Architectural Issues In the Andrew Message System

Nathaniel S. Borenstein
Craig F. Everhart
Jonathan Rosenberg
Adam Stoller
Information Technology Center
Carnegie Mellon University

Submitted to IFIP 6.5 Working Conference on
Message Handling and Distributed Application Systems

Abstract

The Andrew Message System is one of the most ambitious and technically successful systems yet built in the area of electronic communication. In this paper, the authors of the system explain the key decisions in the architecture and reflect on what was done right and what was clearly wrong. Implementation details are mentioned only in passing, in order to maximize the relevance of this paper for the designers of successor systems.

Introduction

The Andrew Message System (AMS) was an ambitious project to build a prototype of the electronic mail and bulletin board systems of the future. Because it was a research-flavored development project, and not an attempt to define a standard for such systems in the future, it was inevitable that certain decisions that were made in the AMS should be reconsidered in successor systems, and particularly in any attempt to standardize such systems.

In this paper, we will briefly describe the Andrew Message System for readers not familiar with it. We will then examine the key architectural decisions that were made in the construction of the system. These are the most basic decisions, the ones most difficult to change once the system is built. We will discuss, in particular, which things seemed to work out well and which we would do differently in a future system.

Background: Andrew & Its Message System

The Andrew Project [1, 2] is a collaborative effort of IBM and Carnegie Mellon University. The goal of the Andrew project was to provide a good environment for university computing. That is, particular emphasis was paid to the needs of the academic and research communities.

As the project evolved, it concentrated on three main parts. The Andrew File System (sometimes referred to as "Vice") [3,4] is a distributed network file system designed to provide the illusion of a uniform central UNIX file system to a large number of users (10,000 workstations was the design goal). The Andrew Toolkit (sometimes referred to as "BE2") [5] is a window-system-independent programming library that supports the development of user interface software. It currently supports a wide range of applications, including a multi-media editor that allows seamless editing of text, various kinds of graphics, and animations.

The third main piece of Andrew is the Andrew Message System, or AMS. The AMS heavily utilizes the file system and the toolkit to provide a large-scale mail and bulletin board system allowing communication that seamlessly includes text, pictures, animations, spreadsheets, equations, and hierarchical drawings. The AMS also supports "old-fashioned" text-only communication with low-end machines such as IBM PC's and with the rest of the electronic mail world. The Andrew Message System has only recently become widely available; the "results" discussed in this paper are really observations of the first large test installation, the Carnegie Mellon Campus, where thousands of students, faculty, and staff have been using the system during its development over the last few years.

A detailed description of the Andrew Message System is beyond the scope of this paper, and can be found elsewhere [6, 7]. This paper will concentrate on the factors that shaped the basic AMS system architecutre.

Key System Constraints & Goals

The AMS project began with a semi-formal Requirements Analysis. The result of that analysis was an informal "wish list" for electronic communications [8]. Although some items on that list needed to be abandoned as impractical, we were left with a series of clear goals for our system, which acted as major design constraints:

Reliability of Message Delivery

Distributed mail delivery systems have, in the past, suffered from extraordinary levels of unreliability. Guaranteeing the delivery of a message, and providing users with a firm promise that a message, once sent, would be either delivered or returned (at least for "local" recipients), was from the outset one of our highest priorities.

Support for Enormous Message Databases

A quick glance at existing bulletin board systems such as UNIX Netnews should be enough to convince anyone that the future of electronic communication includes an enormous volume of data. Extrapolating from the netnews example, it is clear that expanding both the number of participants and the types of data (raster images, for example, are notoriously larger than text) can only make matters worse (that is, make the database larger). A forward-looking message system can not afford to ignore the question of scale, and must explicitly plan for databases we might regard as unthinkably large today.

Support for Wide Range of Machine Types

For better or worse, machine and operating system heterogeneity is a reality. Any system that is tied to the graphics capabilities of a specific machine or class of machines is doomed by the market. Nonetheless, any system that constrains its capabilities to the lowest common denominator will be unlikely to excite its potential users. In our case, we took on the twin goals of fully supporting the graphics capabilities of scientific workstations, while compatibly supporting simpler machines like the IBM PC.

Support for Wide Range of Objects

Given the goal of supporting scientific workstations, we wanted to be able to support a wide range of objects that can be represented on such machines. In particular, we felt it essential to support formatted text, raster images, and line drawings, and we also set as a goal developing an architecture that would make it easy to add new objects (such as music) to the supported set in the future.

Support for Wide Range of Interface Styles

Observation has shown that there are basic differences of individual taste and inclinations that lead to fundamentally different ideas about user interfaces. Moreover some user interfaces, though desirable in the context of a large graphics display on a workstations, may not be possible in other situations, such as a dialup line. For these reasons, it seemed essential that the architecture make it as easy as possible to develop alternative user interfaces.

Support for Active Messages

In most existing systems, a message is a fairly passive object. Once a user receives a mail message, for example, there is not really anything the message can do but allow itself to be read. The user can do a few things to it, such as read it, delete it, and copy it, but the message is not able to take the initiative and do anything itself. We had come up with several cases in which more active messages would be desirable. For example, a message calling for a vote could simply ask the user to choose from the specified alternatives, rather than make him compose his vote answer by hand. Therefore it seemed important that our design be able to handle such active messages in a manner relatively independent of the user interface programs.

The Andrew System Architecture

While all of the above considerations were forming themselves in our minds, the other groups in the Andrew project were proceeding independently. They developed two key components, the Andrew File System and the Andrew Toolkit, which fit in extremely well with our own needs as outlined above.

The Andrew File System

The Andrew File System is a large-scale distributed file system, specifically designed to function in a very large-scale manner. It achieves key efficiencies using whole-file-transfer, substantial caching on the local disks, and callbacks to notify the workstation when the authoritative copies of cached files have changed. The fact that it functions well at a large site was something we could make good use of. However, the whole-file-transfer had serious implications for our databases, which we had to break up into many small files for performance reasons. Additionally, the distributed nature of the file system required unusually careful attention to be paid to failure conditions from file operations previously thought to be "reliable".

The Andrew Toolkit Datastream

Much of the design of the Andrew Toolkit focused on the representation of cooperating independent objects; a text object can include a table object, for example, and a table object can include within itself another text object, and so on. Working with the Toolkit developers, we were able to ensure that the external datastream -- that is, the form in which such nested objects were represented in files -- could be transmitted through existing mail transport mechanisms (notably SMTP and its variants). Thus, the most spectacular feature of the Andrew Message System, its ability to handle a wide range of graphical objects, came almost "for free" from the use of the Andrew Toolkit. However, an implication of this use was that all mail files generated by the system would use this data format, requiring that we introduce translators into several parts of the architecture.

The AMS System Architecture and Rationale

Given the Andrew Toolkit and File System, which were basically what we wanted and were at any rate in large part out of our control, there were still a large number of important architectural decisions to be made.

The Server/Client Split

One basic decision that must be made in any system like this is whether data sharing is to be achieved via a server-based mechanism or a shared data space, such as a shared file system. Because we could begin by using the Andrew File System, one might have expected us to choose the latter approach. However, this use of the file system struck as as potentially too limiting. In particular, low-end machines such as Macintoshes did not have direct access to the file system (although such access is being developed), and we did not want the Andrew Message System to be tied entirely to the Andrew File System. For example, a site with only a few machines might not bother bringing up a central file system, but might still desire centralized mail and bulletin board service.

For this reason, we added a second level of indirection. Although the message database itself does indeed reside in a file system -- either the local UNIX file system, the Andrew File System, or some other file system -- client programs are not expected to access that database directly. Instead, all operations on the database go through a program called the *message server*, which communicates with client (user interface) programs using remote procedure calls. Thus, all message server processes must run on machines with access to the database in the file system, but client programs are under no such constraints. The existence of two levels of indirection in our system (from the client to the message server, and from the message server to the central file system) has worked out extremely well, as described below.

Server Statelessness and Idempotency

Given that the server may be running on a different machine than the clients, questions of optimizations quickly arise. One scenario we particularly wanted to support was to run message server processes on idle machines. By some unspecified mechanism, client programs would find underutilized machines on which to run their message server processes. However, idle machines can be reclaimed, and so we wanted to make it possible for one message server to quit and be replaced by another message server, possibly on another machine, with no negative consequences for the client aside from a delay during the changeover (and possible later slowdowns due to loss of caching optimizations).

The desire to support this kind of transparent server migration led us to specify that the client-message server relationship would be completely stateless; that is, each call to the server could be issued and responded to independently of the history of previous calls.

Another problem that always arises in client-server relationships involves the coordination of retried calls. If the client thinks that the server has timed out, and sends a call again, it does not necessarily want the server to repeat the operation a second time. Given that we wanted an extremely simple remote procedure call mechanism (to make it easier to port the system to a wide variety of client machines), we did not want to count on the RPC mechanism to avoid all such call duplications. Instead, we opted to specify that all remote procedure calls in the system must be idempotent, i.e. they must be repeatable without harmful side-effects.

The SNAP RPC Mechanism

Given the constraints described above, we chose to design a very small, simple remote procedure call mechanism for use by the AMS. SNAP (Simple Network Access Protocol), as our RPC package is known, handles the reliable fragmentation, transmission, and reassembly of RPC requests and responses between the client and the server in a portable manner. Its view of an RPC request is simply a buffer of bytes to be transmitted and a buffer of bytes to be returned. Thus a set of packing/unpacking routines are provided by the application program in order to complete the encapsulation of the network questions and answers behind what appears to be a simple procedure call on the client end.

The Guardian

How, in a system like this, does a client find a server? Many mechanisms are possible, but none is ideal. The normal UNIX "services" mechanism doesn't answer the question satisfactorily for several reasons. First, in order to satisfy the protection requirements of the Andrew File System, it seemed necessary that each server be authenticated as a single user. Thus, while multiple clients can talk to the same messagserver process, they can do so only as long as they are each authenticated as the same user. Given, then, that there will be an aribtrary number of message servers running on a given machine, how does a client find the right address? Even worse, if there is no message server running, how does a client on a remote machine (particularly a non-UNIX machine) start one?

In order to answer these questions, we created a special system program called the guardian. The guardian runs as root, and is started whenever the machine reboots. It is a very small program that does only a few things. It takes client requests to be connected to servers (the message server is one of two servers it currently knows about, but this is easily and dynamically reconfigured), checks the client's authentication (using either the raw password or the Andrew File System authentication tokens), starts new authenticated servers when necessary, and tells the clients where to find their servers (which may be on another machine). The guardian is easily configured to determine what users may start servers on a given machine, whether that machine will act as a server machine for remote clients, and so on. Moreover, if the "migrating server" mechanism mentioned above is ever implemented, it will be straightforward for the guardian to connect clients with servers on other machines.

The CUI Library

On the client's side of the client/server boundary, an additional layer is provided between the remote procedure calls to the server and the client interface itself. This layer is called the CUI (Common User Interface) Library. In addition to encapsulating and hiding the server calls, the CUI Library improves the overall efficiency of the system by providing yet another level of caching, so that certain kinds of questions need go to the server only once, with the answers cached afterward by the library. Additionally, the CUI library provides a few abstractions of context (recall that the client/server interface is stateless) and relatively transparent handling of server timeouts and reconnections.

Beyond the functions described above, however, the CUI library tries to do a few extra things to simplify the task of constructing a new user interface to the system. By packaging up common operations into library units, the CUI library further simplifies the use of the AMS by an interface program. However, such packaging, it turns out, is not without cost. In the process of exectuing certain higher-level CUI library calls, user interaction is necessary. The CUI library stipulates, therefore, that every client program must provide a few simple routines for user interaction, such as "GetBooleanFromUser", "GetStringFromUser", and "ChooseFromList" (a multiple choice question). Each client may implement these routines however it wishes, but must provide routines to allow the answering of such questions.

For example, the way that support for "active messages" is provided in an interface-independent manner is entirely by the CUI library. Each interface is encouraged -- though not required -- to call a single CUI library routine each time it displays a new message to the user. This routine checks for all of the different kinds of active messages, and handles them via the interface-provided routines. Thus, for example, if the message calls for a vote, the CUI library will notice this and will call the interface's "ChooseFromList" procedure.

The Delivery System

Another key decision in the AMS was to entirely separate the mail transport mechanism, not merely from the user agent (the CUI library client) but also from the database agent (the message server process). In this sense, the parts of the AMS described thus far are delivery-system-independent. For example, they run transparently on top of the standard UNIX sendmail-based delivery system, where that system can be made to function effectively.

Unfortunately, that system can not always be made to function effectively. In particular, the standard sendmail tools predate the existence of a central file system, and function very badly in such an environment. For that reason, a rewritten mail delivery system became another major component of the AMS. (Actually, there were other problems with sendmail as well; running sendmail daemons on hundreds of workstations each with their own local disk queue proved to be a maintenance nightmare even without considering the effects of a central file system.)

The basic operation of mail delivery in the AMS Delivery System is pictured in Figure 1. When mail is sent to the user "nsb", the message is written as a new file in the "Mailbox" subdirectory of nsb's home directory. This simplicity, however, belies the difficulty of keeping delivery reliable. When all or part of the central file system is unavailable, it may not be possible to write into nsb's Mailbox. If this occurs, the mail must be queued into a mail queue directory. Such directories are scattered throughout the central file system on mutliple file servers, to ensure reliability. If all of them are unavailable (a rare event, but occurring in the event, for example, of a total network outage), the message is written into a queue of last resort on the local disk.

Once mail is in a queue, it can take one of the following paths to get out; a daemon on the local machine will eventually empty the local queue into one of the central file system queues. Other daemons, running as the distinguished user "postman" on post office machines, will eventually read and deliver mail from the file system queues. Mail destined for non-local recipients is simply written into the file system queues at the initial stage of the delivery process, and is then passed off to the network mail delivery mechanisms by the post office daemons.

Actually, though, the situation is even more complicated. Because mail delivery to 15 people involves writing mail into 15 Mailboxes, it was found to be unacceptable to make users sending mail wait for this process to terminate. Thus the notion of "dropoff" was introduced. When mail is sent, it is not written directly into mailboxes, but is simply queued in the ".Outgoing" subdirectory of the sender’s home directory. Once the items are written into this special queue directory, a special message is sent to the local machine’s mail delivery daemon, telling it to deliver the message when it can. Delivery thus proceeds in parallel with the user’s other actions, but delivery has already been guaranteed by the act of enqueueing the message in the .Outgoing directory in the central file system. Even if the local daemon dies or the machine catches fire, a post office daemon will eventually check the user’s .Outgoing directory and pick up the queued mail.

With all of these daemons picking up queued messages, one might ask why the local component is necessary at all. The answer is that the queueing is for the most part backup to provide reliability; in most cases, all of the work of delivery is done on the workstation itself, reducing the need for post office machines. The post office machines and delivery daemons function as the deliverers of last resort for local mail, and the primary delivery mechanism for off-site mail.

The Unformatted Version of Messages

Given the wide variety of possible client interfaces, it is clear that not all of them will be able to understand multi-media messages in all their splendor. For this reason, a very simple and portable routine was written, completely independent of the Andrew Toolkit, for translating the toolkit’s datastream into a "lowest common denominator" format. Thus, for example, this translator would replace an animation by a sentence like "An animation appeared here, but could not be displayed."

This procedure has found its way into the system at several points. Clients requesting the body of a new message from the message server may request either the formatted or the unformatted version, for example. The message delivery system, too, uses this "unformatting" procedure when sending the message to non-local sites that do not understand the multi-media format.

The White Pages

The White Pages is a database package for mapping names to mail addresses. It can find and report ambiguities (two John Smiths) and can suggest corrections for many misspellings, based on phonetic matching and some heuristic knowledge (about nicknames, for example).

The white pages is accessed by a separate subroutine library, used by several pieces of the system. The delivery system uses the white pages to resolve the ambiguities of delivery addresses. The message server uses the white pages to provide an exported procedure call by which client programs can validate lists of addresses before actually sending mail. Since client programs may not have access to the actual White Pages database, they can not query that database directly (that is, they cannot themselves be white pages client programs) but can obtain the necessary functionality via calls to the messagserver, which *is* a white pages client.

Assessment of AMS Architectural Decisions

The purpose of any architectural review of an implemented system is to provide some guidance or advice to implementers of future, similar systems. In this case, there is a supplemental purpose, which is to assess the utility of the two major new tools used in building the system, namely the Andrew Toolkit and the Andrew File System.

Use of The Andrew File System and Toolkit

It is frankly hard to be anything but positive about the Andrew Toolkit and the Andrew File System. The Toolkit gave us multi-media capabilities essentially for free; using a toolkit-based approach in this regard is clearly the right thing to do, as it provides a high level of consistency across applications. The biggest thing the Andrew Toolkit gave us was the encapsulation of nested multi-media objects both on screen and in a permanent (i.e. file-based) datastream. In addition, the Toolkit gave us window manager-independence; the Messages application already runs virtually identically on two different window managers, with more planned. Similarly, the use of some sort of central file system seems essential for the management of a large volume of data.

A good question to be asked is whether or not the Andrew File System and Toolkit offered significant advantages over other available file systems and toolkits. In the case of a UNIX-based implementation, this really comes down to asking how they compare to the Sun NFS file system and the DEC X Toolkit. The comparison is in both cases quite easy to make. NFS was simply not designed to support nearly as many workstations as the Andrew File System. Although it may perform better for small numbers of workstations, it simply could not handle the load our system would put on it at an installation like the Carnegie Mellon campus as well as the Andrew File System. In addition, the Andrew File System offers the significant advantages of rigorous authentication and disk caching for better performance.

The comparison with the X Toolkit is even more straightforward. While the X Toolkit does offer a very simple model of the window, this simplicity goes much too far; it is, in fact, too simple to allow for co-operating nested objects (for example, our message bodies can include text that includes tables that include text that include raster images and equations) and has, apparently, absolutely no notion of a datastream, and hence no toolkit-level support for permanent storage (or mail transmission) of multi-media objects. In short, the largest part of what we got from the Andrew Toolkit would have had to have been to be re-invented if the X Toolkit were used.

Things We Got Right in the AMS

There were quite a few decisions made in our architecture which, in retrospect, appear to have been right on target. In fact, if we were to do it all over again, the highest level of the architecture would probably be identical to what they are now.

Particularly good was the clean split between four components: the client program, the message server program, the delivery system, and the file system. The ability to replace any of these components relatively cleanly has left us with a very open system that may go in several directions in the future. In particular, our system already can run with two different delivery systems and two different file systems, with one more of each likely to be added in the near future. Moreover, we could relatively easily replace the entire message server, perhaps to use a new database technology, with essentially no changes necessary to any of the user interface programs.

We are especially pleased at having two levels of indirection between the client and the data. It seems likely that our architecture could scale well to a far larger installation -- tens of thousands of workstations, perhaps -- by clustering message server processes on a relatively small number of workstations, all sharing a central file system, and having the client workstations talk only to these servers, not to the central file system. Indeed, such a pyramid architecture is probably worthy of further generalization in future systems.

The separation of the white pages component was also a very good decision, primarily because a very general and powerful user name lookup facility turns out to have many other uses beyond the message system. Within the white pages, the use of a b-tree representation [9] (storing the information in a large number of small files) has given us better database performance than we ever really expected on a file system based on whole file transfer.

Problems With the Client/Server Split

Although the basic split between the client and the message server was, we believe, a good decision, a future effort could improve some of the details substantially.

First of all, the set of routines exported by the server to the client was not designed sufficiently carefully. While most of these routines are of the kind that would be needed in any such system, a careful design would undoubtedly lead to a more coherent set.

The SNAP library for remote procedure calls was designed to be extremely simple, portable, and cheap, especially on the client end. This had its costs, naturally. First of all, SNAP limits transmissions to fixed-size buffers. Although the buffers could be quite large, there are still several places where the code is much less natural than it would be if the RPC mechanism could simulate the return of arbitrary-sized information. In addition, the SNAP architecture is essentially unidirectional, in that the client can call the server but the server can not call the client. This can be particularly painful in situations involving unanticipated errors, where we would really like the server to be able to tell the client to report the problem to the user. Instead, we have to try to make the error codes returned by the server to the client (currently 4 8-bit codes) sufficiently meaningful, which can be very hard to do. This problem could be alleviated without sacrificing the unidirectional architecture by the creation of a distinguished error condition that meant, "ignore the return data and just treat the return packet as an error string," but this would also have serious implications for the structure of the unpacking routines. In another implementation, we would at least reconsider the unidirectional communication.

The requirement that all server calls be idempotent, as described above, appears now to have been overly restrictive. Although this requirement exists largely in order to keep the SNAP level simple, it has led to some complexities and ugliness at the next level higher. Although the RPC layer numbers its calls and keeps track of the results so as to simply prevent the same call from going through twice, it does not handle reconnection to a server itself, and thus cannot tell when a client has reconnected and is retrying the same call (which may, in fact, already have succeeded). Providing more support for reconnection to servers at the RPC level might allow future systems to avoid the idempotency requirement, which would simplify the design of the client-server interface remarkably.

In general, the attempt to keep the client library simple has made the client-server interface uglier. For example, we wrote a rather complicated address parsing library, to parse all of the odd addresses used by various network mailers. In order to keep this complexity out of the client library (and in order to avoid having to think about porting it to low-level machines), we decided that all such parsing would be done by the server. This left us in a situation where a client had a list of addresses, each of which needed to be validated (verified) by the server. However, questions might need to be asked about each individual address, for example if one is ambiguous. Since only the server knows how to separate the addresses, and only the client knows how to query the user, the resulting protocol is strikingly complicated, and the server call related to name validation is harder to explain than any other call in the libraries. It would be very nice if, in a future system, a bit more power on the client end could be assumed.

Problems With the Required Interface Functions in the CUI Library

As discussed above, the CUI Library requires its clients to provide a few basic routines for interaction with the user (e.g., GetBooleanFromUser). Although these seemed simple, harmless, and universal at first glance, they in fact turned out to be surprisingly challenging for certain interfaces. For example, Batmail, the Emacs interface, was implemented largely in mocklisp, the Emacs extension language. The communication with the CUI Library and message server process was done via a C subprogram, known as Robin. Unfortunately, Robin, as the CUI Library client, had to provide the basic interaction routines, so that the Batmail/Robin interface was rendered considerably more complex as batmail had to be ready to answer questions from Robin essentially asynchronously.

Nonetheless, the occasional pain seemed worth the cost. However, a more careful separation of the various parts of the library would make it possible to only include the low-level parts of the library if so desired. In this way, some programs could supply the required interfaces and take advantage of the higher-level library routines, but programs for which this would be problematic, such as batmail, could use only the lower-level routines, duplicate the higher-level functionality, and never need to grapple at all with the implementation of the interaction routines. However, the truth is that nearly any interface should be able to provide such routines, and the basic Emacs/subprocess mechanism that has trouble with this paradigm is itself an ugly (although very widespread) hack.

Problems of Insufficient Generality

One of the major goals of the AMS was to generalize features that had been present in previous systems in more limited forms. In at least a few cases, however, the AMS introduced new features which were only later recognized as worthy of generalization. In these cases, future systems could introduce substantial improvement through generalization.

In particular, the notion of "active messages" was arrived at very late, on the heels of several specific types of active messages. We have not yet stepped back and seriously considered the implications of this idea, but it seems clear that there are many more uses for this concept than the four we have implemented. In a future system, a much wider range of active message types should be supported. (It should be noted that it would be very easy, using the Toolkit, to write a generalized "active message" package, but that this package would not be useful in the lower-functionality AMS application programs. Therefore, it seems necessary to have an interface-independent definition of a general "active message" type.)

Another problematic area is customization. Although we recognized from the start that customization is essential in such a large and complex system, we were not sufficiently aware of the distributed nature of the customization requirements. In particular, each interface will have some customizable parameters. The CUI library and message server each have their own set of customizable parameters. The Toolkit itself provides another, fairly baroque extension mechanism. An older, "standardized" customization mechanism for Andrew is still used by certain library routines on which the Andrew Message System depends. Finally, the language in which new messages can be automatically classified as they arrive is itself a fairly complicated extension mechanism. The net effect is extremely confusing, especially to new users, who have a hard time figuring out in which of five or six places to seek the customization option he desires.

A future system should at least strive to provide a single method by which all of these components can be customized. It should be remembered, in any such effort, that some applications will need extremely complex customizations -- e.g. LISP-like programs -- while others will require simple yes/no options. If a general customization library can not support the former powerfully and the latter simply, alternative customization mechanisms will inevitably creep in, as they did in our system. Once they do, they will be nearly impossible to phase out. Much more attention to customization and extension should be provided from the very beginning of future message systems.

Other Futuristic Ideas

As the AMS has grown, a few new ideas have crept into our thinking, too late to be reflected in the design of the system itself.

One such idea is to further partition functionality into specialized servers. For example, the white pages technology could be reimplemented as a query-to-server mechanism, rather than a database lookup mechanism. Such partitioning could significantly reduce the size of various application programs, and improve their performance as well. To implement this properly, a more generalize service mechanism would be necessary than was available to us. Such mechanisms are beginning to appear now, the most notable example being the Athena Service Management System [10].

Another promising idea for future systems is to entirely abandon RPC as the fundamental communication paradigm, using instead a full-blown programming language for communication with the server. Such an approach has been used with great success in the NeWS window system [11], and is especially appealing for systems like the AMS given that it proved highly desirable to provide a full language in the server as an extension mechanism. If such a language needs to be provided anyway, it is very appealing to use the language as the fundamental method of communication between client and server.

Conclusions and Future Work

Although work continues on the Andrew Message System, it is a maturing technology with a large and growing number of people depending on its continued operation. Therefore it seems likely that, with only a few exceptions, future development will focus on enhancing its portability and reliability.

However, the AMS is obviously not the last word in message systems, although we believe it has become a new "high-water mark" in such systems, and a reasonable object of comparison for future systems. The AMS was built in a few years by a few people, during a time in human history when electronic mail was taken seriously by a relatively small number of people. We believe it is likely that electronic message systems will soon become one of the most crucial technologies in the organization of human societies, at least as important as basic telephone communication. As this happens, it seems inevitable that systems like ours will be replaced by highly engineered systems that represent thousands, rather than merely tens, of man-years of effort. That such systems will operate more efficiently and reliably than ours, and that they will be easier to use, should go without saying. It is our hope that this candid assessment of the virtues and shortcomings of our system will prove of use to those who build such systems in the future.

Acknowledgments

The Andrew Message System was designed and implemented by Nathaniel Borenstein, Craig Everhart, and Jonathan Rosenberg. Substantial additional work was done by Adam Stoller, Mark Chance, Mike Sclafani, Aaron Wohl, Sue Pawlowski, and Bob Glickstein. In addition we wish to acknowledge the dozens of people who were involved in the many facets of the Andrew project, and regret that we cannot name them all. We would, however, be remiss if we did not acknowledge the unique vision of Jim Morris, without whom Andrew as we know it could never have become a reality.

References

[1] Morris, et al., "Andrew: A Distributed Personal Computing Environment", C*ommunications of the ACM*, March, 1986.

[2] Morris, James H., "'Make or Take' Decisions in Andrew", *Proceedings of the USENIX Technical Conference*, February, 1988.

[3] Howard, John. H., "An Overview of the Andrew File System", *Proceedings of the USENIX Technical Conference*, February, 1988.

[4] Kazar, Michael Leon, "Synchronization and Caching Issues in the Andrew File System", *Proceedings of the USENIX Technical Conference*, February, 1988.

[5] Palay, et al., "The Andrew Toolkit: an Overview", *Proceedings of the USENIX Technical Conference*, February, 1988.

[6] Rosenberg, et al., "An Overview of the Andrew Message System", *Proceedings of SIGCOMM '87 Workshop*, Frontiers in Computer Communications Technology, Stowe, Vermont, August, 1987.

[7] Borenstein, et al., "A Multi-media Message System for Andrew", *Proceedings of the USENIX Technical Conference*, February, 1988.

[8] Borenstein, N., and J. Rosenberg, "Electronic Communications Wish List II", published via various Netnews groups and Internet mailing lists, Fall, 1985.

[9] Lehman, Philip L., and S. Bing Yao, Efficient Locking for Concurrent Operations on B-Trees, ACM Transactions on Database Systems, Volume 6, Number 4, December, 1981, pp 650-670.

[10] Rosenstein, et al., "The Athena Service Management System", *Proceedings of the USENIX Technical Conference*, February, 1988.

[11] NeWS Manual, Sun Microsystems, Inc., March, 1987.

Figure 1: The AMS Delivery System