Nick Shipman

Developing Internet Software: AltaVista Mail

The emergence of the Internet as a place where people can conduct business prompted DIGITAL to investigate the development of products specifically for use in this environment. Electronic messaging systems based on Internet technologies provide the communication medium for many businesses today. The development of AltaVista Mail illustrates many of the concerns facing engineers who are designing products for this new customer base. The results of our experience can be helpful in many ways and should be of interest to those involved in designing technologies for running Internet applications.

In late 1993, the Mail Interchange Group (MIG) within DIGITAL started the AltaVista Mail development program. At that time, the members of MIG had substantial experience in the development of electronic mail (e-mail) technologies; however, the new products were being targeted for use on the Internet in an environment that was quite different from the one for their previous products. In an effort to satisfy the new customer base, the members of MIG reexamined their design and development process.

The AltaVista Mail product emerged from efforts to improve MIG's support for Internet-based e-mail technologies. Our previous products were electronic mail and directory servers for network backbone use, based on the most recent X.400 and X.500 standards. Products suitable for the Internet environment would clearly be quite different.

This paper begins by presenting our analysis of Internet services and support software and describing the transmission of e-mail on the Internet. The paper then discusses the implications of developing a product for the Internet environment and explains the impact of those implications on the design and implementation decisions that defined the AltaVista Mail product. The paper concludes with the engineering assumptions and habits that had to be overturned to build the product set.

Internet Services and Software

During our initial analysis of the product possibilities, we made several interesting observations about Internet services and software, particularly in comparison to the mission-critical products in MIG's existing portfolio. (Our observations could more accurately be called assertions—it was and is remarkably difficult to get hard information about Internet use.)

1. The academic/research/technical community determined the nature of the Internet's service offerings. Most of the software defining the Internet's services was generated by and for this community.

2. The real market opportunity would not be among the academic/research/technical community but would be drawn from ordinary businesses.
3. Much of the service software on the Internet was unsatisfactory for routine business use, either because it was unreliable or because it was difficult for end users to deal with it. Even though free software was abundant, much of it did not work. Support for the free software was a risk: some software had excellent peer support; unfortunately, not all end users were aware of its existence or were able to access it.
4. We judged the operating system platforms commonly used for service software to be unsuitable for a large part of the business community. The various UNIX platforms need skilled local staff; corrupted or poorly configured Windows version 3.1 and Macintosh run-time environments would be difficult to diagnose and expensive to support.

Expanding into support of the Internet environment would require us to build native equivalents for some of our existing server software. We believed that the Windows NT platform offered a good framework for systems that would work well in any business environment and be easy to support.

Initially, we required the following products: a server that supported the Simple Mail Transfer Protocol (SMTP) and the Post Office Protocol version 3 (POP3); a gateway to Lotus cc:Mail post offices; and a gateway to Microsoft Mail post offices.

The SMTP/POP3 server is the mail system component responsible for accepting messages from mail client programs. It transmits them toward the recipients' SMTP servers and performs local delivery using the POP3 protocol. This process is described in more detail in the next section.

E-mail on the Internet

This section briefly describes how Internet e-mail is transferred from the originator to the recipient.

The originator's mail client program constructs a message according to the rules described in the Internet standard, RFC 822.[1] The RFC 822 standard defines a message as a sequence of short lines of 7-bit ASCII text, each terminated by a carriage-return and line-feed sequence (CRLF). The first lines are header fields; these are extensible but typically include the originator and recipient e-mail addresses, the date, and the message subject. The header ends with a blank line, and the remaining lines constitute the body of the message. Figure 1 shows an SMTP dialogue that includes an RFC 822 message.

Where appropriate, the mail program can also follow the Multipurpose Internet Mail Extensions (MIME) rules in RFC 1521 and RFC 1522[2,3]; these describe

SMTP command/*response*	Comments
	Caller opens connection
220 server1.altavista.co.uk AltaVista Mail V1.0/1.0 BL22 SMTP ready	Server's welcome message
helo client1.altavista.co.uk	Client gives its own host name
250 OK	Host name was acceptable
mail from:<Fred@altavista.co.uk>	Identifies return path for nondelivery reports
250 OK	Return path was acceptable
rcpt to:<Bill@altavista.co.uk>	A recipient for this message
250 OK	Recipient was acceptable
data	Message follows
354 Start mail input; end with <CRLF>.<CRLF>	OK to start message header
Date: Mon, 7 Jul 1997 08:30:13 +0100	Message's date
From: Fred <Fred@altavista.co.uk>	Originator field
To: Bill <Bill@altavista.co.uk>	Recipient field
Subject: Example message	Subject field
	Blank line ends message-header fields
Hi Bill,	Content lines…
This is a test message.	
It's not very long.	
Fred	
.	…End of content
250 OK	Message has been accepted
quit	No more messages; signing off
221 redsvr.altavista.co.uk closing connection	Finished

Figure 1
Example SMTP Dialogue

how to construct a message body to transfer typed and structured data and how to pass non-ASCII characters in header fields. Figure 2 shows an example of a message constructed according to the MIME standard.

The originator's client submits the message to a nearby SMTP server using the SMTP protocol.[4] This very simple protocol uses short, CRLF-terminated lines of 7-bit ASCII text to transfer its commands and responses. To submit a message, three commands are used: the MAIL, RCPT, and DATA commands introduce the originator's e-mail address, the recipients' e-mail addresses, and the RFC 822 message data, respectively.

The SMTP server examines each recipient's e-mail address to decide where the message should be sent. Recipients are routed by consulting the Domain Name System (DNS), a distributed directory that associates domain names with sets of typed resource records that denote the published properties of each domain.[5–7] For a recipient, user@domain.name, the target domain.name is looked up and the resource records of type MX (for Mail eXchange) are retrieved.[5] These records list the hosts that the domain nominates to receive its mail; each host has a numeric preference value. Eventually, the mail must be delivered to the most preferred host.

For each target domain, the SMTP server uses the SMTP protocol to transfer the message to the domain's most preferred, reachable MX host. (The most preferred host may be unreachable from the local server: it may be switched off for a while, or it may be behind a firewall, a machine that protects a private network by limiting access from the open Internet to the machines inside the protected network.) The chosen host, if it is not the most preferred, forwards the message to a more preferred host and so on, until the message reaches the recipient domain's most preferred host. That host then delivers the message to an area from which the recipient's mail client program can fetch it.

Fetching a message is often a platform-specific operation, but a standard protocol such as POP3 can also be used.[8] This simple, text-based protocol allows the mail client to list the messages waiting to be fetched, to fetch individual messages, and to delete them from the server once they are safely stored within the client.

Newer, more feature-rich protocols and interfaces, such as the Internet Message Access Protocol version

MIME data	Comments
Date: Mon, 7 Jul 1997 08:30:13 +0100	Normal RFC 822 header fields
From: Fred <Fred@altavista.co.uk>	
To: Bill <Bill@altavista.co.uk>	
Subject: Binary attachment	
MIME-version: 1.0	This is a MIME message…
Content-type: multipart/mixed;	
boundary="zzzBoundaryzzz"	… consisting of a list of body parts
	Blank line ends message-header fields
	Start…
--zzzBoundaryzzz	… of first body part…
Content-type: text/plain; charset="us-ascii"	… in ASCII plain text…
Content-Transfer-Encoding: 7bit	… using 7-bit encoding
	Blank line ends body-part-header fields
Hi Bill,	Body-part contents
Here's a binary file.	
It's four bytes of all 1's.	
Fred	
	End…
--zzzBoundaryzzz	… of first body part and start of second…
Content-type: application/octet-stream;	… a stream of bytes called foo.dat…
name="foo.dat"	
Content-Transfer-Encoding: base64	… using base64 encoding
	Blank line ends body-part-header fields
/////w==	Hex FFFFFFFF encoded in base64
	End…
--zzzBoundaryzzz--	… of message

Figure 2
Example MIME Message

4 (IMAP4), do offer certain user advantages; however, they do not perform the basic job of delivering messages any better than POP3.[9] Even though support for IMAP4 was added to AltaVista Mail version 2.0, POP3 remains the method of choice for fetching messages from a remote server: this protocol is so simple that it is hard to implement incorrectly.

Product Design Decisions

The definition of the AltaVista Mail product set did not start with technical issues. Instead, it started with an assumption about the purchase price of a product. Even though the price we chose was not used for the released product, our assumption turned out to be, perhaps, the most useful and powerful design tool available during development.

Product Pricing and Organizational Concerns

We were interested in exploring the implications of offering a product at a very low price and selling in very large quantities to make the business worthwhile. MIG's previous products had been priced at the opposite end of the price scale: they were expensive, but they were valuable to the relatively few customers who needed their functions.

(Interestingly, as we explored the necessary organizational and technical changes involved in moving to the low end, we realized that they would in no way jeopardize our ability to sell at the high end. The organizational changes improve efficiency no matter what the product price, and the technical changes make for a better, more usable product regardless of the customer profile.)

Our starting point was to investigate the engineering implications of building a product that would sell for $100. We concluded the following:

- To maximize the number of products sold, we would have to satisfy the largest imaginable customer base and not exclude a potential customer for any reason.
- Customers attracted to a low purchase price will also require low running costs. No hidden costs could be associated with running the product.
- Support costs would have to be kept to a minimum. If each customer needed telephone support several times over the life of the product, the $100 price would not cover support expenses. We would have to aim at receiving zero support calls.
- We would have to minimize the implementation and maintenance cost and deliver products and updates as early as possible. The Internet market moves quickly, particularly at the low end, and a long development cycle loses sales.

Our existing approach to development involved obtaining agreement from many groups within DIGITAL concerning the nature of a problem area and the architecture of any solutions, and then implementing product versions against the architecture. This process is slow and expensive with considerable management overhead.

For the AltaVista Mail product, we decided instead to direct a small team to generate a product-quality prototype as quickly as possible and to ship that prototype as a product. In the interests of rapid development, we would deliberately discard much of the traditional Phase Review Process but would use regular, informal monitoring to ensure that the prototype remained acceptable to our target customer.

All design and implementation decisions would be judged by their effect on this target customer, not by their adherence to an architecture. All future development would be guided by customer feedback. This method is far less expensive, delivers a product far sooner, and is more likely to reflect current customer needs.

Technological Concerns

Our ideas on product pricing and the design process led to three initial design decisions.

First, nonexpert users must be able to get the full value from the product. Setting up and configuring the product must involve answering the minimum number of questions. Each question must relate to a topic on which the user can reasonably be expected to have an opinion. The user must not be asked questions about the internal operation of the product, only about topics with an external significance.

The product must offer the minimum number of operational controls. (Some high-end customers demand many controls. If necessary, these controls could be added in a later version; but the product must not depend on them, they should not be presented to the average user, and those users who insist on seeing them should be charged a premium to cover the additional support costs. We would explicitly accept that there are certain customers we should not aim to satisfy and certain features we should never offer.)

Second, the product must never go wrong. The product must never encounter any internal errors, only those caused by failures in its operational environment. Any environmental failure must be reported completely and accurately in terms that the user can understand. After a failure has been fixed, the product must start working again with no further intervention. If an environmental failure or an operator intervention corrupted the software, reinstalling the kit must get the system working again. The product must not depend on any product that does not follow these rules.

Third, the product must be inexpensive to build and maintain and must use a rapid development cycle. The product—and each of its components—should deliver the maximum customer-perceived value for the minimum engineering investment. Although the number of features should be minimized, the functions delivered should be sufficient to be useful to a large customer base.

Implementation Decisions

The most important decision was to aim for simplicity above all else: simplicity of design, implementation, and presentation. Simplicity delivers reliability and inexpensive implementation and maintenance. It also helps to ensure that a product is comprehensible to its end user and does not behave in baffling ways, even when it is not working due to an external influence.

A related decision was to use no method or tool that might encourage complexity by helping to manage it: no formal design methods, no automated design verification, and no automated system test. The developer should immediately feel the pain of building a complex structure or one that requires an elaborate system test and thus be encouraged to think again.

Of course, proper engineering practice dictated that we should use a repeatable system test with properties that were well understood, but we deliberately never automated the test. We also used remedial tools such as locally developed libraries to look for memory and handle leaks. (These tools do not tempt developers into bad habits.)

User Interaction with the Server

Ideally, the server should perform its job with no comment, and the user should feel no need to think about the server's performance. Product documentation should be minimized, and there should be no printed documentation at all.

We designed the server to make many of the product's operational decisions, rather than leave the decisions to an administrator:

- The administrator cannot control the routing process. Messages are sent to the targets defined in DNS, and no local rules nominating other targets are supported. The administrator can nominate a firewall but cannot say when to use it (the server uses it automatically, when all the other targets have been found unreachable).
- The operational logs are purged automatically. The administrator can only control how long any logged event is guaranteed to be stored before being purged, and this interval cannot be selected on a per-log or per-event basis.
- The server's network connections are scheduled by the server itself. The administrator can only control the minimum and maximum retry intervals for SMTP connection attempts, not the specific times at which the server tries to communicate.
- The server determines if an event is relevant to system security and responds according to its own rules. Repeated authentication failures result in mailboxes and originating host addresses being locked out for a time; the administrator can manually reset the lockout but cannot control how long it lasts, nor how many failures are judged to be an attack.

Unfortunately, user interfaces cannot be avoided entirely. Therefore the goal must be to minimize the amount of user interaction required with the server and to make user interaction easy to perform and as reassuring as possible to the user. We were able to design the SMTP/POP3 server to be easy to set up and use, and we reduced the user interaction with the gateways to the minimum. Gateways are notoriously difficult to set up; we simplified the process of installation to the extent that a central cc:Mail or MSmail administrator can talk an inexperienced user through the installation.

The SMTP/POP3 server requires an administrator to perform only the following four actions:

- Load the software to a chosen volume
- Tell the server about the local network configuration
- Set up mail accounts (mailboxes) for the local users
- Check that the server is not experiencing problems

The AltaVista Mail product implements these tasks through three user interfaces: the setup (installation) procedure, a Windows-based administration graphical user interface (GUI), and World Wide Web forms.

Setup Procedure. Apart from loading the software, the setup procedure has several other functions. If the software has been corrupted, the setup procedure repairs the service by resetting the server's entire environment to a known state. The server's account privileges are reset, a new password is generated, and the database directories and files have their file protection reset.

Setup continues by asking the minimum number of questions required to allow the server to work. The administrator responds by naming any firewall used, or if in dial-up mode, naming the remote mail server. Finally, it runs the server's self-test, which is described in more detail later in this paper. It is important that self-test run during setup. If the system is not working, the administrator needs to be told the precise causes of the problem immediately, before he or she can become confused by subsequent symptoms of system failure.

Windows-based Administration GUI. The Windows-based administration GUI controls the AltaVista Mail server through a simple, text-based administration

protocol over a Transmission Control Protocol/Internet Protocol (TCP/IP) link. Therefore the GUI can control servers elsewhere in the network.

The GUI has two modes. The default mode is a simple menu with links to functions that invoke the most common administrative tasks: network configuration; adding users and mailing lists; redirecting mail and changing passwords; and self-test. This mode, shown in Figure 3, is a nonthreatening interface for inexperienced administrators; they do not need any other information to use the server successfully. This means that local users with no specialized knowledge can set up small and undemanding sites.

Administrators of large or busy sites need to use the advanced mode, shown in Figure 4. This interface is similar to the style of the Windows Explorer GUI and gives access to every server and mailbox control, the messages in the server, and its operational logs.

Both modes include an on-line help file that gives a brief introduction to the system, reference information on all the visible controls, and assistance in troubleshooting. Apart from the equivalent help text in the World Wide Web forms, this is the only product documentation.

The self-test is one of the most important administration functions for sites at which mail is a mission-critical service. The self-test checks that all aspects of the local machine's environment that are necessary for the server to operate do indeed work; it also checks that the server is responding correctly on all the network ports it serves. In a redundant environment, the self-test checks every element to make sure partial failures are reported. For example, a host generally knows of two or more DNS servers, only one of which needs to be working for the mail server to run. Because the mail server will not see a problem until the last DNS server dies, the self-test must report any partial failure.

The self-test is vital: a regular check is the only way to be sure that a background server is working. A server cannot be guaranteed to inform an administrator of

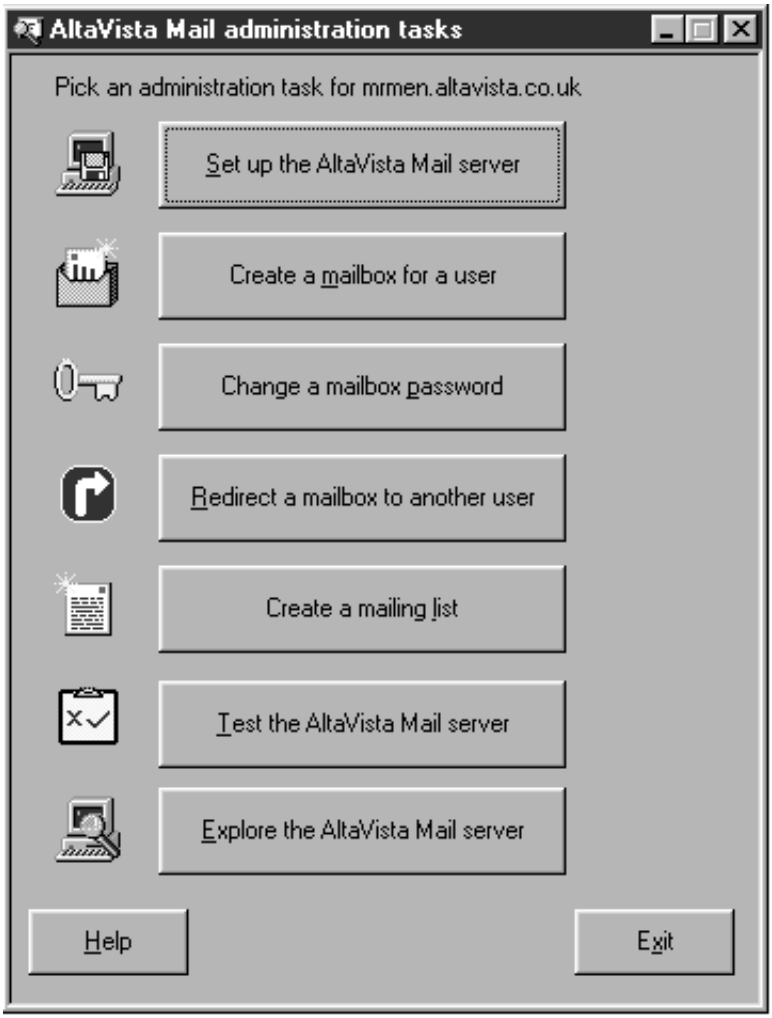

Figure 3
Windows-based Administration GUI, Simple Menu

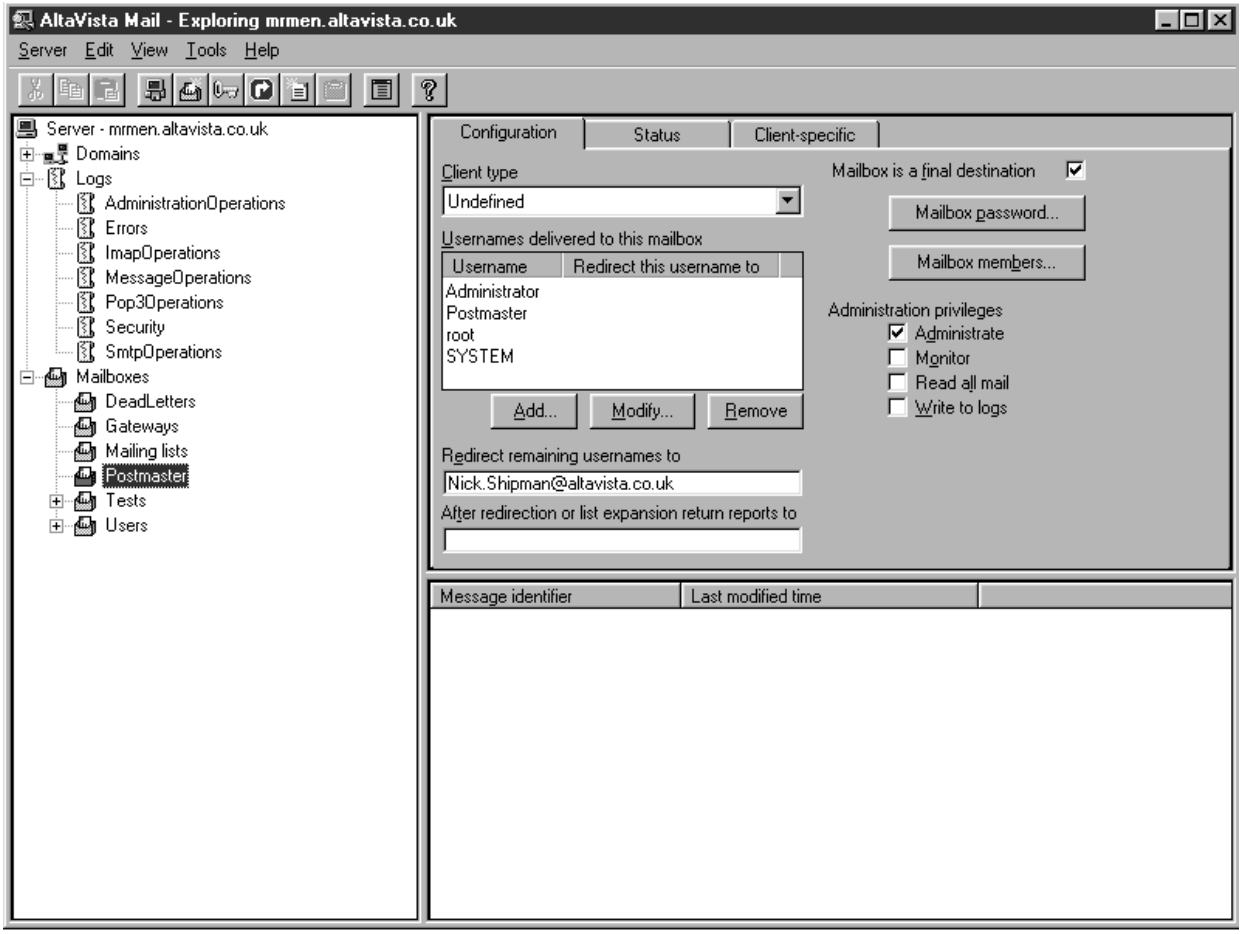

Figure 4
Windows-based Administration GUI, Advanced Mode

problems because the problems may affect the notification path used. The AltaVista Mail self-test allows an administrator to perform the necessary check with a single "button click." The self-test also runs as part of a regular cleanup procedure. Errors are reported to the server's error log, so a less active administrator can monitor the system by reading this log every few days.

World Wide Web Forms. The World Wide Web (WWW) forms interface is provided by a built-in Hypertext Transfer Protocol (HTTP) server. This interface offers the same facilities as the Windows administration GUI. Because a Windows Explorer-style GUI is more difficult to present using a Web browser, we implemented the "power user" options on the top-level task menu of the WWW form. These options make the initial interface somewhat intimidating, because they include controls whose function may not be understood by an inexperienced administrator. The familiar controls, however, are grouped together. The Web GUI is shown in Figure 5.

Several details help make the three user interfaces approachable and nonthreatening.

When the software requires the user to answer a series of questions, it presents a dialogue box chain (sometimes known as a "wizard"). Used properly, this technique allows the user to concentrate on one thing at a time, with all distracting material hidden.

Every question in a dialogue box chain gives an explanation of what information is needed, any suitable defaults or examples, a suggestion of whom to contact to find the answer, and a safe way to abort the process. If the user knows the answer, he or she will be able to recognize it in the example. Users who do not know the answer will not be intimidated by the wording.

The logic works in terms the user understands, not in terms of the software's operation. The gateways, for example, contain a question that the software does not use other than to make the text of the succeeding dialogues relate to the user's environment.

Some controls can easily be invoked in error but cannot be redefined to make the error less likely. In

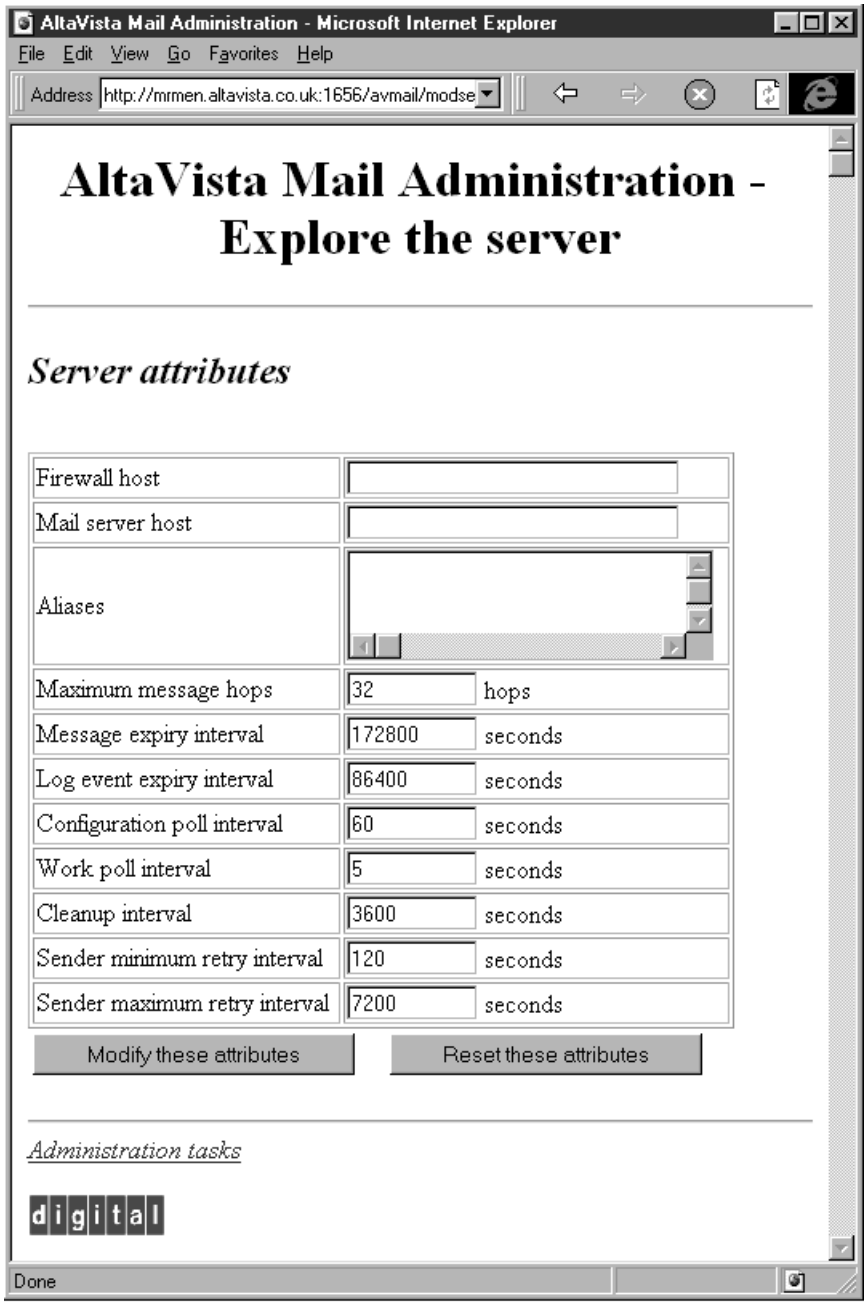

Figure 5
WWW Forms Administration GUI

these cases, the resulting dialogue box confirms the control's function and offers the opportunity to try again. For example, it is easy to hit the add-username-to-mailbox control instead of the add-mailbox control, and this confusion cannot easily be eliminated with a revised definition. The add-username dialogue therefore warns that it does not add a mailbox but offers a route to the dialogue that does.

The mechanical operation of all controls is smooth. Appropriate default functions are always active; for example, when an input field is empty, the default function might be "Next" or "Skip"; but the moment any text is entered, the default function changes to "Add" (or whatever normally happens to the input text). This helps the user ignore the interface and concentrate on the meaning.

Performance

In a mail server product, performance, measured as the number of messages processed per unit time, is usually a major concern. In previous products designed by MIG, performance was among the hand-

ful of top-priority goals, and from these we had learned a great deal about designing for the highest possible performance. We had also learned that the single-minded pursuit of performance is expensive, disruptive to implementation, and prone to error.

This experience yielded a set of informal rules for cost-effective design regarding application performance. These rules proved to be effective during the development of the AltaVista Mail product set. (Remember, these rules relate to the performance of typical applications: they would not apply to writing an operating system or other low-level code.)

1. Estimate the minimum disk I/O that the product operations will require, but treat this value only as a sanity check against gross waste of resources. Do not insist that this minimum be achieved.
2. Avoid checking special cases in a task's input data to avoid processing steps. Such optimizations are very likely to cause maintenance errors to go unnoticed and greatly increase the cost of the system test.
3. Never optimize tasks that consume only CPU time; examine only those algorithms suspected of being high-order consumers. It is almost never worth optimizing error cases.
4. Do not use complicated buffer management schemes to avoid copying data. Complicated code is prone to maintenance errors, and performance will not be helped much. Modern computers are very good at copying buffers of data but relatively slow at executing the complex branch logic that might be required to avoid copying data.
5. Take advantage of the high-performance system routines in modern operating systems. Do not build a memory allocator or a disk cache: the operating system developer has already spent far more on performance than an application developer can afford to spend. (Even if the operating system routines do not perform well, the problem will be common to all applications that use the platform.) Spend time solving the customer's problem, not repeating operating system development.
6. Use the operating system disk cache. It can be worthwhile to read even quite large amounts of file data in multiple passes if that will simplify program logic: the data will normally stay in memory for a subsequent pass. If the data has been flushed from the cache for the next pass, then the system had a better use for the memory, and any attempt to avoid multiple passes by remembering substantial state will degrade performance, not improve it.
7. Never offer an asynchronous application programmer interface (API), and avoid using asynchronous modes of otherwise synchronous services, even if that technique is presented as the way to obtain good performance. When more than a tiny region of code is directly affected by asynchronous events, it becomes difficult to be convinced that the code works. In addition, the code is unlikely to keep working as it is maintained. Synchronous methods—multithreading with thread-synchronous calls, polling, and timeouts—will normally yield perfectly adequate performance; they need only be avoided for very low-level code such as GUI window procedures.
8. Where latency requirements permit, polling for new work or configuration changes can be a better solution than an active notification path. Polling is easy to implement and robust. There is no need to ensure that an idle program is completely inactive: since modern operating systems page rather than swap, minor CPU attention every few seconds imposes no noticeable performance penalty.

Our approach to performance—and its nearly complete subordination to simplicity—can be seen in several aspects of AltaVista Mail's operation.

The server's function is to switch messages, so its database is a set of messages organized by target. Each message is held as a single text file; the text consists of the SMTP commands that would introduce each message to the server.

A message file is held in a directory that denotes its target, whether that target is a remote domain, a local mailbox, or a thread of the SMTP server itself. When the router decides on the target, it copies the message to its target directory and deletes the original. When a message splits to multiple targets, no attempt is made to share the common message data.

Although we could have designed a far more efficient way of representing the message database and splitting the message data as it flows through the server, our experience with previous products suggested that it was not necessary. Because the storage system we chose was a simple one, we could afford to throw it away if it did not work under load. Happily, it did work, and we gained a highly robust storage system with excellent performance for a trivial investment.

When a message is passed from one thread of the SMTP server to another, it is left in the target thread's input directory. There is no notification path; the target thread discovers the message by polling.

To make the source code as comprehensible as possible, the server uses extremely simple protocol parsers. The source code is organized in terms of a programmer's understanding of the protocol, not some abstraction that might be more efficient. The parsers scan the input data as many times as is convenient to extract the data they need at each level. The result is secure and reliable protocol machines that are easy to verify and modify when necessary.

Despite the apparent lack of care for conventional performance concerns, extreme workloads must be supported. All the server's components must support

huge numbers of messages, messages of huge size, messages with huge recipient lists, and messages bound for huge numbers of targets. Ideally, the system should impose no limits below those imposed by the underlying machines. An extreme workload should cause no problems or substantially reduce the work rate.

The POP3 server has to be able to support tens of thousands of messages in a mailbox. On connecting, most POP3 clients ask for a mailbox listing, which involves counting the size of each message in the mailbox. If thousands of messages are present, this can take so long that the client disconnects, believing the server has failed. Subsequent reconnections see exactly the same problem, and no mail flows. To avoid this problem, the server returns a partial listing to the client if it believes the full listing is taking too long. When the full listing is finished, it is written to disk to be returned to the client the next time it connects.

SMTP rules state that servers must support at least 100 recipients per message, and that ideally there should be no limit. Our existing customer base expects no limit. Arbitrarily large recipient lists can be allocated in virtual memory simply by configuring a page file of sufficient size. However, very large lists then impose a severe and highly variable load on the system. To keep the system load within reasonable bounds, we placed an upper bound on the amount of virtual memory claimed. The server limits the number of splits a single message can make while being routed. (A block of virtual memory is required for each split, not for each recipient; however, in the worst case, each recipient requires its own split.) Once a message's recipient list has split to more than 64 targets, subsequent recipients are moved to a new copy of the message that will be routed separately, regardless of whether the recipients could have been served by an existing split.

Error Handling and Error Reporting

The server has to work reliably, even in the face of errors in the local environment. Experience suggests that code containing many error paths does not work. If a function can fail for any of several individually identified reasons, and the calling code has to handle each reason separately, sooner or later some of those error paths will be incorrect. Checking that each path continues to work as development and maintenance continues makes the system test very expensive; furthermore, it is difficult to cause every possible error to arise when testing.

To ensure its reliability, the server uses the following implementation rule: any function is allowed to fail, but its calling code is not allowed to distinguish between the various reasons for failure. Indeed, it cannot make a distinction, because only one failure return code is defined. Functions were reworked as necessary, so the rule could be observed.

(A related observation is that the code will be more reliable when only one kind of success outcome is allowed. We found this to be true to some extent: it is equivalent to saying that a linear, nonconditional flow of control is more reliable than a highly conditional one. However, as long as every success outcome does occur during the product's normal operation, the code that handles it will probably continue to work, and the system test will have reason to check that it does.)

The server also has to report any errors it encounters: it must say which server operation could not be performed, precisely what system condition caused it to fail, and what to do about the problem. This apparently conflicting requirement is handled with a second rule: error handling must be completely separated from error reporting.

Error reporting works with the structure of the server. The server's flow of control and breakdown into threads were designed specifically to support precise error reporting. Each thread has a well-defined purpose that can, if necessary, be explained to the administrator, or at least named in a diagnostic report without causing confusion. Because a thread is fully in charge of its task, it is able to report the significance of any failures it encounters to the administrator.

A routine uses a simple stack-based error-posting module to report an encountered error. The report includes a description of the failed operation, the operation's parameters, and other helpful information such as the name and return status of any failed system call. The routine then returns the standard error status code. Its caller sees that this routine has failed and generates a report, logging the failure and adding any relevant parameters it holds. The caller then releases any resources associated with the failed request and returns the standard error status code. Eventually a high-level routine handles the failure, typically by logging the stack of error reports and continuing with its next item of work.

Our approach to error handling and reporting yielded extremely good results, but it had two serious implications. First, we could not import any source code from existing systems: all the code in the server had to use the error handling and reporting methods just described. Often, we could not use an established API definition for common functions. Second, the product and each of its components needs built-in knowledge of its function as perceived by the user, so it can report the true status of any problem and ideally give suggestions for fixing the problem. It needs to report the implications of the problem, not merely the facts of the problem. For these reasons, we could not use the powerful UNIX-style approach of building complex systems out of small, general-purpose tools.

Additional Necessary Features

In general, we tried to avoid adding features and keep in mind the server's one basic function: to move messages from place to place with minimal user intervention. When forced to add a feature, we aimed to keep it as simple and inexpensive to implement as possible, yet ensure that the feature offered the greatest real value. Obviously, this involved a trade-off, but it was usually clear how far to go.

The server needed a log subsystem to report important events, for example, errors encountered and suspected security violations (attempts to break into the server). To gain the maximum benefit from the log subsystem, we also used it to report the normal activities of the server in sufficient depth to perform a complete analysis of its work. This allows the logs to be used for load monitoring, performance analysis, message tracing, and billing and accounting. Enough information is logged to identify exactly what has happened to each message submitted to the server. Header information allows the originator and the recipients to be identified, and checksums for envelope and content allow duplicate messages to be detected. Duplicate detection is useful as a diagnostic aid and to avoid billing multiple times for a single message.

The POP3 protocol does not report a message's actual recipients (as opposed to the To: and Cc: fields, which may not be complete or even related to the real recipients). It therefore cannot be used as a way of delivering messages to gateways: it is only suitable for final delivery to recipients. For this reason, the addition of a proprietary interface could not be avoided. We chose to implement an API because it offers the simplest possible interface to the message data: sequential access to the return path, the recipients, the header fields, and the lines of content data. At the same time, it provides routines that encapsulate much of the complicated and error-prone logic that gateways often need.

For example, the API allows a gateway that is fetching a message to handle each recipient individually: it can accept, nondeliver, redirect, expand, or send for retry each of the recipients it sees. The gateway automatically generates any required new messages, including nondelivery reports, messages containing those recipients sent for retry, and messages with new recipients added by redirection or mailing list expansion.

The use of an API also guarantees that a gateway's operation is fully logged. When message-IDs and originator and recipient addresses are translated between SMTP and the foreign representation, the correspondence is logged. Messages can then be traced, even across gateway boundaries.

In addition to these features, we extended the services of the built-in HTTP server, which offers the administration Web pages. With a combination of server-parsed hypertext mark-up language (HTML) and a low-level attribute handling system to read and write server data, a customer can change both the appearance and the function of the Web pages, simply by editing HTML files.

Experience with the Product

The AltaVista Mail product set has achieved its design goals and has validated the implementation rules we imposed on it. It has been inexpensive to develop, support, and maintain; is a well-behaved and effective solution; offers excellent performance; and has yielded very few bugs.

Its major deficiency is that AltaVista Mail, by itself, does not form a complete solution. Despite our efforts to ensure that it can be run by inexperienced administrators, it relies on complex external technologies. Dial-up networking, Internet Protocol address assignment, and the other aspects of the interface to the Internet service provider present problems that are not easy to deal with.

Our major problem is the configuration of MX (routing information) records in DNS. Although the product reports misconfiguration accurately, users call us to find out what to do about it. Better integration with DNS would substantially reduce our support load.

Overall, we tried to keep the number of controls to the minimum. In retrospect, we did build in a few that perhaps should not have been provided. For example, the administrator has complete control over the SMTP timeouts. Although this is required by the relevant RFC documents, we should have had the confidence to pick values that worked everywhere rather than provide the control.

On the other hand, providing more control in certain areas would have expanded the range of customers who could use the product. For example, some low-end customers need to control the schedule on which SMTP connections and DNS requests are made, and the dial-up facilities we provide are too crude to do this.

Conclusions

This section reviews the organizational, design, and implementation rules we found most helpful in building the initial AltaVista Mail products. These ideas are the ones we recommend to engineers who are starting a new area of work.

Running a Development Project

If possible, develop a new application as a product-quality prototype, not as the implementation of an architecture. A prototype can be brought to market quickly and inexpensively and will generate helpful

feedback from customers. It is much more difficult to make sure an abstract architecture is not expensively addressing problems that do not need to be solved.

Do not develop prototypes that are not product-quality. A body of unshippable code that has been built as a proof of concept does not demonstrate the practicality of building a product. Take shortcuts, but not in any area that affects the application's fitness for use or maintainability. If the outcome is a shippable prototype, the choice can then be made whether to ship it or not.

Accept requirements input from anyone who will express an opinion but grant veto power only to those who are funding the project. Do not let a search for consensus slow the application's entry into the market but be very clear about why the application is the right thing, keep track of the risks, and be ready to respond to changes in the market. Do not use the beta test (open field test) merely to check if the product works; use it to decide whether the product needs changes. A change of mind at the last minute is not a failure.

Underfund software development. Allow enough time to produce the core of the solution and no more. This helps developers in two ways: they concentrate on what really matters, so they keep looking for the most effective ways to solve the largest possible parts of the problem, and they do not start "knitting." Developers enjoy developing; given adequate time, they will increase the functionality of the product. This will reduce the product's quality, not increase it. Given extra resources, it is unlikely that those funding development would choose the extra functionality a developer would like to build.

Defining a Product

Technical issues are never the most important considerations in defining a product. The most important thing to analyze is the customer profile. Instead of building a product, the goal should be solving problems for the customer. The more accurately the customer's problems can be characterized, the more effectively the product will solve them.

At the beginning, assume that the product will be sold at a very low price. Think through the implications; they will indicate the nature of an acceptable product. Then, if the price is higher, add the extra features that the increased price will require.

Assume that the product will be supported directly from the engineering group. Imagine what would be necessary to support the product and define requirements to impose on the product that will minimize the cost of providing support. Then, if the product is supported elsewhere, add the extra features that the external support structure will require.

Design and Implementation

Only a limited number of clever and difficult components can be designed and built into the product. Make sure they are the ones that will make the most difference, and make sure each difficult part is as small as it can be by encapsulating it in a simple interface. Outside the most critical areas, use the simplest designs possible. Do not simplify to spend less time on design: simplify to improve the quality of the product and to reduce the cost of implementation and maintenance.

Early in the project, determine the performance goals for the product. Choose a small number of areas in which to be careful and ruthlessly simplify the rest. Once learned, design for performance is a skill that is difficult to keep under control. However, time spent on performance is an investment, and a deliberate choice of investments is needed.

References

1. D. Crocker, "Standard for the Format of ARPA Internet Text Messages," RFC 822 (August 1982).
2. N. Borenstein and N. Freed, "MIME (Multipurpose Internet Mail Extensions) Part One: Mechanisms for Specifying and Describing the Format of Internet Message Bodies," RFC 1521 (September 1993).
3. K. Moore, "MIME (Multipurpose Internet Mail Extensions) Part Two: Message Header Extensions for Non-ASCII Text," RFC 1522 (September 1993).
4. J. Postel, "Simple Mail Transfer Protocol," RFC 821 (August 1982).
5. C. Partridge, "Mail Routing and the Domain System," RFC 974 (January 1986).
6. P. Mockapetris, "Domain Names—Concepts and Facilities," RFC 1034 (November 1987).
7. P. Mockapetris, "Domain Names—Implementation and Specification," RFC 1035 (November 1987).
8. J. Myers and M. Rose, "Post Office Protocol—Version 3," RFC 1725 (November 1994).
9. M. Crispin, "Internet Message Access Protocol—Version 4, Revision 1," RFC 1730 (December 1996).

Biography

Nick Shipman joined DIGITAL in 1982 after earning a B.Sc. in computer science from University College London. As a member of the Mail Interchange Group (MIG), Nick has been involved with the design and development of a variety of e-mail and directory server products and their test tools. He is currently a principal software engineer with the DIGITAL UK Engineering Group, where he is working on advanced development of Internet-based servers.