An Active Mail System

John Hogg
Stelios Gamvroulas

Computer Systems Research Group
University of Toronto

ABSTRACT: Conventional electronic mail is *passive* text that is created, sent and read. Any further actions must be initiated by the recipient. By contrast, an "intelligent message" (*imessage*) is an active program that carries on a dialogue with the recipient. The imessage may subsequently route itself to other users as a result of the responses it receives, and it eventually returns these responses to the original sender.

This paper describes a prototype intelligent mail system and discusses some of the problems involved in implementing such a system in a distributed environment.

Introduction

In the growing field of office systems [ELL80], no system today is considered complete unless it has an electronic mail facility. These systems are basically similar: they allow text messages to be composed and sent to other users on the system or network.

A *role* in an office environment is defined as a person performing a function (e.g., a secretary). Communication among roles in the office has in past been carried out by means of passive messages. Traditionally, a message (i.e., a piece of text conveying some information) is created by a sender and is shipped to a number of recipients. A message can only convey information; it cannot collect information. Its task is finished when it has been read by the recipient.

A different approach is to make the message "intelligent". An intelligent message is able to perform complicated tasks such as collecting responses from recipients [VIT81] and routing itself to recipients according to predefined rules [MAZ83], [TSI84].

At this point a specific example may make things clearer. Suppose that a user finds himself with a number of tickets to a concert being given by Wild Willy. Somehow, he must publicise the fact that these tickets are for sale. The conventional approach would be to issue a broadcast message to all system users saying "Wild Willy concert tickets for sale". However, since this particular artist attracts a limited following, such a broadcast would be merely annoying junk mail to most of its recipients. Furthermore, if a large number of people actually want tickets then the user will find that he does not have enough.

The intelligent mail (*imail*) approach is different. The sender identifies a potential set of Wild Willy fans and sends them an intelligent message (*imessage*). It asks them if they are interested in tickets and if so, how many they would like. It then asks them for the names of other possible buyers. When all tickets have been sold (or no more potential buyers can be suggested) the imessage *terminates*, disappearing from mailboxes and returning its results to the original sender.

This paper describes the design and implementation of an imessage system called *imail*. Section 2 explains the concept of an imessage. Sections 3 and 4 explain the reception and creation of an imessage respectively. Section 5 describes the language used to build an imessage. An example of an actual script is given in Section 6, and implementation details are briefly covered in Section 7. Finally, Section 8 discusses the problems involved in putting imail on a distributed system.

Imessage Concepts

Conventional electronic mail is passive text that is *read*. An imessage, by contrast, is a program that is *run*. It carries on a dialogue with each recipient, asking questions and collecting answers. Thus, the sender does not need to depend upon the recipient creating another passive message in response to some question, as this is done by the imessage itself. Conventional message routing is *static*. The original sender specifies a list of recipients and the message is sent to exactly the destinations on that list. Imessage routing, however, is *dynamic* [MAZ83]. After each invocation an imessage may route itself to further destinations depending upon the answers it receives. As demonstrated in the example above, this routing need not be to a single destination; several recipients can be sent an imessage at the same time. This can cause problems, as discussed later.

In general, a collection of responses will need to be processed in some manner. This processing can be done "on the fly" as responses come in, or after the fact when there are no more copies of the imessage awaiting invocation and the imessage has terminated. Both approaches have advantages. In the example above, the imessage has to perform a simple summing of numbers in order to decide when to terminate. This clearly has to be done after each invocation. On the other hand, complex statistical massaging of a set of responses can more easily be done by the sender with the aid of a library of routines. This should therefore be done after all responses have been collected. Therefore, the imail approach was to allow both options by providing "global variables" to retain numeric or text data from one invocation to another, and also returning all responses to the sender when the imessage terminates.

Receiving Imail

Imail was developed on a UNIX* system, so the interface was made similar to that of the UNIX *mail* system [SHO79]. When it is invoked by typing the command *imail* with no arguments [GAM83], a list of mail headers is displayed showing the sender's name, the sending date and an optional subject description. The recipient then specifies the number of an imessage and a command r(un), d(elete) or q(uit). If no number is specified, the next imessage in the list is always taken. If no command is specified, the message is run. Thus, a new user can receive imail by knowing only the command name and the location of the RETURN key.

If the recipient quits before all imessages have been disposed of, they remain in his or her imailbox. A "junk" imessage can be disposed of by using the delete command; the message will be removed from the imailbox and the fact of its deletion will be recorded as part of the imessage's *history*. When an imessage invocation has been started it may still be aborted in the usual UNIX fashion (by hitting the RUBOUT key) and the replies given up to that point will be thrown away, and this is also recorded in the imessage history. In fact, the fact that an imessage header has been seen is logged as well. This was done because it was technically feasible and useful in a system under development. The human factors implications, however, may make this undesirable in a full-scale system.

Creating Imail

An imessage is created by writing an imessage script and giving it as input to imail invoked with a list of initial destinations as arguments. Two other arguments are possible: a subject which will appear in the header that the recipients see and a timeout. This is an absolute or relative date (eg., "next Wednesday") at which the imessage will be terminated if it

* UNIX is a trademark of Bell Laboratories

has not already completed its task.

The imessage script itself is written in a simple imail language. Other alternatives were considered, such as a menu system or a "by-example" type of interface [ZLO80]. However, a small programming language was considered to be the simplest option to implement. On top of this, a menu system for applications-oriented imessages can be built.

The imessage language itself is described in the following section.

The Imessage Language

Initially, the imessage language was conceived of as a very simple set of commands which would be usable by typical office workers. As the project progressed, power won out over simplicity, so that the present version is more suited to its present environment, a university setting. However, in relation to conventional programming languages it is straightforward and is no harder to use than a typical text editor.

An imessage is a series of *questions*. Each question is indicated by a line starting with a '>' and containing an optional label (a word). Questions are blocks of text, followed by a line indented one tab stop containing the command, "get ...", where "..." is a description of the number and type of replies expected. An upper and lower bound may be given and the type of the reply can be numbers, words, a block of text, or users' id's. (In the UNIX environment, logins were used as userid's.) If the recipient should give an incorrect reply to a question, the get command will explain the error and reprompt for a more appropriate reply automatically. Subsequent commands are also tabbed in one or more positions. A simple "if" command is provided; the block of commands to which it applies are tabbed in one additional stop, and ifs may be nested arbitrarily deep.

The set of commands is small. *Print* will print out a short message. An imessage may be terminated with the *terminate* command. It was mentioned earlier that the questions posed in an imessage dialogue may depend upon earlier replies; this is accomplished by using the *next* command which specifies the question to be posed next as an absolute or relative question number or as a string label. One or more new destinations for the imessage may be added using the *ship* command. Normally, an imessage will not be sent to a recipient more than once. If a revisiting is desired, then *reship* must be used. Finally, *set* is used to set variable values.

There is one *response variable* associated with each question. The answer given by a recipient is assigned to it, and it may be referenced thereafter by the #<label> in the case of a labelled question or #<number> and #<-number> in any case. For instance, within the 12th question, the response to a tenth question labelled "tickets" may be referred to as "#tickets", "#10" or "#-2". In addition there are *local* and *global* variables in which the response variable "#" has been replaced by a "!" and a "?" respectively. Local variables may be used to store the results of intermediate computations within an invocation, and disappear when the dialogue has finished. Global variables exist from invocation to invocation and thus can be used to keep running sums and the like. In this way, processing of responses may be done while an imessage is in circulation as mentioned earlier. Initializations of locals and globals may be given at the beginning of the script. Globals are initialized once and locals are reinitialized each time the script is run.

An Example of an Imessage Script

A concrete example of an imessage script might be helpful at this point. The ticket sales problem explained above can be handled by the following:

```
number ?numleft = 10
>
I have a few tickets to the "Wild Willy"
concert on Saturday and will sell them
for $5.00 apiece. Do you want any?
        get 1 boolean
        if #1 = no
                next who
>num
How many would you like? There are
only ?numleft left.
```

```
        get 1 number
        if #num > ?numleft
                print Sorry, there are only
                print ?numleft left.
                next num
        set ?numleft = ?numleft - #2
        if ?numleft = 0
                print Thanks!
                terminate
>who
Who else do you know that might be
interested? (logins, please)
        get logins
        ship #who
>
Thanks!
```

When the imessage is created there are ten tickets that the user wishes to sell, so a global variable *?numleft* is initialized to 10. This imessage is initially sent out to a set of people whom the sender knows are Wild Willy aficionados. Each time the imessage is run, it asks whether the recipient is interested in purchasing tickets. If the answer begins with an 'n' then the second question is skipped. Otherwise the number of tickets desired is requested (with care being taken not to sell more tickets than are available) and the number remaining is decremented. Note that "*#2*" and "*#num*" are synonyms for the same thing: the response to this question. If buyers have been found for all the tickets, the imessage terminates, removing itself from all mailboxes. Otherwise it asks for additional possible buyers. If this imessage has already been sent to them, they will not be revisited (since *ship* and not *reship* has been used); otherwise the imessage will be added to their mailbox.

Implementation of Imail

Imail was developed on a VAX 11-780 running Berkeley UNIX. It has also been ported to a MC68000-based SUN workstation. While it is UNIX-based, a variant could be built for an MC68000 PBX and this would be a natural environment for such a system.

The programming was done in "C" [KER78] with the aid of the LEX scanner generator [LES75] and the YACC parser generator [JOH75]. The latter two were used to in building the translator which converts imail scripts to *csh*, the C Shell [JOY80]. By using csh as a very powerful target language, program development was greatly simplified.

The present version of imail runs on a single machine and uses a single set of files for all the copies of each imessage. This means that simple locking has to be performed to prevent two invocations from destroying data, but it avoids other coordination problems. The files are stored in a central post office account and are owned by the post office.

This is an important point. Imail is much more powerful than conventional mail and therefore potentially far more dangerous. An imessage is a program whose actions are not known by the recipient, and there is clearly a great opportunity for electronic guerrillas to throw letter bombs that delete all the recipient's files. The defence is to restrict the abilities of the sender and also to keep the imessage out of the recipient's environment, at some cost in power. Imessages are not arbitrary programs; they are composed only of sequences of commands of the sort shown above. They are therefore unable to damage files or surreptitiously collect information. Furthermore, using the UNIX "setuserid" permission on the imail program makes the recipient's environment be that of the post office for the duration of the imail session. In this way only the post office could be hurt.

The imail program can be used not only for sending and receiving imail, but also monitoring its progress. The *monitor* mode is menu driven and allows the sender of an imessage to see where it has been, where it is going and the responses it has collected to date, and also to alter the address list, i.e., add or remove the imessage from user's mailboxes. This can of course only be done by the owner (sender) of the imessage.

Distributed Imail

Imail as described can be run on a centralized environment, e.g., on a local net or PBX. Implementing imail over a local net does not present great difficulties. One machine can contain the post office and there can be complete coordination between all copies of an imessage. As long as this machine does not go down, the problem is basically the same as running imail on a single processor.

Problems arise, however, when we attempt to achieve true distribution of a system, with distributed control [BRU75], [SHO82]. In this paper, the "distribution" that we discuss is distribution over a loosely-coupled network. Here the problems become much more challenging. Control must be decentralized and, due to the slow speed and the unreliability of links between nodes, copies may no longer be aware of each other's existence. An example of a net where these conditions hold is USENET. It consists of a large number of machines (several hundred, mostly running UNIX) connected in an irregular topology by periodic connections over phone lines. Individual machines or links are frequently down and even when they are not it takes days for a message to travel across the entire net. Centralized control is not possible.

USENET can be described as an irregular point-to-point net. The exact topology of the network should not, however, affect the problems of distributing imail to any great extent. Imessages are sent from one user to another. While users at the same node may be treated specially, users at different nodes can be treated in the same way whether they are one hop apart or many. The important concept is that they are remote from each other.

Imessages as Trees

After an imessage copy is run it may be sent to zero or more other users. In order to model the behaviour of an imessage we need to represent this routing. One possibility is to consider each imessage as a tree, with nodes representing copies. Then each time a copy is run it produces zero or more children and dies (or perhaps becomes dormant). The root of the tree is thus the copy held by the original sender. Internal nodes are inactive copies that have been sent and run and have spawned off new copies, and leaves are of two types: those copies that have not yet been run and those that have been run but have spawned no children. Consider the following examples:

(a) A simple non-intelligent message can be sent to one or more stations. After it is received, it dies. The imessage copy has zero children.

(b) A wide variety of applications can be envisaged in which a single imessage copy mimics a paper document by being passed along to a single station each time that it is run. This can instead be viewed as an imessage copy generating a single child.

(c) Another example used in the single-machine prototype was the group search, where the imessage asked for a list of likely destinations and was shipped to all of them. If a destination was specified by more than one respondent, the imessage was only sent to that station once. In other words, each copy had n children which could be merged with each other.

(d) "Chain letters" are a familiar if ridiculous paper form of communication. They would be better named "pyramid letters", but they are worth considering in the imail context because they have an important property. A pyramid letter has a list of the last n recipients appended. The recipient is instructed to mail something to the name at the top of the list, delete this name, and add his or her own name to the bottom. Thus each copy of the letter is unique. If a pyramid letter is implemented in imail, each copy will have m children, none of which can be merged with any other copy.

(e) A facility that has been proposed or included in various office system prototypes is an appointment scheduler. Given a set of calendars to work from it can arrange an appointment between two or more peo-

ple by finding a time when all are free. One way to set this up in a distributed system would be to start up an imessage which locates the first free slot in the originator's calendar then sends a copy of this proposed time to each of the other calendars. If any of these messages detects a conflict it scans down the recipient's calendar to the next free slot and sends its own proposal to all the other calendars. Success occurs when each calendar receives and accepts a copy of the imessage for some given time.

Should a copy of the imessage "meet" a copy that has an earlier proposed date, the older one is killed since its proposal will obviously be unacceptable somewhere. Thus, we have an example of imessage routing where copies can generate a number of children but can kill off distant relatives.

The advantage of this "family tree" representation is that it breaks the imessage down into units that can be dealt with. Each copy has a short lifetime, a small task and a limited amount of communication to perform thereafter. Treating a copy as an entity that participates in may dialogues and is shipped to many nodes makes it much more complicated and difficult to handle.

Imessages as Objects

Another way of viewing an imessage is as an *object*, or rather a class of objects [ROB81] travelling through a network. In this model each of the imessage copies is an instance of the object class. The copies coordinate with each other through the use of messages (in the Smalltalk sense of the word) and the coordination actions are carried out by methods. The imessage script is itself a special type of method.

This view of an imessage is not mutually exclusive with that of a tree and has been used in other systems sharing some of imail's properties [BYR82]. The model has to be extended slightly to allow an object (an imessage copy) to be sent, not as a message to another object, but as an entity in itself, since the script or method of interacting with a recipient is a property of the imessage. This is a useful model which we are continuing to develop.

Distribution Problems

Much work has been done in the past on the coordination of distributed processes, particularly in the area of databases. Imessages, however, introduce a new set of problems. In the database environment all the databases and their locations are both known and fixed. Imessage copies move about about a network and may spawn multiple copies as they go. These may be sent to nodes unknown to both the original sender and also other copies. The problems are therefore not traditional locking ones, but rather difficulties in locating and remaining in communication with a varying number of processes in possibly unknown and changing locations.

An imessage is sent off to perform a specific task. When this task is completed the imessage returns a response to the sender and terminates. There are several problems involved in doing this. The most obvious ones are, "How do we know that we have completed the task?", and, "How do we terminate?".

In the simplest cases there is no problem in deciding when an imessage has completed its task; the most trivial imessage requires no response. A slightly more complex imessage will use one copy to collect one response from one recipient and then terminate. Extending this further, an imessage may generate a number of copies which independently return separate results and die. So long as the sender knows how many responses to expect (perhaps by parents reporting births) the problems are still straightforward.

A more challenging problem involves an imessage which spawns an arbitrary number of copies but must return a minimal number of responses, after which all copies should die. One solution involves having a copy produce its offspring then remain at the node where it was run; all copies report responses to their parents. This means that a copy cannot die until its children have completed their tasks. Results will then percolate up the family tree

until the root (or perhaps one of its descendants) finds that the minimal number of responses has been obtained. This method has many drawbacks, however; to begin with, it relies upon nodes containing dormant parents always being up.

A very difficult problem is the handling of an imessage which is to return a given set of responses. An example would be an imessage to satisfy a knapsack problem where the items are spread over the net. Can we obtain a solution (or, in realistic time, an approximation) using imessages? What classes of problems can imessages handle?

When an imessage has completed its task, there still remains the problem of informing a possibly unknown number of copies at an unknown set of sites of this fact. This problem is in some ways similar to that of coordinating responses, and could be solved using family trees with parents sending death wishes to their children before dying. Once again, this requires that nodes holding parents remain up, even though the parent itself is only being used to perform this termination coordination.

Self-Repairing Imessages

The example of a loosely-coupled net used above, USENET, is notorious for poor reliability. Nodes are constantly going down and when they are up, many messages are lost or mangled. An imessage scheme that relies on all transmissions being successful could not be used in such an environment. Copies would be left waiting for results that would never arrive, or would disappear into the net spawning children as they went with no control from above. This makes the idea of a self-repairing imessage very attractive. If an extremely intelligent imessage copy expects a communication from a parent or child and none arrives after a certain interval, it could determine the nature of the damage to the message as a whole. Then the missing copies could be reconstructed, possibly with the loss of some information. In a major disaster, merely being able to identify that a problem exists and being able to kill "lost" copies would be sufficient. This, however, requires that some information be exchanged between copies on a parent-child and/or sibling level. The minimal amount of information that can be used to detect or repair a given amount of damage then becomes an interesting problem.

Ongoing Work in Distribution

These are just some of the problems to be solved in the area of distributed imail. Clearly, the first task is to develop a notation that can be used to state these problems formally. This notation should be able to handle at least three views of an imessage. The first is that of an omniscient observer who can see the state of all network nodes and copies simultaneously. The second is that of the sender who watches responses come in in pieces, and the third is a copy's view of its environment with its limited knowledge of other copies and other nodes.

Conclusions

The imail project is an ongoing one. The imail language, as mentioned above, is not as simple as it could be and is in fact still being developed. In the long run, some other approach than a procedural programming language one would probably be preferable. At present, our main area of interest is in the distribution of imail and the problems involved in doing so.

One set of problems that we have not yet addressed are those concerning the human factors and sociological implications of an imail system. While this is due to a desire to have a working system to discuss first, it does not make these problems less important. A conventional message is sent to a recipient's mailbox and becomes his or her private property. It may be read, refiled for future consideration, or deleted, but only by the recipient. By contrast, the nature of an imessage requires that it belong to and eventually return to its sender. We have added additional capabilities that tell the sender when the message was sent to a recipient, or when its header was seen, or when it was partially or fully run or deleted. A recipient cannot claim, "I haven't read my mail yet" unless this is actually the case. While this may encourage truthfulness in human communication, it ignores human reality.

Imail is a prototype system and we have included facilities that we feel would not be appropriate in a working office environment, however convenient they may be to system developers.

The best way to present imail to an office environment is thus still very much an open question. Nonetheless, imail appears to be a potentially powerful tool for office communication. We are continuing to explore this power.

References

[BRU75] J. Brunner, *The Shockwave Rider*, Ballantine, New York, U.S.A.

[BYR82] R.J. Byrd, S.E. Smith and S.P. deJong, "An Actor-Based Programming System", SIGOA Conference on Office Information Systems, *SIGOA Newsletter, Vol. 3, Nos. 1 and 2.*

[ELL80] C.A. Ellis and G.J. Nutt, "Office Information Systems and Computer Science", *ACM Computing Surveys, Vol. 12, No. 1*, March 1980.

[GAM83] S. Gamvroulas, *The Imail User's Manual*, working paper.

[JOH75] S. C. Johnson, *Yacc: Yet Another Compiler Compiler*, Comp. Sci. Tech. Rep. No. 32, Bell Laboratories, Murray Hill, N.J., U.S.A.

[JOY80] W. Joy, "An Introduction to the C Shell", *UNIX Programmer's Manual, Vol. 2c*, Department of Electrical Engineering and Computer Science, University of California, Berkeley, U.S.A.

[KER78] B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Prentice-Hall, N.J., U.S.A.

[LES75] M. E. Lesk and E. Schmidt, *Lex - A Lexical Analyzer Generator*, Comp. Sci. Tech. Rep. No. 39, Bell Laboratories, Murray Hill, N.J., U.S.A.

[MAZ83] M. S. Mazer, *The Specification of Routings in a Message Management System*, M.Sc. Thesis, Department of Computer Science, U. of Toronto.

[ROB81] D. Robson, "Object-Oriented Software Systems", *Byte Vol. 6, No. 8* (August 1981).

[SHO82] J.F. Shoch and J.A. Hupp, "The 'Worm' Programs - Early Experience with a Distributed Computation", *CACM Vol. 25, No. 3* (March 1982).

[SHO79] K. Shoens, "Mail Reference Manual, Version 1.3", *UNIX Manuals.*

[TSI84] D. Tsichritzis, "Message Addressing Schemes", *ACM Transactions on Office Information Systems Vol. 2, No. 1* (January 1984).

[VIT81] J. Vittal, "Active Message Processing: Messages as Messengers," in *Computer Message Systems*, R. P. Uhlig (editor), North-Holland.

[ZLO80] M.M. Zloof, "A Language for Office and Business Automation", *1980 Office Automation Conference Digest*, Atlanta, U.S.A., March 1980.