A PRACTICAL, DISTRIBUTED ENVIRONMENT FOR MACINTOSH SOFTWARE DEVELOPMENT

Mark Sherman, Ann Marks, Rob Collins,
Heather Anderson, Jerry Godes, Denis Devlin,
Leonid Spector, Vivian Sewelson

Technical Report PCS-TR86-116

A Practical, Distributed Environment for Macintosh Software Development

Mark Sherman[1]
Ann Marks[2]
Rob Collins[1]
Heather Anderson[2]
Jerry Godes[1]
Denis Devlin[1]
Leonid Spector[3]
Vivian Sewelson[1]

[1]Department of Mathematics and Computer Science
Dartmouth College
Hanover, NH 03755

[2]Thayer School of Engineering
Dartmouth College
Hanover, NH 03755

[3]Department of Psychology
Carnegie-Mellon University
Pittsburgh, PA. 15213

Abstract

We describe a development environment we created for prototyping software for the Macintosh. The programs are developed and executed on a large time-shared computer but can use the full facilities of the Macintosh. By using this system, we combine the advantages of the large system, such as large amounts of disk storage and automatic file backups, with the advantages of the Macintosh, such as advanced graphics, mouse control and sound synthesis. We also describe several projects that used the distributed development system. We conclude with a description of our future plans for this environment.

Overview

In this paper, we describe a distributed environment we created that allows us to use the facilities and tools

on a large-time sharing computer to write programs for the Macintosh [Apple 1, Williams 84] without the use of cross-compilers. We first describe the facilities provided by our main computer that we want to keep (and that are not provided by the Macintosh), then we describe how we created a distributed system that allows programs developed on the main computer to appear as if they were running on the Macintosh with access to the Macintosh's facilities. In the next section, we present a critique of our system and explain some generic problems that may be encountered by others using our method for building other distributed environments. We conclude by relating some projects that used the system and discussing the future of our system.

Introduction

The Macintosh is a relatively new personal computer with limited tools for native development work. However, the Macintosh offers many exciting possibilities for use in the classroom because of its graphics, fonts, sound system and pointer device (mouse). When we started acquiring Macintoshes over 18 months ago, we needed a way to create programs for both ourselves and our students.

Unfortunately, the best development environment available was the Lisa Development System, essentially another stand-alone personal computer that had to be dedicated to a single user. We did not have the resources to provide a Lisa system to everyone who wanted to program the Macintosh, so we developed an alternative: a distributed development environment split between the Macintosh and our large time-sharing facility.

Development Facilities Provided by a Time-Sharing System

Large, shared computer systems provide a variety of facilities that we take for granted but that are essential

for developing large systems. Unfortunately, most of these facilties are unavailable for a small personal computer, such as the Macintosh. Some of these facilities are described below.

High Speed Printers

Programmers need to produce copies of their documentation, design specifications and code for study and exchange. A moderate sized program on the Macintosh represents nearly a hundred pages of Pascal code. However, printers capable of producing large amounts of printed material can be economically run only on a shared system. The Macintosh's printers are far too slow for serious development work.

Large File System

Most projects store large numbers of interface files, multiple versions of released programs, test data, and partially built systems. Plain Macintoshes have limited storage facilities, about 800K per machine. Although harddisks are now available, they are relatively expensive and generally provide 10-20 megabtyes of storage. Much of that space is used for system overhead (editors, compilers, utilities and so on) so relatively little is left for application development.

Automatic Backup

Over the years, computation centers have developed procedures for archiving files on a daily, weekly and monthly basis. In every project, there is some catastrophe which wipes out the current version of the work. Naturally, we now rely on automatic backups to recover from such disasters. Unfortunately, backups on personal computers is time consuming, inconvenient, and therefore, haphazardly done. Therefore, we would prefer to let files be safe-guarded by time-tested techniques.

Fast, Debugged Compilers

The PL/1 compiler on the main computer system has been carefully optimized over the years to produce excellent code quickly. By contrast, most compilers for the Macintosh are early releases that are being shaken down. Further, the main computer has more computational horsepower available, though in smaller spurts. For many short test programs, a small amount of high-powered processing works better than complete access to low-powered processing.

Easy Access

The main computers can be accessed from thousands of terminals throughout the campus whereas a Macintosh development system has two limitations: hardware access and software access. Hardware access is limited to those Macintoshes that have the necessary amounts of memory and disk space for program development -- there are relatively few of these machines. Software access is limited to those machines for which an appropriate package has been purchased. On the main computer, a tool purchased for one user is available to all. Therefore everyone on the main computer has easy access to all debuggers, linkers, editors, compilers and file utilities. Unfortunately, each tool on the Macintosh is a separate program which must be individually purchased for each person doing development. This situation drastically limits the access to software for doing development.

Sophisticated Editors

The main computer has text editors that use windows, work on difference files between several versions of a program, do pattern matching and replacement, do pretty printing and perform other editing operations.

The Macintosh comes with a single program editor that contains a subset of these facilities.

Batch Facility

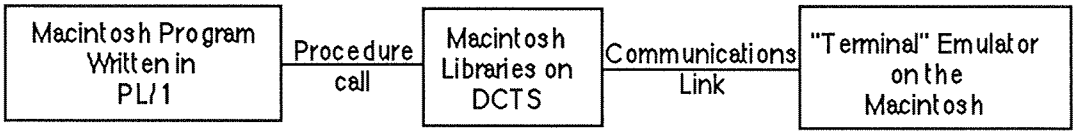
The main computer has facilities for collecting a series of operations and having those operations performed together, usually without programmer intervention. Like most personal computers, the Macintosh requires the programmer to be present during each step of the translation process and manually indicate which files to compile, link, load and execute. The manual intervention is tedious and wasteful when an entire system-creation cycle takes 30 minutes.

Mail Facility

People working a project need a way to communicate with one another. Computer mail works well, but is available only on our time sharing system. Therefore, a multi-person project has trouble working exclusively on a Macintosh -- they need to use the main system for communication and file sharing.

Tying the Main Frame to the Macintosh

We wanted to keep the facilities described above but also wanted programmers to use the full range of Macintosh facilities, such as good bit-mapped graphics, mouse interaction, sound generation and windows. As an answer, we created a symbiotic development environment that was distributed between the Macintosh and our main computer. A programmer can write programs in PL/1 using libraries that essentially look like the system libraries on the Macintosh. The libraries on the main computer communicate with a special terminal program on the Macintosh, giving the appearance that the program on the time sharing system is actually running on the Macintosh. The system architecture is illustrated below:



We have two version of the Macintosh communications program, one that works over serial lines at 9600 baud, the other works over the AppleTalk [Apple 2] network at 220 kilobit/second. The communications program on the Macintosh operates in two modes: either as an ordinary terminal, or in the special "graphics" mode which implements the protocol for remotely calling procedures on the Macintosh from the main computer.

Maintaining Macintosh Fidelity

Several projects have used our development system wih good results. But in trying to make programs running on the main computer appear to be running on the Macintosh, several Macintosh features had to be altered slightly. We believe that these problems are inherint in using our technique between any two computers that have different architectures.

Real-time response

Application programs running on the main computer do not possess the same real-time response as native Macintosh programs. For example, the mouse does not track as well and screen animation by alternating bitmaps works poorly. There are two reasons why the real-time performance is limited. First, the Macintosh can generate and accept data faster than the communications media can move them. Second, the time-sharing system cannot guarantee a processing quantum when the Macintosh delivers information to

the application program. Even though the usual device buffering will ensure that nothing is lost, the data received by the application program are usually too old to be useful as real-time feedback.

Although an application program can use the standard Macintosh calls and live with poor performance, we distributed some of the Macintosh features in order to improve response. For example, any calls of the operating system that use the mouse internally are processed locally. Thus Macintosh dialogs work as well in our distributed system as in native programs. A second technique we used was to rely on the native file system and Macintosh "resources" to reduce the amount of traffic between the host and the Macintosh. To display a picture (or play a song), for example, a programmer would place the picture (song) in a file on the Macintosh and just send down a request to display the picture (or play the song). Such an approach is substantially faster than shipping down the entire bitmap for the picture (or sound record for the song).

Dynamic Storage

Application programs running on the main computers have a different view of storage than native applications. The main computer has a larger address space and a larger word size than the Macintosh, so application programs can build larger data structures on the main computer than if they were running only on the Macintosh. Further, an application will appear to have even more memory than is available on the main computer because internal Macintosh operating system requests for requested are done locally on the Macintosh. For example, the definition of a polygon requires the Macintosh to allocate and manipulate an internal data structure. Normally, memory requested by an application on a Macintosh would would compete for the space used by the polygon. In our system, the storage requested within the operating system stays on the Macintosh; storage requested by the application stays on the main computer.

Architectural Difference

Differences in the representation of numbers can always cause problems when moving between two machines. We attempted to simulate Macintosh precision and accuracy with PL/1 variables. The libraries on the main computer check for inconsistencies and try to correct them. Although close, we know of situations where the different representations of integers and floating point numbers could cause problems.

Communication Enforced Semantics

Because we are using a message-passing semantics for procedure calls, we had to alter Pascal semantics slightly. First, all procedures that use VAR parameters (pass by reference) actually use value-result mechanisms. This is a standard technique for remote procedure calls [Birrell 84]. However, problems could occur in situations where several VAR parameters are aliased to each other. Second, we never pass pointers directly. As necessary, our distributed system exchanges small integers between the communications program on the Macintosh and the libraries on the main machine. The small integer is used to index a table of pointers on the Macintosh, where local data structures resides. Thus pointers can be interpreted only through the use of calls provided by the libraries. Direct pointer manipulation is not allowed. Our technqiue is a simplification of the more general pointer reconstruction techniques discussed elsewhere. [Herlihy 80]

Sample Projects

Several projects have used this system to develop prototype Macintosh applications. One is an implementation of the word game Boggle® that pits a player against the computer. With the large dictionary and fast disk of the main computer behind the prototype, we believe that the distributed version

runs better than will the final version on the Macintosh alone. Two modeling programs, one for an economy and the other for automobile traffic, were also developed. All three programs heavily used the graphics facilities of the Macintosh to illustrate program operation and status, and graphs evaluating various results.

A fourth project, providing speech for the handicapped, used many more of the facilities. For example, the program presented several dialogs that were used to control the MacInTalk speech synthesizer [Apple 3]. Severely paralyzed patients were able to use this program (along with a special mouse replacement) to communicate with other people.

The student programmers on these projects used the same PL/1 facilities that they had learned in class and had little trouble writing Macintosh programs in the environment.

Conclusions

We have been using the current system for nearly a year now and are satisfied with it. The libraries have been transported from our DCTS machine and PL/1 to a Vax/Unix system and C, and various projects are using the new facilities at Dartmouth and Carnegie-Mellon University. Several new projects are using the software and our system provides a natural path for translating programs that only ran on the main computer into native Macintosh applications.

Although we have not seen anything published, a group at Brown University is building a similar system for Unix where the libraries and emulator are called *Harold* and *Maude* . Apple has developed something similar internally [Zarakov 86]. Therefore, we conclude that the techniques we used to create our development environment, namely the building of the libraries, the terminal emulation program, and the protocol definition, seem to be a practical way to quickly bootstrap a realistic development environment for

a new personal machine. Although an exact match to a native system is not possible, we can get a reasonable good approximation for relatively little effort.

References

[Apple 1] Apple Computer, *Inside Macintosh*, Addison-Wesley, Reading, MA. 1986.

[Apple 2] Apple Computer, *Applebus Developers Handbook*, 20525 Mariani Ave., Cupertino, CA. 95014, July 20, 1984.

[Apple 3] Apple Computer, *MacInTalk 1.1: The Macintosh Speech Synthesizer*, 20525 Mariani Ave., Cupertino, CA. 95014, June 13, 1985.

[Birrell 84] Andrew D. Birrell and Bruce Jay Nelson, "Implementing Remote Procedure Calls," *ACM Transactions on Computer Systems*, Vol. 2, No. 1, February 1984, p. 39-59.

[Herlihy 80] Maurice Peter Herlihy, *Transmitting Abstract Values in Messages*, Technical Report MIT/LCS/TR-234, Department of Electrical Engineering and Computer Science, M.I.T., Cambridge, MA. 02139, 1980.

[Williams 84] Gregg Williams, "The Apple Macintosh Computer," *Byte*, February 1984, p. 30-53.

[Zarakov 86] Eric Zarakov, "MacWorkStation," *Wheels for the Mind*, Vol. 2, No. 1, Winter 1986, Computer Sciences F430, Boston College, Chestnut Hill, MA. 02167, p. 56-59.