on line service provider. Until now, none of these organizations have chosen to come forward and publish their service architecture.

Additionally, the architecting of very high performance truly distributed services is still in its infancy. The issues of distributed storage and load balancing have few, if any, available solutions that are both robust enough and perform satisfactorily for our purposes.

The astute reader will certainly notice that the architecture we describe here bears a great deal of similarity to what we have already described as our news service architecture [Christ97a]. This, of course, is no accident. We've found a general set of principles which we have adapted to meet the needs of both services, and many of the important issues discussed in that article are equally applicable here.

In the design of any of our service architectures, we have several requirements that must be met before we would consider deployment. For email, the first of these is message integrity. It is absolutely essential that messages, once they are accepted by our system, be delivered to their proper destination intact. Second, the system must be robust. That is, in as much as is possible, the system should survive component outages gracefully. Additionally, the entire system design should minimize the number of single points of failure. Third, the system must be scalable. When EarthLink began deployment of the current architecture, in January of 1996, we had about $25,000$ subscribers. In September of 1997, EarthLink provided email service for over $350,000$ subscribers with a 99.9+% service uptime record. In fact, we expect the current system to scale to well over $1,000,000$ users without significant alteration of the architecture as presented here. Moreover, one should be able to accomplish the scaling of any service with a minimum of outage time, preferably with none. In all cases the performance of the service must be at least adequate, and the service must be maintainable. Problems must be easily recognizable, and it should be obvious, whenever possible, what is the cause of the outage. Further, its solution should be easy to implement and, in the meantime, the impact of the outage should be small and locally confined. Finally, we would like the service architecture to be cost–effective, not just in terms of equipment acquisition, but, more critically, in terms of maintenance.

## 2 Architecture Description

There are several logically distinct components which make up the operation of EarthLink's email service. The first, which we call the "front end" of our email system (front defined as the portion which receives data from the Internet) are the systems that receive mail for "username@earthlink.net". These machines are also called the SMTP machines. The second component is the POP service, the servers to which subscribers connect to retrieve their mail. These same computers also send the mail originating from our subscribers to the Internet. (At the time of this writing, EarthLink has not deployed an IMAP service.) The third component is the file servers, which do nothing except store the mailboxes, mail queues, and auxiliary files associated with the email service. The fourth component is the authentication database which holds the username/password information, information on where mailboxes are stored, and data on auxiliary email services to which that account may have subscribed. This architecture is demonstrated graphically in Figure 1 included at the end of this paper. All the servers we use in this architecture, except for the file servers, are running some flavor of Unix.

With the exception of our file servers and the authentication database servers, our architecture calls for all of the servers involved in our email service (as well as all our other services) to be dataless. That is, each server should store on local disk its own operating system, service software, swap space, temporary file storage for nonessential data—and nothing else. This allows us to add or subtract servers from service with which the Internet or our subscribers interact without affecting the data stored.

### 2.1 Front End

Mail Exchange (MX) DNS records for earthlink.net and mail.earthlink.net point, with high preference, to a series of servers in Round Robin. These servers all run a recent version of the freely distributable stock sendmail [Allman86] as their SMTP MTA (Simple Mail Transfer Protocol, Mail Transfer Agent; see [Postel82] for details). Writing and maintaining an SMTP MTA is a difficult and expensive task. Therefore, we have geared our architecture to allow us to use sendmail without any source modification. Since sendmail typically undergoes several significant revisions during each calendar year, it's important that we be able to use sendmail in a form as close to the stock distribution as possible, since updating a modified sendmail every few months to reflect local modifications would be about as time consuming as maintaining our own MTA.

An electronic mail message to be delivered to "username@earthlink.net" follows the DNS MX records for earthlink.net. We maintain several machines with high precedence MX records using Round Robin DNS to distribute the load. If any of these machines become overloaded or otherwise unavailable, we maintain a single machine with a lower precedence MX record to act as a spillway. This server does not deliver email directly, but it does hold it for forwarding to the first available front end server. Our backup MX machine could easily be configured to deliver email itself, but we have chosen not to do this to protect against the possibility of transient errors in the mailbox locking process. This way, if the locking scheme becomes overloaded, we have a server that can still accept mail on behalf of the "earthlink.net" domain. It is our intention to minimize at all times the amount of email queued around the Internet for delivery to EarthLink.

The key to using the stock sendmail in our architecture is to insure that the sendmail program itself attempts to do no authentication or lookup of user names. This is actually quite simple to do. One merely must remove the "w" flag in the entry for the local delivery agent in the `sendmail.cf` file. Even if one runs a standard authentication scheme, we've found that this modification provides a considerable performance boost if one has a large, unhashed (i.e. linear lookup) `passwd` file, and the service is not completely inundated with email intended for non–existent users.

The portion of the mail reception service that we did modify heavily was the mail delivery agent. This is the program that receives the mail from the MTA and actually appends it to the user's mailbox. On most systems, this program is `/bin/mail`. The sendmail distribution provides a delivery agent called `mail.local` which we have rewritten to use our authentication methods and understand how we store mailboxes. This is a small program which hasn't changed substantially in years, so it is easy to maintain; hence, it is a better place to add knowledge about our email architecture than a moving target like sendmail.

In addition to authentication and mailbox location, the mail delivery agent also knows about mailbox quotas which we impose on our subscribers. If the current mailbox size is over the quota for that user, the default being 10 MB, then the message is bounced back to the MTA with reason, "User npc, mailbox full." In addition to preventing resource abuse on the part of subscribers, this also helps mitigate possible damaging effects of mail bombing by malicious people on the Internet. We believe that a 10 MB quota is quite generous, especially considering over a 28.8 modem using very high quality line speeds and no network bottlenecks, one could expect to take over an hour to download the contents of a 10 MB mailbox.

### 2.2 POP Daemon

What we call the "back end" of our architecture is a set of machines using Round Robin DNS which act as the POP servers. They are the targets of the A records, but not the MX records, for earthlink.net and mail.earthlink.net. These are the servers to which the subscribers connect to retrieve and send email. If these machines receive email bound for user@earthlink.net (from our subscribers, Internet machines compliant with the SMTP protocol must follow the MX records and send this message to a front end machine), these messages are redirected to our SMTP machines at the front end. The POP servers do no local delivery. They do, however, deliver directly to the Internet. We've debated the notion of having the POP servers forward mail on to yet another set of servers for Internet delivery, but have thus far elected against it. The benefit to doing this would be further compartmentalization of physical server function by logical operation, which we consider to be inherently good. We'd also reduce the likelihood of POP session slowdowns in the case that a subscriber floods the server with an inordinate amount of mail to be delivered. If the POP and Internet delivery functions were separate, the POP server would expend very few resources to hand this mail off to the delivery servers, whereas it would otherwise be required to try to send and possibly queue this mail itself. On the other hand, we don't observe this being a significant problem. Additionally, if we had a separate Internet delivery service within our mail architecture, we'd have to deploy an additional machine to maintain our complete N+1 redundancy to every component, at additional cost. Someday, we probably will make such a separation, but it does not seem to be justified for the present volume.

Like the delivery agent, the POP daemon must also know about both our modified authentication system and mailbox locations. The base implementation we started with was Qualcomm's POP daemon version 2.2, although like `mail.local`, we have modified it substantially. In the near future, we plan to completely rewrite the POP daemon tuned for efficiency in our environment implementing many of the lessons learned from developments in Web server software.

### 2.3 Mailbox Storage

On a conventional Unix platform, mailboxes are typically stored in `/var/mail` or `/var/spool/mail`. The `passwd` file, used to store valid user names for incoming mail and to authenticate POP connections, is usually located in `/etc`, and mail which cannot be delivered immediately is stored in `/var/spool/mqueue`. This is where our mail architecture started out as well, but we made some significant changes as we went along.

As with Usenet news, we use the Network Appliance [Hitz95] family of servers as our network file storage for essentially the same reasons: Very good performance, high reliability of the systems, easy maintenance, and the advantages provided by the WAFL filesystem [Hitz94].

Due to performance considerations, the spool is split across several file servers and each is mounted on the SMTP and POP servers as `/var/mail#`, where `#` is the number of the mount point, in single digits as of this writing.

Currently, we're using version 2 of the NFS protocol. While version 3 does give some significant performance benefits, we give it all back because of the implementation of the READDIRPLUS procedure [Callag95] which prefetches attribute information on all files in that directory, whether we need them or not. Since we store a large number of files in the same directory and are only interested in operating on one of them at a time, this is significant overhead that we don't need. On balance, for our email system, the performance difference between versions 2 and 3 of the NFS protocol is so small as to defy precise measurement. We typically change it whenever we suspect the current version might be responsible for some strange behavior we notice. The NFS version has never turned out to be the problem, so we usually leave that version in place until we feel the need to change it again in order to eliminate it as a factor in some other problem we face.

Even though the Network Appliance's WAFL filesystem provides excellent protection against the performance penalties one normally encounters when there are very large numbers of files in a single directory using most other file systems, there are still significant advantages to breaking them up further. Since we have more files and require more storage and throughput than we can realize with any one file server, we need to split the spool up and provide some mechanism to locate mailboxes within this tree. So, we create a balanced hash for each mailbox over 319 possible subdirectories (the prime base of the hash) and divide these directories over the number of file servers that compose the spool. Thus, a path to a mailbox may look something like /var/mail2/00118/npc. The POP daemon and the local delivery agent are the only parts of the mail system that need to know about these locations.

Once we have this mechanism for multiple locations of mailboxes in place, we are able to extend this to allow us to dynamically balance the mailboxes or expand capacity. In addition to the notion of the "proper" location of each mailbox, both mail.local and popper (the POP daemon) understand the notion of the "old" mailbox location. If the system receives email for a given mailbox, mail.local checks the "proper" location for the mailbox, and if it finds it there, appends the message in the normal manner. If the mailbox isn't there, it checks the "old" location for the mailbox. If it is found there, mail.local locks the mailbox in both locations, moves it to the new location and appends the message in the normal manner. If the mailbox exists in neither place, it creates a new mailbox in the "proper" location. The POP daemon also knows this information. It looks in the "proper" location first, and if it is not there, it consults the "old" location. In either case, it operates on the mailbox entirely in the place where it was found.

Only mail.local actually moves the mailbox. We felt that it would be better to confine the mailbox moving logic in the simpler of the two programs. Because the mailbox can only be in one of the two places and the delivery agent and POP daemon use a common locking system (described below), there's no danger of confusion as to the mailbox location.

The data on what constitutes the "old" and "proper" mailbox locations are kept in the authentication database (explained below), and this information is returned to the client process when authentication information is accessed.

This feature has a major benefit for our mail system. This allows us to move large numbers of mailboxes around without interrupting service. For example, if we have three file servers containing mailboxes and they are either getting to be full or running out of bandwidth capacity, we can create a new mount point, /var/mail4 for instance, mount a new file server on the mail servers, create the hash value subdirectories that will reside there, and then slide a new mail.local and popper in place (POP daemon first!) that know which of the subdirectories from each of the first three file servers will now be housed on the fourth. Then, as mailboxes receive new mail, they are moved onto the new file server. After a few hours, days, or weeks (depending on how much of a hurry we're in), we can start a second process of individually locking and moving mailboxes independent of any other activity on the systems. Thus, we have now expanded our email system without any downtime.

### 2.4 Authentication

One thing we quickly realized is that the standard Unix authentication systems were wholly inadequate for a service of this magnitude. The first problem one runs into is that depending on the specific operating system, one is typically confined to between 30,000 and 65,535 distinct user identities. Fortunately, since none of these users have shell access to these servers (or any access other than POP access), we can have a single UID own every one of these mailboxes as long as the POP daemon is careful not to grant access to other mailboxes without re–authenticating.

While this postpones several problems, it isn't sufficient by itself to scale as far as we'd like. First, several Unix operating systems behave quite strangely, not to mention inappropriately, when the passwd file exceeds 60,000 lines. This isn't completely unexpected—after all how many OS vendor test suites are likely to include these cases—but some of these problems manifest themselves a great distance from the passwd file and, thus, can be difficult to track. Just as important, when the passwd file gets this large, the linear lookups of individual user names become expensive and time consuming. Therefore, the first thing we did was make a hashed passwd–like file using the Berkeley NEWDB scheme [Seltze91] that both popper and mail.local would consult for authentication. This eliminated the need to carry a large passwd file and greatly increased performance of the system. A separate machine working in a tight loop continually rebuilt the hashed passwd file as the text file was continually being modified by the the new account provisioning system.

The next logical extension of this was to store the passwd file in a SQL database and replace getpwnam() calls with SQL equivalents. This provides another quantum improvement. First, this eliminates the necessity of continually rebuilding the hash file from the flat file, with savings in processor and delay times for user account modification. Second, this database may be used by other applications including RADIUS [Rigney97], Usenet news, etc.... Third, it's a logical place to store additional important information about that account. For example, when a username lookup by `mail.local` or a username/password pair is authenticated for `popper`, the "old" and "proper" mailbox locations are returned to the application rather than having to be stored in flat files on the system or hardcoded into the respective binary. We also intend to use this database as a repository for a great deal more information, for example storing variable mailbox quotas and lists of domains from whom to refuse mail on a user by user basis, etc....

Obviously, this database is critical to not only the operation of our electronic mail system, but to other components of our overall service architecture as well. If the authentication service isn't operating, electronic mail comes to a halt. Because of this, we have taken special pains to make certain that this application is always on line by using a clustered system with failover using a dual attached storage unit for the database to meet our high availability requirements. If it becomes necessary, we can still fail over to the old common hashed `passwd` file with only a marginal loss of functionality and performance.

## 2.5 File Locking

In any distributed system, concurrency issues are of paramount importance. In our email system, these manifest themselves in terms of file locking. It is so important, we have given the topic its own section in this paper to discuss the issues which the implementor faces.

### Yesterday

For data stored on local disk, `flock()` suffices to ward against two processes attempting to process the same message or modify the same mailbox at the same time. Since all of our persistent data is accessed over NFS, this presents some significant problems for us.

When using `flock()` on an NFS mounted file system, these calls get translated to requests via `rpc.lockd`. Now, lockd isn't the most robust file locking mechanism ever devised. It isn't advisable to bank on lockd working entirely as advertised. In addition to this, many systems have lockd tables too small for our purposes. We can routinely require thousands of outstanding lock requests on a given NFS mounted filesystem at any one time, and few commercial solutions have lock tables large enough and/or lock table lookup algorithms fast enough to meet our needs.

### Today

Therefore, wherever possible, we use the file system to perform locking. This consists of requesting an `open()` system call to create a new (lock) file with the `O_EXCL` flag of a file of a predetermined name, typically mailboxname.lock, in a given location, which would typically be the mail spool. In our case, in order to keep the spool directory sizes down as much as possible and performance as high as possible, we store these files on their own shared file system.

This may set off alarms in the heads of those familiar with NFS. One might well ask, "How can you be certain that this is atomic on an NFS system? How do other clients know that one has locked a given file?" Recent implementations of NFS client software ignores the attribute cache on `open()` calls which attempt to exclusively create a new file. Note, however, that other `open()` calls do not ignore the attribute cache. This means that if a process's exclusive `open()` on the lock file succeeds, that process has successfully locked that file. This allows us to use file locking on the mailboxes, as long as we are mortally certain that all NFS clients operate in exactly the same way. One can find both NFS v2 and v3 implementations that behave this way. It cannot be overstated how important it is to be certain that all NFS clients behave in this manner.

It is always possible that the process which creates the lock will die without having the opportunity to remove it. For this reason, all processes creating locks must touch the lock files to update their attributes periodically so that if these processes die, after a certain amount of time other processes will know that an existing lock is no longer valid and can be eliminated. Therefore, we need a function that, again, will bypass the cache and be guaranteed to immediately update the attributes on the lock file.

Let us suppose that one process on one NFS client has created a lock on the mailbox "npc". Let us also suppose that a process on a different NFS client then tried to lock that mailbox immediately afterwards and discovered the existing lock, as it must. It's always possible that the first process has somehow died, so it's important to understand how long the second process must wait before it can assume that the first process no longer exists, at which time it can delete the lock file, lock the file itself, and perform operations on that mailbox. Again, let us suppose that all the NFS clients are set to refresh their mailbox lock every five minutes, and that the NFS attribute cache is set on each client to be three minutes.

One scenario is for a process on client1 to successfully lock the mailbox and then have client2 immediately attempt to lock the same mailbox and fail. At this point, the information on the locked mailbox is saved for three minutes, the duration of the attribute cache, after which client2 gets the same attribute information as before, because five minutes has not elapsed, therefore client1 has not yet refreshed the lock. At the five minute mark, client1 refreshes the lock file using `utime()`, since it also bypasses the NFS cache and operates synchronously on the lock file, but client2 has not noticed because it will be looking at the attributes in its cache until the six minute mark, when its cache expires, and it can now gets the updated information. This is represented graphically in Figure 2.

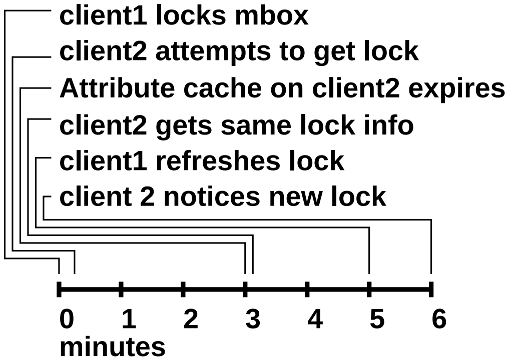
client1 locks mbox
client2 attempts to get lock
Attribute cache on client2 expires
client2 gets same lock info
client1 refreshes lock
client 2 notices new lock
0
1
2
3
4
5
6
minutes

**Figure 2**

The worst case scenario is presented in Figure 3. Here we have client1 creating a lock on a mailbox and then immediately dying. Just before the lock is scheduled to be updated, client2 attempts to lock the mailbox and fails. Client2 cannot learn that the lock hasn't been updated until just before the eight minute mark, at which point it has license to remove the lock file and proceed with its actions.

Unfortunately, this potentially gives us a window of eight minutes in which real users may not be able to access their mailboxes under pathological conditions. For example, if the subscriber interrupts a POP session at the wrong moment, the POP daemon on the mail server may exit without cleaning up its lock file. Further, we explain below why we must delay even longer than this to allow for other concerns.

If the file servers ever get saturated with requests, the server can seem to "disappear" to client processes for many seconds or even minutes. This can happen as part of normal subscriber growth if one does not upgrade the capacity to handle load before it is needed. In these circumstances, problems usually manifest themselves as a sudden change in performance response from acceptable to unacceptable over a very small change in load. The mathematicians would call this a catastrophic response, where "...gradually changing forces produce sudden effects." [Zeeman77] If a file server's load is near, but not at the critical point, it can be pushed over the edge by a sudden change in the profile of normal email use or by a small number of malicious or negligent individuals.

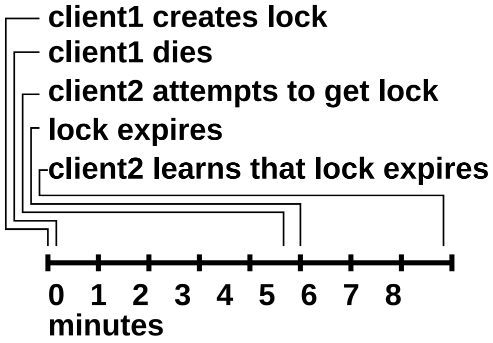
client1 creates lock
client1 dies
client2 attempts to get lock
lock expires
client2 learns that lock expires
0
1
2
3
4
5
6
7
8
minutes

**Figure 3**

It is self–evident that one wants to provide enough surplus performance to prevent small changes from breaking the performance envelope, and certainly we strive for this, yet it is not always possible. As an example, consider a two week period in August 1996 where the total volume of email EarthLink was called upon to handle doubled for reasons that are still not fully understood. While not routine, these events are not uncommon in the ISP business and, because the subscriber has a much greater ability to impact service, represent a fundamental difference between providing Internet services and, for example, providing electric power or dial tone service.

In any case, when one enters into one of these catastrophic regimes, one often encounters pathologic behavior on the part of any or all of the components of the service. Client requests can come too fast for the server(s) to handle; consequently the RPC packets can get dropped before they are processed. This can result in retransmissions by multiple clients, and on top of an already saturated system, the problem is compounded.

Let us suppose that we have a saturated system where the client base demand is 105% of the server's capacity to deliver it over a given period of time, not counting the load put on the server because of retransmissions. Each of these clients will now retransmit their requests after a number of tenths of seconds specified by the `timeo` value in the `/etc/vfstab` or equivalent file. If this request does not receive a response, the client waits for twice `timeo` and retransmits again. This process is repeated until the value of the `retry` variable is reached. If `retry` is exceeded, then the client prints a message, typically "NFS server raid not responding, still trying," and continues to retry at the maximum retry value. This maximum value will never exceed 30 seconds [Stern91]. Under these conditions, we cannot achieve a "steady state" condition, the amount of traffic grows, quite dramatically, without bound until something breaks.

If this condition were to persist for 30 minutes, at this time as much as 25% of the requests sent to the servers may be over 5 minutes old. Note that this represents a true pathological condition, it's highly unlikely that a client machine would either be able to maintain this load given the lack of responsiveness of the server, or that the client load would be constant, but we haven't yet developed our mathematical models sufficiently to account for all the known variables, so we're being conservative. Given these assumptions, if we are adding $2,000,000$ new email messages to our spool in a day, a half an hour of operation at this level of saturation with a lock timeout of only 5 minutes, we must expect there to be on the order of a thousand mailbox corruptions due to multiple processes proceeding to modify mailboxes on the assumption that they have exclusive access to it. This is because they have encountered expired lock files which are actually valid, the owning process just hasn't been able to get the server to ack it's update of the lock file. The mathematics behind this analysis and an in depth examination of the ramifications of this will be fully explored in [Christ97b]. Therefore, it is important that our locking mechanism allow for the possibility that a client process may not be able to get their request through to the server for several minutes after the normal locking timeout window has closed. We use a lock timeout value of 15 minutes to allow for this possibility.

With regard to locking, one area of concern we have is with sendmail. Current versions want to use `flock()` to lock files in mail queues. On our email system, the depth of these queues is extreme and the number of processes that can concurrently be trying to drain them can be as high as several hundred per machine, requiring a large number of outstanding lock requests at any one time, often too many for either the client lock daemon or the file server to accommodate. Because of this, we have two choices. Either we can put the mail queues on locally attached disk, violating our stateless architecture principle and losing the benefits of the WAFL file system in handling directories with large numbers of files, or we can modify sendmail to use a different locking mechanism, thus violating our intention to use an unmodified SMTP MTA. Fortunately, the current sendmail implementation has very modular locking code which can be easily replaced without fundamentally altering the distribution. However, we'd like any folks working on sendmail to consider allowing a preference for various locking mechanisms to be #define'd in the source code.

### Tomorrow

While the mailbox locking mechanism we've just described has worked satisfactorily, it is not without its drawbacks. One drawback is the fact that locks may be orphaned, and other clients must wait up to 15 minutes before being able to assume they are no longer valid. Another drawback is that the synchronous NFS operations we employ greatly increase the load placed on the NFS servers which hold the lock files.

Therefore, we are in the process of designing and building our next generation lock management system. In accordance with our design parameters, what we really want is a distributed lock system with no single points of failure. It has to maintain state in the case of a crash or hardware failure, and it must be able to handle at least several hundred transactions per second.

We tried using a SQL database for this purpose, but we were not satisfied with the performance. A program like a large commercial database such as this requires too much overhead to be efficient in this manner. However, we can learn from the database style locking mechanisms and, essentially, strip away those portions of the database system which we don't need to create our own lean and mean network lock server.

We plan to deploy two machines clustered together around a shared RAID system to act as our lock service. If the primary machine were to suffer some form of failure, the other would take over with a target transition time of less than five seconds. We intend to deploy the same hardware configuration that we use for our authentication database. All the lock requests get written to the file system using unbuffered writes before they are acknowledged so that in case of machine failure there is no loss of state.

The clients open up a socket to the lock daemon on the lock server and request a lock for a given mailbox, which the daemon either accepts or denies. If it is denied, the client waits for some pseudo–random time and tries again. We project that this system will scale well into the millions of mailboxes for a single set of lock managers. To get this scheme to scale indefinitely, it's a simple matter of having the clients query different lock servers for different ranges of mailbox names.

## 3 Operation

One of our primary design goals was to deploy a system that would be cost–effective to maintain. This service accomplishes those goals in several ways.

First, by centralizing authentication in a single system, we reduce the problems associated with both maintaining multiple parallel authentication systems and insuring that they are synchronized. This is a considerable overhead savings.

Second, one of the key criteria in selecting the Network Appliance as our storage system was its ease of maintenance. Because its operating system has been stripped down, eliminating functionality not necessary to its operation as a file server, the server is less likely to fail and, if it does fail, it is easier to discover and remedy the problem due to the greatly reduced number of degrees of freedom presented by the operating system.

Third, because the POP servers themselves are dataless, they require much less maintenance than their stateful equivalents. The only files which differ between these computers are those that contain their host names and/or IP addresses. This means that new servers can be brought online in a very short time via cloning an active server. Just as significant, it means that since these machines contain no important persistent data (aside from the operating system), there are few reasons for system administration to log on to the system and make changes. This helps eliminate one of the arch–nemeses of distributed computing—"state drift," the tendency for systems intended to be identical or nearly identical to become more and more different over time.

At EarthLink, one of the things we do most often is to grow an existing service to accommodate more subscribers. The efforts we have made to allow this to happen easily and with a minimum of interruption contribute greatly to lowering the cost of operation. We've already explained how we use the concept of "old" and "proper" mailbox locations to scale both file system storage and bandwidth by adding additional file servers easily and with no downtime. The network implementation we're using at this time is switched FDDI, which also scales well. As we've already mentioned the POP servers are dataless and, therefore, should lack of these resources present a problem, in very little time, and again, with a minimum of effort, we can clone and deploy a new system. This results in our email service being extremely scalable on short notice.

We attempt to maintain $N + 1$ redundancy in every possible component of the system. Our data storage systems use RAID to protect against single disk failure. We keep extra data storage servers near–line in case of failure for rapid exchange with the downed system. We keep extra FDDI cards in the switch and an extra switch chassis nearby in case these components fail. We also keep one more SMTP and POP server online than loading metrics indicate is necessary. Thus, if one fails, we can pull it out of Round Robin DNS without impacting service, aside from the problems caused by the initial component failure. Additionally, we get the benefit of not having to repair the failed server immediately. Instead, we can take time to ensure that everything else is in proper running order, and then we can diagnose and repair the failed server at our leisure. On top of all this, we use a monitoring system that flags problems with each component of the service in our Network Operations Center, which is staffed 24x7x365 and contacts appropriate on–site personnel.

## 4 Shortcomings

We consider the architecture presented above to have considerable merit as one of the better solutions available for satisfying high volume mail service. It is, of course, not without its limitations, some of which we mention here. One of the first problems is with sendmail as an MTA. When Eric Allman developed the original sendmail, it was not envisioned that it would still be in service over fifteen years later and be pushed, rewritten, and extended to the extent that it has. It is a testament to the skill of its creator and maintainer that it has performed as well as it has for this long. Nonetheless, if one were to code an SMTP MTA today, we doubt anyone would want it to take the form of sendmail. Despite this, we don't see an MTA that would provide enough significant advantages that we would want to migrate to it in the immediate future. Of course, these statements about sendmail could have been uttered five years ago without alteration. The bottom line is that we would prefer to run an SMTP MTA that is tighter, more efficient, and has fewer potential places for security bugs to creep in, but there isn't one available that meets our needs at this time.

Probably the biggest problem with our architecture is that due to the nature of NFS, when we add additional file servers to address our performance and storage needs, we end up adding multiple single points of failure. Despite the fact that the Network Appliance file servers have been quite stable and recover quickly from problems, we feel that this is not easily scalable forever. Therefore, it is our opinion that at some point we need to abandon NFS as our distributed systems protocol for something better.

Our ideal protocol would be very high performance; be completely distributed and, thus, highly scalable, local failures would cause local, not global outages, and would allow for redundant storage that eliminates local single points of failure. Unfortunately, given the current state of distributed computing, it's hard enough to find a system that adequately addresses one of these points, and nothing seems close to providing good solutions for all of them. Consequently, we are currently in the process of designing our own distributed system to accommodate our next generation architecture requirements.

## 5 Current Data

Today, the system described here is in operation as EarthLink's core electronic mail system. At the time of this writing, this system supports about 460,000 mailboxes for over 350,000 users. The system processes, incoming and outgoing, over 13,000,000 email messages each week. This means we average about 20 incoming messages each second. We average about 20 new POP connections/second and hold open about 600 concurrent active POP daemons at peak time, with spikes to over 1000 concurrent outstanding POP connections at any one time.

## 6 Conclusion

In conclusion, we believe we have architected a mechanism to extend a standard, freely distributable, open systems email system to handle from hundreds of thousands to millions of distinct email accounts with a minimum of modification to the underlying components. We also believe that this system meets, to the best of our ability to deliver, the required criteria we set out in the Introduction.

## 7 Acknowledgments

The authors of this paper are by no means the only folks who have put a lot of effort into the development and operation of this system. We wish to especially thank Jay Cai and Max Chern who did a significant portion of the software development on this system. Thanks to Steve Dougherty and Mark Wagnon, who provided helpful comments, and to Jim Larson, who provided valuable input on the precise mathematics of packet retransmissions. Also, a great deal of thanks go to Trent Baker and his system administration team who maintain all our services: Gianni Carlo Contardo, Jason Savoy, Marty Greer, Alwin Sun, Horst Simon, Jewel Clark, Tom Hsieh, Lori Barfield, David Van Sandt, Larry Wang, Hong Zhang, and Kenny Chen. We also wish to extend a special thank–you to Scott Holstad who made many excellent editorial improvements to early versions of this paper.

## References

[Albitz97] P. Albitz, C Liu, *DNS and BIND, 2nd Ed.*, O'Reilly & Associates, Inc., Sebastopol, CA, 1997, p. 212.

[Allman86] E. Allman, Sendmail: An Internetwork Mail Router, *BSD UNIX Documentation Set*, University of California, Berkeley, CA, 1986.

[Callag95] B. Callaghan, B. Pawlowski, P. Stau–bach, *RFC 1813, NFS Version 3 Protocol Specification*, June 1995.

[Christ97a] N. Christenson, D. Beckemeyer, T. Baker, A Scalable News Architecture on a Single Spool, *;login:* vol. 22 (1997), no. 3, pp. 41–45.

[Christ97b] N. Christenson, J. Larson, Work in progress.

[Grubb96] M. Grubb, How to Get There From Here: Scaling the Enterprise–Wide Mail Infrastructure, *Proceedings of the Tenth USENIX Systems Administration Conference (LISA '96)*, Chicago, IL, 1996, pp. 131–138.

[Hitz94] D. Hitz, J. Lau, M. Malcom, File System Design for an NFS File Server Appliance, *Proceedings of the 1994 Winter USENIX*, San Francisco, CA, 1994, pp. 235–246.

[Hitz95] D. Hitz, An NFS File Server Appliance, `http://www.netapp.com/technology/level3/3001.html`.

[Postel82] J. Postel, *RFC 821, Simple Mail Transfer Protocol*, August 1982.

[Rigney97] C. Rigney, A. Rubens, W. Simpson, S. Willens, *RFC 2058, Remote Authentication Dial In User Service (RADIUS)*, January 1997.

[Seltze91] M. Seltzer, O. Yigit, A New Hashing Package for UNIX, *Proceedings of the 1991 Winter USENIX*, Dallas, TX, 1991.

[Stern91] H. Stern, *Managing NFS and NIS*, O'Reilly & Associates, Inc., Sebastopol, CA, 1991, chapter 12.

[Zeeman77] E. Zeeman, *Catastrophe Theory, Selected Papers 1972-1977*, Addison–Wesley Publishing Company, Inc., Reading, MA, 1977, p. ix.

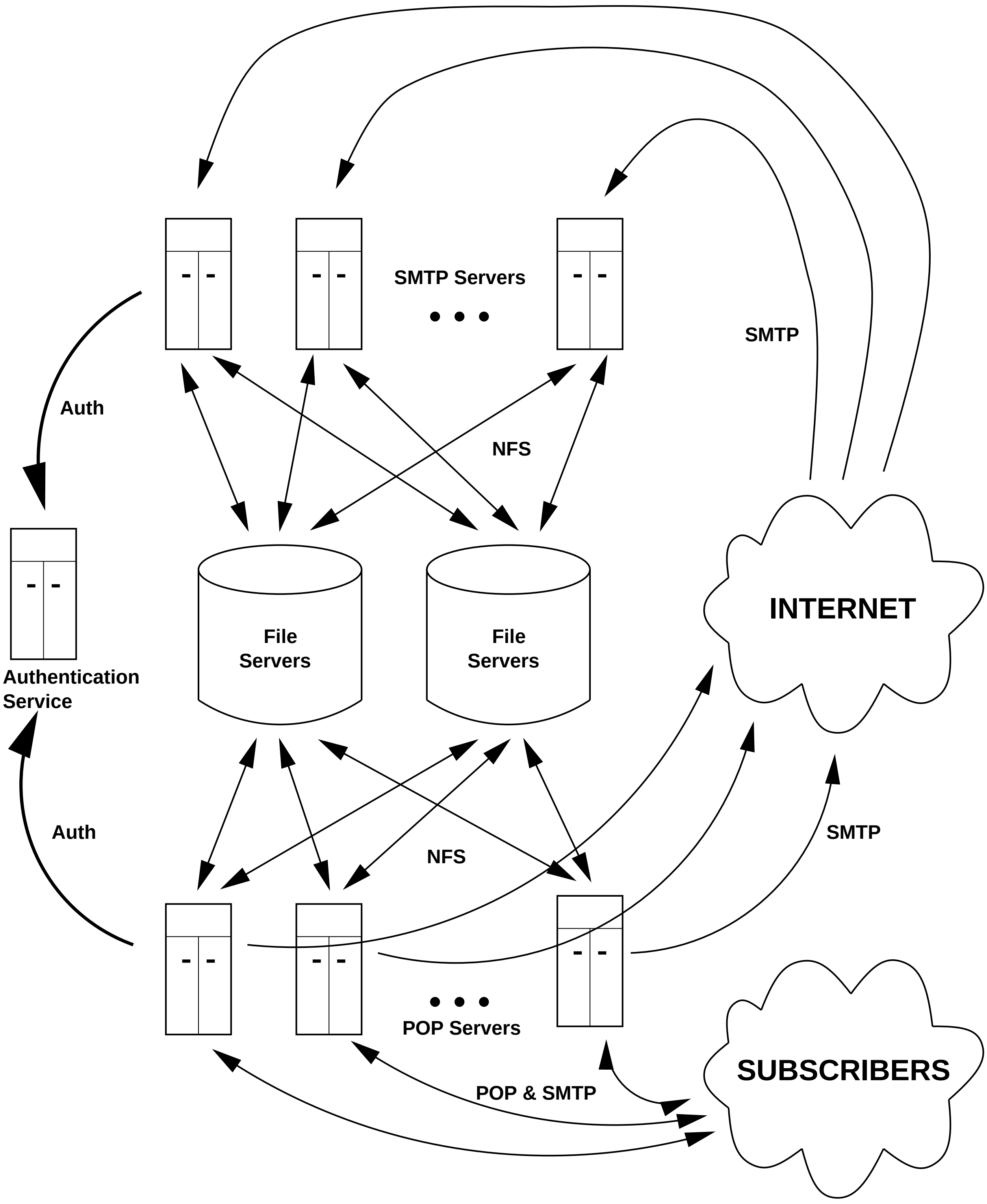
SMTP Servers
• • •
SMTP
Auth
NFS
Authentication Service
File Servers
File Servers
INTERNET
Auth
NFS
SMTP
• • •
POP Servers
SUBSCRIBERS
POP & SMTP

Figure 1