January 1982

SOFTWARE DESIGN ISSUES IN THE ARCHITECTURE
AND IMPLEMENTATION OF DISTRIBUTED
TEXT EDITORS

by

Robert N. Goldberg

DCS-TR-110

Department of Computer Science
Hill Center for the Mathematical Sciences
Busch Campus, Rutgers University
New Brunswick, N.J. 08903

A Ph.D. thesis submitted to The Graduate School -- New Brunswick, RUTGERS,
The State University of New Jersey, in partial fulfillment of the requirements
for the degree of Doctor of Philosophy, Graduate Program in Computer Science.
Copyright (C) 1981
Robert N. Goldberg
ALL RIGHTS RESERVED
In Loving Memory of my Grandparents

Clara Reinhardt
Max and Esther Goldberg

who proudly watched my progress through graduate school
but did not live to see me graduate, and

In Honor of my Grandfathers

Benjamin Reinhardt
and
Kenneth Oakley

who did.
ABSTRACT OF THE THESIS

SOFTWARE DESIGN ISSUES
IN THE
ARCHITECTURE AND IMPLEMENTATION
OF
 DISTRIBUTED TEXT EDITORS

by Robert N. Goldberg

Thesis Director: Professor Robert L. Smith

This thesis analyzes the performance improvements achieved by distributing text editing software between a dedicated microprocessor and a timeshared host computer. Theoretical analysis of the problem coupled with detailed simulations of possible architectures, using data collected from over 15,000 actual editing sessions, provide strong evidence that simple, easily implemented schemes produce significant improvements of response time to user commands, communications bandwidth requirements, and host CPU utilization.

Consisting of an editor server program running on a host and a local editor running on a microprocessor, a distributed editor would a) provide general, flexible editing capabilities that are functionally indistinguishable from those of conventional video editors, b) provide the same average response on 1200 baud communications lines as conventional editors running at 9600 baud, c) isolate the user from most delays due to host timesharing pauses and communications network slowdowns, and d) reduce considerably the number of program activations and CPU cycles at the host, making distributed video editors consume fewer host resources.
than conventional line editors. The implementation of a prototype distributed editor based on the conclusions of the study demonstrates the practicality of implementing the required software on existing, inexpensive microprocessors such as those found in conventional terminals.
ACKNOWLEDGMENTS

I wish to give special thanks and recognition to my thesis advisor, Robert L. Smith, for providing continued encouragement during my career as a graduate student, especially during the thesis research stage, for his never ending battles with the University bureaucracy to provide students with a more reasonable computing, learning, and research environment, and for suggesting this thesis topic to me at a time in my career as a student when I had strong inclinations to abandon the Ph.D. program.

As director of the LCSR computing facility, Charles Hedrick not only provided the superb computing environment in which this work was accomplished but also helped me by instrumenting the TVEDIT and EMACS editors to collect data from user editing sessions. Roy Marantz also provided significant help with the EMACS instrumentation.

The other members of my thesis committee, Marvin Paull, Saul Levy, and Thomas Mitchell, were most helpful to me throughout my thesis research. I would especially like to thank them for reading my dissertation draft so thoroughly and so promptly.

Thanks go to Jeff Segal, who provided me with the Bliss code to make the Terak simulate a Visual 200 terminal, and to Jack Ostroff and Donald Smith for many helpful discussions concerning implementation of local Terak editors. Robert Schwanke provided assistance in helping me surmount the ignorance of Bliss. Donald Smith also implemented the RUTSIS document type in Scribe, with which this thesis was compiled.
I would also like to thank Casimir Kulikowski for helping provide financial support during my entire graduate career, first with a research assistantship in the medical applications area and later with a Special University Fellowship that enabled me to pursue my own research interests.

Many other people contributed indirectly to this thesis by providing a pleasant working environment and good friendship, including my fellow full time graduate students, the staff of the Rutgers LCSR DECsystem-20, and the DCS staff. Angela Dicorrado provided me with cheerful help and assistance that far exceeded her official duties, allowing me to concentrate on my research.

Finally, I would like to thank my dear wife, Judy, for helping me conceive the two major algorithms in this thesis, the *ideal prefetch algorithm* and the *minimal string difference algorithm*, and also for providing me with the kind of support that only a spouse who has lived through a Ph.D. herself could.
TABLE OF CONTENTS

1. Introduction 1
   1.1. Problem Overview 1
   1.2. Review of Previous Work 2
      1.2.1. Conventional Editors 2
      1.2.2. An Early Distributed Editor 3
      1.2.3. "Block Transfer" Distributed Editors 3
   1.3. Thesis 4
   1.4. Scope 4
   1.5. Relevance of Problem 5
   1.6. Plan of Thesis 8

2. Text Editors 10
   2.1. What is a Text Editor? 10
   2.2. Brief History 11
   2.3. Editing Primitives 15
   2.4. Logical Organization 16
      2.4.1. Differences Between Line Editors and Video Editors 17
      2.4.2. "Block Transfer" Terminals and Editors 18
   2.5. What are Distributed Text Editors? 20
   2.6. Design Choices 22
      2.6.1. Where should each editing primitive be processed? 22
      2.6.2. How should the processors communicate? 23
      2.6.3. What local hardware is required? 24
   2.7. Research Procedure 24

3. Transmission from Host to Local Processor 25
3.1. Motivation for Prefetching

3.2. The Bandwidth of an Editing Session

3.3. The Ideal Prefetch Scheme

3.4. Simulating Ideal Prefetching

3.5. Data Analysis and Reduction

3.6. A Class of Practical Text Fetch Schemes
   3.6.1. Conventional Demand Method - M0
   3.6.2. Unlimited Memory - M∞
   3.6.3. Unlimited Prefetch-Next - M∞ N∞
   3.6.4. Limited Prefetch-Next - Mn Nn
   3.6.5. Direction-Dependent Prefetch - M∞ Dn
   3.6.6. Limited Prefetch-Next-Previous - M∞ Nn Pn

3.7. Specialized Prefetching Schemes

3.8. Summary

4. Transmission from Local Processor to Host
   4.1. Motivation for Studying File Difference Representations
   4.2. A Class of Difference Problems
   4.3. Previously Solved Related Problems
   4.4. Minimal String Difference Encodings
      4.4.1. Problem Definition
      4.4.2. Definitions and Notation
      4.4.3. Algorithm
         4.4.3.1. Computing the Cost Array, L
         4.4.3.2. Procedure DeriveOpSeq
         4.4.3.3. Example
      4.4.4. Complexity Analysis
      4.4.5. Comments on the Algorithm
   4.5. Heuristic Approximations to Minimal Difference Encodings
      4.5.1. Common Head and Tail Method
      4.5.2. Optimization of User Commands
4.6. Need for Empirical Experiments

5. Empirical Study

5.1. Instrumentation of the Editors
  5.1.1. Statistics File Entries
  5.1.2. Instrumentation of MEDIT
  5.1.3. Instrumentation of TVEDIT
  5.1.4. Instrumentation of EMACS

5.2. General Editing Statistics

5.3. Programs used to Simulate Prefetching

5.4. Results of Prefetch Simulations
  5.4.1. MEDIT Sessions
  5.4.2. TVEDIT and EMACS Sessions
    5.4.2.1. Normal Editing
    5.4.2.2. Read-Only Editing
    5.4.2.3. Unnecessary Host Output

5.5. Host Interrupts

5.6. Analysis of File Change Descriptions

5.7. Discussion

6. The Prototype Distributed Editor: DISTED

6.1. Hardware and Software Used
  6.1.1. Hardware
  6.1.2. Software Characteristics

6.2. User Operation of the Prototype Editor

6.3. Overview of the Interprocessor Software Communication
  6.3.1. Packet Protocol
  6.3.2. Special Considerations

6.4. File Representation
  6.4.1. The Chosen Representation for Files
  6.4.2. Other Possible File Representations

6.5. The Host Editor Server
6.5.1. Editor Server Commands

6.5.2. Host Crash Recovery

6.6. The Local Editor

6.6.1. Software Organization

6.6.2. Prefetching

6.6.3. Local Editor Commands

6.6.4. Local Crash Recovery

6.7. Experience of Use

6.8. Possible Design Improvements

6.9. Conclusions

7. Conclusions and Future Work

7.1. Design Choices

7.1.1. Transmission of Text

7.1.2. Local Memory

7.1.3. Where should each editing primitive be processed?

7.1.4. How should the processors communicate?

7.2. Summary of Hardware Requirements

7.3. Summary of Software Requirements

7.4. Summary of Performance Enhancements

7.5. Future Research and Development
LIST OF FIGURES

Figure 1-1: Proposed Class of Distributed Computing Architectures 5

Figure 2-1: Logical Organization of a Text Editor 17

Figure 3-1: Commonly Available Communications Speeds, including the 26
time (secs) required to transmit a full 24 by 80 character screen

Figure 3-2: Algorithm to Simulate Ideal Prefetching 32

Figure 4-1: Algorithm for Computing the Entries of L[i,j,op] 51

Figure 4-2: Procedure to Derive a Minimal Cost OpSeq Given Array L 52

Figure 4-3: Example of the Minimum Cost Array L[i,j] for the strings 54
s = "ice cream" and t = "I scream". One of the derived
minimal difference encodings is delete(4), insert(\"i s\")

Figure 5-1: General Editing Statistics for 14,437 TVEDIT Sessions 66

Figure 5-2: General Editing Statistics for 1,100 EMACS Sessions 66

Figure 5-3: General Editing Statistics for 179 MEDIT Sessions 67

Figure 5-4: Distribution of Editing Session Lengths 68

Figure 5-5: Distribution of File Sizes Edited 69

Figure 5-6: Percent of File Displayed vs. File Size 70

Figure 5-7: Linked List Structure Used by the Prefetch Simulation 72
Program

Figure 5-8: Delay Ratio vs. Communications Speed (baud) using Ideal 74
Prefetch Simulation (179 MEDIT sessions)

Figure 5-9: Mean Delay Time in Seconds vs. Baud Rate 76

Figure 5-10: 1200 Baud Prefetch Simulations of 14,000 TVEDIT 79
Sessions
CHAPTER 1
INTRODUCTION

1.1. Problem Overview

Conventional text editors consist of a program that runs on a single host machine and utilizes the CPU, disk, and other resources of the host to process every character that the user types on the terminal. This arrangement has several serious drawbacks, namely:

- **slow response to user commands**, both as a consequence of timesharing delays and the limited rate at which characters are sent from the host to the terminal.

- **significant use of scarce host resources**, especially by video editors, including CPU cycles and I/O processing, and

- **communications cost**, especially when using packet-switched networks.

These undesirable aspects of editing files have become important issues to users of text editors on large mainframe systems.

The availability of inexpensive microprocessors encourages schemes that use them to help solve the problems. Thus the motivation for the distributed editor capabilities proposed in this work results from the insufficiency of existing editing facilities on timeshared computers as well as from the trend toward "personal" computers and workstations. Several groups at various institutions currently face the task of implementing distributed editors in response to overloaded host machines and high network communications costs, further underscoring the need for analysis of design decisions and architectures in this area.
1.2. Review of Previous Work

In reviewing the literature on the subject of text editing, one finds that little analysis of design decisions for distributed editing has been published. Indeed, not much literature is available on the architecture of conventional text editors, despite the large number of such editors that have been implemented. The few articles on the subject that have been published, such as [Hunt 78], generally provide a survey of several existing architectures rather than an analysis of the requirements of distributed editors and the design decisions faced by implementors.

1.2.1. Conventional Editors

Only recently have any serious efforts been made to document in the literature what has been done in the area of text editing. A recent ACM-sponsored conference was devoted to text editing, text processing systems, and general methods of handling text. The conference proceedings, [ACMTxtManipConf 81], contain descriptions of many text editors and text processors, including both production and experimental systems. For the most part, these papers describe actual implementations rather than studying design decisions. Recently, articles describing language-directed editors and systems, such as [Teitelbaum 81], have also appeared in the literature.

Most other published works on text editors and text editing, such as [Card 78, Card 80, Roberts 79, Fraser 80, Embly 81], deal with differences in user interfaces and other human factors as they impact user productivity and learning. These issues are orthogonal to the system architecture issues addressed in this thesis, as summarized in Section 1.3.
1.2.2. An Early Distributed Editor

The earliest known distributed editor, TVEDIT, was implemented at the Institute for Mathematical Studies in the Social Sciences, Stanford University, by John Prebus [Prebus 70] circa 1970. The editor used an IMLAC PDS-1 as a local processor which interacted with a DECSYSTEM-10 timeshared host. Significantly, editing commands were handled by the IMLAC in a manner transparent to the user. Insufficient memory in the IMLAC\(^1\) prevented the use of local file storage beyond the few lines displayed on the screen. The communications protocol between the two computers proved both slow and unreliable, despite the use of 9600 baud communications lines. When inexpensive video terminals became available, the distributed approach was abandoned to produce the current version of TVEDIT [Kanerva 80] which uses the host system to process every keystroke.

1.2.3. "Block Transfer" Distributed Editors

In the past, various ad hoc attempts have been made to implement distributed editors. For example, many of the major computer manufacturers support some kind of terminal data communications system designed to download some of the editing operations from the host CPU into local "intelligent terminals" that contain some editing capabilities. The interface between the host and terminal is usually rather limited. Typically the user requests the host computer to transmit some text to the terminal, edits the text locally, and then retransmits the edited text back to the host. The many drawbacks of this editing architecture will be discussed in Chapter 2.

\(^1\)A mere 4K words for character generators, program, and text!
1.3. Thesis

The editing problems related to host loading, user response time, and low bandwidth communications can be solved to a much greater degree than they have in the past by a suitable distributed computing architecture using a dedicated local microprocessor with a reasonably large random access memory, interfaced with a remote host over a conventional data communications line that is slow relative to the speed of either processor (see Figure 1-1). The distribution of the editor between the two processors can be made transparent to the user, except that the "terminal" will appear to be connected to a higher speed line that it actually is, and response to all local editing changes will always be immediate regardless of host system load. Furthermore, these results can be achieved without any loss of flexibility in editor functionality.

This thesis consists of an exposition and analysis of the alternatives available in the design of possible architectures, mathematical and algorithmic analyses of the alternatives in terms of the overall goals of editing, and experimental determinations of parameters that indicate the relative importance of various theoretical issues as they affect actual editing sessions. This dissertation provides the analytical groundwork for the implementation of future state-of-the-art distributed text editing systems, rather than advocating a single solution.

1.4. Scope

In dealing with the problem of text editing, this work addresses the communications interface between the local and host processors, including the strategy for transmission of text and commands between the processors, and the communication primitives used to accomplish the interaction. The local and host systems must interact with each other extremely closely so that they appear to the user to be a single editing program. To accomplish this goal, protocols for the interaction that describe the communication primitives must be defined. Design of the protocols, in turn, depends on an analysis of communication needs of editors.
Figure 1-1: Proposed Class of Distributed Computing Architectures

This work does not deal with the design of editor command languages describing the keystrokes to perform various editing primitives, nor will it define the set of possible primitives. In particular, no restrictions on the command language will be imposed. The editing primitives "insert", "delete", and "replace" are sufficient to perform arbitrary changes to a text file, but the proposed architecture must handle all reasonable editing primitives such as "justify this paragraph" and others available in existing general text editors such as SOS [Weihrauch 70], MEDIT [Goldberg 81], EMACS [Stallman 80], and TVEDIT [Kanerva 80]. Although the work emphasizes screen-oriented editors, it will be shown that the general results apply also to line-oriented hard copy editors.

1.5. Relevance of Problem

The use of interactive text processing systems, both in computer applications areas and in business and office applications, is becoming more and more widespread. At many computer installations, text editing comprises a significant fraction of the total machine usage. As the popularity of computer text processing
grows, users of mainframe systems face increasingly loaded systems that give poor response to highly interactive tasks such as editing. Together with the decreasing prices of microcomputers, this trend suggests a move away from systems in which a large mainframe computer is shared by many users for all of their computing needs, and a move toward small individual machines.

Several important problems stand in the way of universal abandonment of timeshared mainframe systems. While the cost of microcomputers has fallen considerably, economy of scale still applies to the cost of peripheral devices such as disk drives. Most organizations still cannot afford to provide each mainframe user with a dedicated microcomputer workstation complete with peripherals, particularly one that can match the capacity of the timeshared system. On the other hand, supplying new terminals to connect to the existing system is relatively inexpensive. Even if the cost of stand-alone workstations were no object, large systems would still be useful as centers for electronic message communication and data sharing among users. For these reasons, data will probably continue to be stored in central "host" computers to which single-user machines can connect. Thus, editing files that reside at a central host will remain an issue.

Despite the rapidly dropping cost of microcomputers, data communications lines will continue to be a scarce resource, especially if many applications demand sufficient bandwidth to match the speed inherent in these computers. Frazer and Branscomb [Frazer 79, Branscomb 79] estimate that while the cost of memory and logic hardware will decrease at the rate of about 28% per year during the 1980's, the cost of communications will drop at only 11% per year. While shortages of CPU power on timeshared hosts (the von Neumann bottleneck) can be avoided, at least for text editing, by employing one processor per user, the only practical solution to the data communications bottleneck appears to be avoiding unnecessary

---

2The projected drop in communications costs is based on the use of fiber optics and satellite technologies.
communications as much as possible. Thus this work will focus heavily on minimizing the need for high information transfer rates between the host and local processor without affecting system versatility.

It should be pointed out that even if infinite bandwidth communications lines existed between a host timesharing system and its users, the host itself could not keep up with the demands of many highly I/O-bound jobs. Thus a distributed editing scheme that limited the amount of I/O would still be desirable even if fiber optics communications were suddenly installed in every home and office in the world.

More realistically, of course, it will probably be many years before the newest communications technology replaces every copper wire. In particular, the user of home terminals connected to a distant host appears limited for the present to the bandwidth of a voice-grade telephone line (e.g. 300 to 1200 baud using commonly available modems). Judging by the explosion of data communications that has already taken place, future communications costs will probably continue to be a function of the bandwidth. 3

During the course of the research presented in this dissertation, I had numerous occasions to discuss the thesis with experts in the field of text editing. Most of the comments on the feasibility of distributed editing fell into the following categories:

• The research is not needed because distributed editing will obviously work!

• The research is not needed because distributed editing will not work well enough to justify using it in an editor.

• Distributed editing is good in theory, but in practice it is too complex to implement reliably.

3 Claims by some participants in the fiber optics "revolution" that this technology will provide virtually free communication remind one of the claims once made about nuclear power plants, namely that they would provide essentially free electric power.
• Distributed editing might work for simple editors, but not for more powerful editors such as EMACS.

• To get good performance, the editor will have to maintain a complex model of the user's editing behavior so that it can guess which parts of the file need to be present locally.

I took the fact that there was such disagreement by experts on the subject, coupled with the impact that a successful design could have on editing performance and cost, as an indicator that the research was, in fact, needed.

1.6. Plan of Thesis

This chapter has described problems associated with conventional editing on timeshared systems and has suggested the exploration of methods for downloading some or all of the editing to a dedicated microprocessor that communicates with the host system. Motivation for work in this area has been presented as well.

Chapter 2 discusses text editors in general, including the role they play in the computing milieu, the kinds of file manipulation primitives they provide, and the way they interact with the user. The notion of distributed editing is defined, problems with previous attempts to provide it are discussed in detail, and the space of design decisions faced by prospective implementors is explored.

Communications between the two processors is crucial. During an editing session, parts of the file must be transmitted from the host to the local system to be viewed and edited by the user. Chapter 3 considers ways to improve the effective bandwidth of the communications line using local memory and various schemes for guessing in advance which parts of the file the user will want to edit. Ways of simulating various schemes using data from specially instrumented conventional editors are devised, and the ideal prefetch algorithm is introduced as a benchmark for measuring the performance of potential schemes. A class of simple but realizable schemes is defined.
Chapter 4 considers communication of file changes from the local processor to the host. An algorithm to determine the minimal difference between two strings with respect to a set of editing primitives is developed, both as a potential procedure within an actual distributed editor and as a benchmark against which simpler heuristic methods can be judged.

Chapter 5 presents empirical results based on 15,000 user editing sessions, obtained by instrumenting several conventional editors to produce detailed session data files. The data collected from the editors is summarized, and the results of simulations of the prefetch methods discussed in Chapter 3 and the string difference methods introduced in Chapter 4 are presented.

Chapter 6 describes the implementation and operation of a prototype distributed editor, based on the results obtained in Chapter 5.

Chapter 7 reexamines the space of design choices for implementing distributed editors, based on the simulation results and data summarized in Chapter 5 as well as on experience with the prototype editor discussed in Chapter 6. The hardware and software requirements for an implementation and the performance enhancements provided by the distributed editor are summarized. Finally, potential areas of future research and development in distributed editing are discussed.
CHAPTER 2

TEXT EDITORS

This chapter introduces the notion of a general purpose text editor, gives a brief history of the development of interactive text editors, describes the logical software modules of a conventional system, and introduces the concept of distributed text editing. Design decisions inherent in the construction of distributed text editors are considered, motivating the analysis and empirical work done in the remainder of this thesis.

2.1. What is a Text Editor?

A text editor is a computer program used to create and modify text files stored in the computer. General purpose text editors are useful for creating and modifying files consisting of any arbitrary sequence of characters including computer programs, English language documents, and other data.

The basic job of a text editor is to provide the user with editing primitives that operate on a virtual file that can be manipulated in ways that actual physical storage media cannot. The basic primitives of insertion and deletion of characters are sufficient to perform arbitrary changes to a file and generally form the kernel of primitives provided by text editors. Neither of these operations is supported by typical storage media such as disk or tape storage devices so the editor must simulate them.

On many computer systems, the writing and modification of programs, documents, reports, and the like dominates all other activity of the user community.
Some informal measurements made on the research computing facility at Rutgers in the past year support the following claims:

- Most of the system “connect time” is devoted to text editing.
- Program activations to handle character input happen at an enormous frequency.
- The text editors are the most frequently used class of programs.
- More file space is devoted to the storage of documents than programs.

Informal observation at other central computer facilities suggests that these measurements are typical.

Text editors are of greater importance than generally recognized. They are arguably the most important class of programs in existence, because of their central role in any application, and because of the amount of time people spend using them. Computer Scientists have studied the design of operating systems and programming languages in great detail, but have generally regarded text editors merely as utility programs. In reviewing the literature on the subject, one finds that very little has been written beyond broad surveys of existing software, and almost no analysis has been done of the design decisions faced by the implementor of a text editing system.

2.2. Brief History

The development of text editors is intertwined with the development of early interactive timeshared computers. In the batch systems that predated timesharing, the user communicated with the computer by punched cards that were prepared off-line at a keypunch machine. With the advent of interactive timesharing systems, the medium of interaction between user and computer switched from punched cards to the Teletype\(^4\), a typewriter-like device that converts characters typed on its keyboard into sequences of binary pulses that can be read by the computer, and

\(^4\)Trademark of Teletype Corporation
which can type the computer's response on paper, producing hard copy. The Teletype was originally developed as a written communications device for use by news wire services, but it quickly became the standard method of communications with computers.

Early text editors were developed to solve a problem of compatibility between existing batch programs and interactive methods of user access to the computer. A typical program designed for batch use expected its input data to be presented all at once as a series of correctly punched cards. While it was easy enough for the operating system to translate a line of characters typed on a Teletype into a card image that could be passed on to the program, some method was needed to store these lines as a group, analogous to the way the user could store a deck of punched cards for later reuse.

This need gave rise to the notion of a file, a named group of card images or lines that could be stored and easily retrieved by the computer, typically on a magnetic disk. Also, it was necessary to provide a way to enter and correct program input in advance, analogous to the way cards are punched, examined, inserted, deleted, and corrected by the user before they are passed to a batch program. Thus, the first interactive text editing programs were created, primarily to provide the user with an interface to the computer file system. Batch programs could read files created through a text editor similarly to the way they would read a deck of cards.

The mode of operation of early editors was limited by the method and speed of communication with the user, and by the experience of the card-oriented implementors who wrote them. Typically, the user would type a command line to the editor, which would then respond by making the requested changes to the file. There was invariably a command that caused the editor to type the current contents of the file on the terminal. Other commands allowed the user to delete, insert,
retype, or rearrange a line or group of adjacent lines. These basic primitives duplicated what could be done on a keypunch machine with cards. Other commands were found useful as well, such as a command to search for a given string of characters in the file, and a command to change all occurrences of one string to another. Some editors provided commands to let the user edit individual characters within a line, rather than requiring that the entire line be retyped to correct an error.

On computer systems that supported full duplex terminals, this capability could be implemented by having the user type commands that were not echoed directly, but whose effects were seen. As people learned more about what kinds of editing primitives were useful, many additional special purpose commands were added to increase the power of these line-oriented and Teletype-oriented text editors. Until recently, most text editors were of this type, that is, general purpose text editors designed to be used on Teletype-like terminals.

More recent developments in text editing have taken two orthogonal paths away from the general purpose editors described above. One is the idea of structure-oriented and language directed editors with primitives that correspond to a particular programming language or data structure. Early examples of this kind of editor include various LISP editors designed to edit list structures, and to a limited extent the standard BASIC environment which includes an editor that translates input lines into an internal “compiled” representation\(^5\) that can be interpreted efficiently. Current experimental projects of this kind include editors that automatically detect syntax errors in PASCAL or PL/1 programs as they are typed in by the user, and editing environments that perform incremental compilation on source programs as they are edited and show control flow by cursor movement on the program source during execution and debugging [Teitelbaum 81].

A second, orthogonal direction in text editing involves the use of video

\(^5\)This approach is used on some systems
terminals whose screens constantly show the current edited state of the file. The earliest known system of this kind was implemented on a PDP-1 computer at Stanford University [McCarthy 67]. More recently, many video terminals and editors that support them have become available and popular. These editors require close interaction between the program and the terminal: each character typed on the keyboard is processed by the editor and each screen operation that results is sent explicitly to the terminal by the program. Furthermore, these editors generally require higher rates of character transmission between the computer and the terminal for effective editing. Most editors of this kind work with screens that contain about 24 lines with a maximum length of 80 characters each, and the choice of unique characters is limited to about 100.

Current experimental projects with video screen editing include work with high resolution bit mapped screens that can display a large variety of characters and fonts, including multiple alphabets, italics, bold faced characters, and a variety of sizes. These systems are quite useful for preparing typeset documents since the user can see a representation of the final output form of the document right on the terminal screen. Other projects involve editors for diagrams, pictures, and other graphics forms merged with text. There is also work in progress on structure editors for English language documents in conjunction with a video display of the text. The reader is referred to [ACMTextManipConf 81] for a comprehensive collection of current research on these editors.

Despite the emphasis on special purpose editing languages in current research, the notion of a general purpose text editor has not been abandoned. There are several reasons for the continued popularity of the general purpose text editor. It is easy to learn and its primitives correspond nicely to human visual perception of written language. It is flexible in its ability to edit many different kinds of documents and files. Finally, it presents a single command language that need be learned only once.
In contrast, in a world of special purpose editors it is necessary to learn many different and possibly inconsistent command languages, one for each different editing task. Unless one particular special kind of task is performed very frequently, it may not be worth the user's effort to learn the special command language. Having a different editor for each computer language raises serious manpower issues from an implementor's point of view as well. Current work on general purpose editors includes research on ways to provide extensibility and customizability of the basic editor so that it can be adapted to special purpose tasks while retaining the same basic command primitives ([Stallman 80, Stallman 81]). Other work focuses on integrating a general purpose editor and a screen manager to provide multiprocess control and to allow flexible editing of process typescripts ([Lantz 79]).

2.3. Editing Primitives

The virtual file model used by most general purpose editors may be thought of as a continuous string of atomic objects. For concreteness, the atoms will usually be referred to as "characters" but they could also be a textual representation of binary words of computer memory, for example. Record-oriented and line-oriented file models are a special case of the continuous string model in that they impose restrictions on the allowable sequence of characters that the editor may produce. Special record structures or assumptions about the text being edited will not be assumed in the interest of generality of results.

Given the file model, an editor provides primitives for manipulating the characters in the file. From the user's point of view, the editor performs various functions in response to commands. Different editors accept different commands and perform somewhat different actions; however, all editors provide a similar set of underlying text manipulation primitives, that fall roughly into three classes.
- Local editing primitives
- Global search
- Global changes

Local editing primitives include random positioning on the currently displayed portion of the file and the portions adjacent to it, and the primitive operations of insertion, deletion, and replacement of atomic objects (characters) in the file. The ability to display the lines in the context of the current location within the file is included implicitly. Notice that these primitives can be used by the editor to implement higher level commands such as “justify the current screen of text” or “search for x on the screen”.

Global search primitives include the ability to move to a specified line or character directly without displaying the text between the present and desired locations, either by specifying the position of the line within the file (e.g., line and page number) or by specifying character patterns for which to search. These primitives imply the ability to search the file without transmitting it over the communications line.

Global change primitives include any commands that perform global changes to the file, such as global text justification, global formatting of an Algol program, and the ability to copy and transfer large portions of text from one part of the file to another or between files. Global changes are generally not displayed because they perform their action on most or all of the file.

2.4. Logical Organization

A text editor typically consists of a user command decoder that reads commands typed or otherwise input by the user; a module of editing primitives that performs the operations corresponding to the editor commands; a module to simulate the virtual file; and a module that performs output to the user’s terminal.
or display device. This organization is shown in Figure 2-1. In conventional text editors, these functions are all contained within a program running on a single computer.

![Logical Organization of a Text Editor](image.png)

**Figure 2-1:** Logical Organization of a Text Editor

2.4.1. Differences Between Line Editors and Video Editors

Line editors typically accept input a line at a time, using a host operating system call that reads a complete line from the terminal and allows the user to correct mistakes (e.g., with the rubout key). On full duplex systems, each character typed is echoed by the host at the operating system level, which avoids scheduling the editor each time the user hits a key. Most video text editors process every user keystroke as it is made because the line input routines provided by the operating system are not specialized enough to perform adequate screen manipulation functions. As a result, video editors must be rescheduled by the timesharing monitor on every keystroke, thus consuming more system resources.
than line editors and suffering more dramatic performance degradation when the
timeshared computer on which they run becomes heavily loaded. They also tend to
send more characters to the terminal during a typical editing session than non-video
editors, and they require higher terminal communications rates for effective editing.

Many people justify the extra host resources consumed by video text editors
by the increase in productivity using them. At reasonably fast communications
speeds (e.g., at least 1200 baud) and on systems that are not heavily loaded, a good
video editor allows the user to concentrate on the text without mentally keeping
track of the most recent changes made and what the current location in the file is.
However, as the system on which the editor runs becomes more loaded, the
productivity of the user drops appreciably. Also, using these editors over
telecommunications networks becomes a problem due to delays in the transmission
of characters as long as several seconds. Furthermore, some systems do not
support the required terminal handling to provide flexible video editing (e.g., half
duplex systems).

2.4.2. "Block Transfer" Terminals and Editors

So-called "smart terminals" have been proposed by some manufacturers and
vendors as a solution to the inefficiencies introduced by video editors. These
terminals typically consist of a keyboard, a display screen, and a microprocessor
that provides some local editing capabilities on the screen of the terminal. Most of
the keyboard keys change what is on the screen without transmitting to the host
directly. When the user has edited the screen as desired using the terminal's editing
capability, he presses a special key to transmit the current screen back to the
program running on the host. To edit a text file, the user invokes a text editor on
the host that knows how to share the editing responsibility with the terminal.

This method of text editing can considerably reduce the number of program
activations at the host computer because user keystrokes required to perform
simple editing operations are handled locally (with immediate response) and do not interrupt the host. However, this method may not reduce the total amount of I/O that must be done during an editing session because the entire screen of text must be transmitted back after the user has made local changes to it. In fact, this architecture may be less acceptable for very low bandwidth connections than conventional terminals and editors, especially if relatively few changes are made compared to the total number of characters in the transmitted text.

Using general purpose "block mode" terminals for editing has many disadvantages from the human engineering point of view as well. We list some of the specific drawbacks below; some are caused by "bookkeeping" information that the user must keep in mind while using the terminal; others are due to the restricted nature of the terminal's local editing capabilities that do not match a good text editor.

- When sending lines of the file to a general purpose video terminal, the editor must include line markers along with the actual text so that it can later identify which screen lines correspond to which lines in the file. If the line markers of some lines are deleted accidentally by the user, the editor program will become confused about the identity of those lines when the screen is sent back to the host and consequently may not update the file as the user intended. On some terminals, the user must be very careful not to violate the conventions used by the editor to mark the lines. Other terminals support special constructs such as "protected fields" that make it more difficult, but not impossible, for the user to violate the conventions.

- If the user is responsible for hitting a key to retransmit the screen when editing changes have been made, he must remember to do so. Because retransmission may take time, users will try to optimize their time by avoiding retransmission when they have not changed the screen. If they make a change and forget to send the screen, the change will not be made in the file at the host.

- The restricted size of the screen is a source of trouble. When characters or lines are inserted, existing characters may scroll off the right hand side of the line or off the bottom of the screen. These characters will never be seen by the host so they will disappear from the file. For example, whenever a line is inserted on a full screen of

---

6 Of course, one could design a terminal that knows a particular editor's conventions and thus can prevent the user from destroying line markers, but this special terminal would no longer be general purpose.
text, the bottom line will disappear. If that line was changed since it was received from the host, the changes will be lost because the line will not be sent back. As another example, whenever a phrase is added to a paragraph, the line the phrase is on will grow too large and some characters will be lost at the right. The user must then delete the characters in partially lost words and insert by hand all of the lost words at the beginning of the next line. The next line will often be too long, so the process must be repeated until the end of the paragraph is reached.

- Currently available “smart terminals” do not provide as rich a set of commands as a good text editor. They typically provide keys to insert and delete characters, lines, and screens, but no keys to act on words or other logical constituents of the file. The ability to search the screen locally is typically not provided. Repeat counts for commands cannot generally be given; instead, the user must hold down a function key and let it “auto-repeat.” This process can be tedious when moving diagonally from one end of the screen to the other.

- If the user accidentally hits a key that causes a change to the screen, such as deleting the current line or screen of text, there is typically no way to undo the mistaken command. In a well-designed text editor, the keystrokes required to delete a large amount of text are chosen so as to prevent it from happening accidentally, and an “undo” command is usually provided for deletions.

- If the host system crashes, the user may continue editing the current screen for a long time before discovering that his editing is not going to matter.

To avoid these pitfalls, the user must exercise extreme care. Users of such editing systems must keep in mind what the terminal is doing, what the host system is doing, and how the two systems interact.

2.5. What are Distributed Text Editors?

A distributed editor is an editor whose logical functions are distributed among more than one computer system, with each system handling a non-trivial task and with some possible overlap or duplication of tasks among the systems. Many possible arrangements of computer systems motivate the use of distributed editors, from simple two processor systems to distributed file systems spread out among many processors hooked together in a network. The problems that motivate distributed editing include:

- Host overloading: A significant amount of CPU time of the host is
consumed in processing editing commands and input text for each editing "job". When the host machine is a timesharing system with more than one user, the von Neumann bottleneck of the host CPU becomes a problem, especially since each character typed by the user and each group of characters output by the editor requires a context switch from user mode to operating-system mode. It is not at all atypical for large interactive timesharing systems to spend 50% of their CPU cycles doing I/O to user terminals during editing. The time spent on terminal I/O steals CPU cycles from all of the users on the system, affecting not only editing but other programs and tasks on the host computer as well.

- **Slow response:** Each time the user wishes to see a portion of the file being edited, the editor must transmit part of the file to the user’s terminal. Because the bandwidth of the connection between the terminal and the host system is limited, there is a delay while the portion of the file is transmitted. The delay can significantly impede user productivity, especially with relatively low bandwidth phone line connections. When complex network arrangements are used to access a computer that is many miles away, the delay can be both slow and unpredictable.

- **Communications cost:** In some cases, the cost of transmitting single character "packets" over a network can be significantly greater than the cost of transmitting the same characters in fewer but larger packets. Conventional editors and other interactive host programs require that each user keystroke be sent as a separate packet over the network. A distributed editor can take advantage of the local processor to handle many user keystrokes locally, storing up the changes made to the file and transmitting them over the network in large but relatively infrequent packets.

The use of local editing features of "smart terminals" by some existing text editors comprises a rudimentary form of distributed editing that was motivated partly by the above issues, but largely by the inadequacies of half duplex host systems. These systems suffer from suboptimal use of the communications line (especially at low speeds) in addition to the human engineering inadequacies outlined earlier.
2.6. Design Choices

To keep the space of possible system architectures manageable, this work concentrates on the distribution of editing functionality between two systems, a timeshared host and a dedicated local system, although some of the conclusions will apply to very generally distributed systems as well. Therefore, the following scenario will guide the investigation:

- The file resides at a "host" computer and must be updated there when the editing session is over.
- The user performs editing at a dedicated "local" processor (possibly just a terminal).
- The "local" and "host" systems are linked via a communications line that has a low bandwidth compared to the speed of either processor (e.g., 300 to 9600 baud) but high compared to the rate at which the user types.

Given this scenario, various decisions must be made concerning the implementation of the editor. The major decisions to be made include the division of labor between the two processors, the methods of communication to use, and the hardware needed for the local processor.

2.6.1. Where should each editing primitive be processed?

Conventional editors use the host processor to interpret and process all primitives. A distributed editor may process primitives either at the host or at the local processor. Which editing primitives should be handled by the host and which locally? Should some be handled in both places? Should the editor front end (user interface) be located in the local processor, in the host, or split between the two? These issues are discussed in Chapters 6 and 7.
2.6.2. How should the processors communicate?

What communications protocols should be used between the host and local systems? The protocol should be reliable and provide error recovery, but at the same time it must be reasonable compact. What problems will arise when interfacing to existing mainframe systems? How can these problems be overcome? Protocol issues are examined in detail in Chapter 6.

How should the file be transferred from the host to the local system for viewing? The entire file can be transmitted at the beginning of the editing session, or sections of the file can be transmitted on demand, as they are needed. What size sections should be used? Methods of guessing in advance which sections of the file will be needed can be used. If so, which methods will work well? Should adaptive methods that take user behavior and file type into account be used? What adaptive parameters are the most useful? The transmission of the file from the host to the local system is considered in detail in Chapter 3, and simulation results are presented in Chapter 5.

How should changes made to the file locally be transmitted back to the host? The entire file can be transmitted back to the host at the end of the editing session, or parts of the file (e.g. individual screens) can be transmitted back incrementally during the session. Incremental changes can be stored up or sent as generated. Alternatively, the user's actual keystrokes can be transmitted back, either as they occur or in batches. What size batches should be used? Is there some way to determine the difference between the original file and the changed version automatically and efficiently? Can the most concise difference be generated so as to minimize the use of the communications channel? Methods for describing local editing changes to the host are presented in Chapter 4, and simulation results are presented in Chapter 5.
2.6.3. What local hardware is required?

How much CPU power is required in the local processor? How much memory should the local processor have? Hard copy terminals have no extra memory for file storage. Simple video terminals have memory for only the currently displayed screenful of text. What is the smallest amount of memory required to get good performance out of a distributed editor system? Does it help to have enough memory to fit the largest possible file? Would it help to have enough memory for several files? If there is not enough local memory to fit the file, some scheme for reusing local memory may have to be implemented. What schemes will work well? What can be gained by adding local mass storage media? The memory requirements of actual editing sessions are summarized in Chapter 5. These issues are discussed in detail in Chapters 6 and 7.

2.7. Research Procedure

Because the ultimate justification of a distributed editor lies in its performance, and performance factors such as user response time and host program activations depend critically on the nature of communications between the host and local systems, the next few chapters will concentrate on communications. Chapter 3 will analyze transmission of the file from the host to the local system, and Chapter 4 will analyze transmission of file changes from the local system to the host. The discourse in these two chapters will demonstrate that specific empirical experiments of typical editing are required to make meaningful conclusions about the effectiveness of various design decisions. The empirical experiments will be described in Chapter 5. Many of the other design decisions raised in the current chapter depend on the results presented in Chapter 5 and on the choice of communications strategies.
CHAPTER 3
TRANSMISSION FROM HOST TO LOCAL PROCESSOR

During an editing session, parts of the file must be sent to the local processor for viewing by the user. The simplest way to implement the transmission of the file is for the necessary parts of the file to be transmitted as needed for viewing. We will call this method demand fetching of the file. This method is used by conventional text editors. When slow communications lines and demand fetching are employed, the user will experience a delay each time he wants to view a different part of the file.

To avoid the delay, the local processor can remember every screenful displayed during the current session so that if the user returns to one of these screens it will not have to be fetched from the host. However, there will still be a delay when the user asks to view a previously unseen part of the file. To avoid delay in this case, the local processor might prefetch (fetch in advance) parts of the file while the user is performing local editing on the current screen of the file and the communications channel is not being used.

This chapter motivates the basic notion of "prefetching" as it applies to distributed text editor design. The problem of measuring the utility of a prefetch method is considered, and the notion of the ideal prefetch method is introduced as a benchmark. A class of simple but general prefetch schemes is defined and the most interesting examples of this class are described. Chapter 5 describes the collection of empirical data from in vivo conventional editing sessions and the simulations of various prefetching schemes using this data.
3.1. Motivation for Prefetching

An important question to consider when designing distributed editing architectures is "What bandwidth of the communications line will suffice to ensure fast response of the system to user commands?". The answer depends on many design and use factors, such as the way the editing task is divided between the host and local processor, how much of the file is stored locally, what prefetching schemes are used, and what kinds of demands the user places on the system by the editing command sequences he chooses to use and the time that elapses between commands. Figure 3-1 summarizes some of the commonly available communications speeds in use today.

<table>
<thead>
<tr>
<th>Baud</th>
<th>Chars/Sec</th>
<th>Time</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>110</td>
<td>10</td>
<td>192</td>
<td>Old Teletype speed, not used much</td>
</tr>
<tr>
<td>300</td>
<td>30</td>
<td>64</td>
<td>Most common phone line speed</td>
</tr>
<tr>
<td>1200</td>
<td>120</td>
<td>16</td>
<td>Highest commonly used phone line speed</td>
</tr>
<tr>
<td>2400</td>
<td>240</td>
<td>8</td>
<td>Currently requires high quality phone lines</td>
</tr>
<tr>
<td>4800</td>
<td>480</td>
<td>4</td>
<td>Currently requires direct wiring to host</td>
</tr>
<tr>
<td>9600</td>
<td>960</td>
<td>2</td>
<td>Highest popular timesharing speed</td>
</tr>
<tr>
<td>19200</td>
<td>1920</td>
<td>1</td>
<td>Some dedicated systems</td>
</tr>
</tbody>
</table>

Figure 3-1: Commonly Available Communications Speeds, including the time (secs) required to transmit a full 24 by 80 character screen

Consider an application in which the communications line bandwidth must be kept as low as possible and local memory is plentiful. Some preliminary empirical results collected from actual editing indicated that in typical editing session on 9600 baud lines, the bandwidth that would be required if the I/O were uniformly spread over the entire editing session is about 12 characters per second using a hard copy editor, and about 40 characters per second using a video editor. However, I/O activity during editing (using existing architectures) occurs in "bursts" that are separated by periods of relative inactivity. Thus one experiences delays when editing using low bandwidths while needed portions of the file are transmitted from the host to the local processor for viewing, whether it be the first or subsequent time that a particular portion of the file is viewed. For example, at
1200 baud, a typical "fast" communications speed by phone, it takes 16 seconds to fill a 24 line by 80 character screen (see Figure 3-1).

Can these delays be avoided using some prefetching scheme? While the 12 and 40 cps averages suggests the possibility, the answer depends on various characteristics of typical editing, such as how closely spaced the bursts are, and the locality of editing within a file. In the worst case situation, all of the demand for communications occurs during the initial part of the editing session (e.g., the user requests that the entire file be displayed immediately with no delay for reading or editing). In this case, no prefetching scheme will avoid the delay in initially transmitting the file over a low bandwidth connection. In the ideal case, the demand for transmission is spread uniformly over the editing session and the required transmission rate is thus about 12 cps for hard copy editing and about 40 cps for video editing. The distribution of bursts in typical editing falls somewhere between these two extremes. Thus we cannot answer the proposed question using purely analytical techniques, but rather we must collect empirical measurements from actual editing sessions and characterize the behavior of prefetching schemes given those measurements.

3.2. The Bandwidth of an Editing Session

Given a typical editing session, how can one characterize the distribution of bursts of I/O activity? For this experiment, we will collect data about editing using a conventional architecture with a fast (e.g. 9600 baud) communications line between a host system and a terminal. Specifically, the local memory consists solely of the terminal's screen and no prefetching is done. To analyze the distribution of data transmission from the host system to the terminal, we will collect the time of day (e.g. in milliseconds) of transmission of each contiguous "chunk" of text that is sent.\footnote{Later on we will consider transmission from the local to host system, but for now we will focus on this direction since it demands the greatest bandwidth.}
Given this information, how can we determine the performance of an architecture that uses prefetching? In particular, how much can we reduce the bandwidth without unduly increasing the delay experienced by the user over what he experiences using a conventional editor and high speed communications?

The information that is collected in the above experiment cannot easily be analyzed without simulating an editing session at a particular communications rate. For example, consider the following segment of data collected during an actual session at 9600 baud (960 cps):

<table>
<thead>
<tr>
<th>Characters Requested</th>
<th>Time When Requested (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>.</td>
<td></td>
</tr>
<tr>
<td>.</td>
<td></td>
</tr>
<tr>
<td>.</td>
<td></td>
</tr>
<tr>
<td>1:2000</td>
<td>20.0</td>
</tr>
<tr>
<td>2001:4000</td>
<td>40.0</td>
</tr>
<tr>
<td>.</td>
<td></td>
</tr>
</tbody>
</table>

Twenty seconds after the editing session starts, the user issues a command that requires the first 2000 characters in the file to be displayed on the terminal. Twenty seconds later, the next 2000 characters must be displayed. The above segment of data without the prior history is insufficient to predict the delay that the same user would have experienced at the slower rate of 1000 baud (100 cps) because we do not know if characters 1 through 2000 would already be present in the local processor at the time they were requested by the user.

Suppose that characters 1 through 2000 are already present in local memory 20 seconds after the start of the editing session, either because they were part of an earlier screenful of text or because they were prefetched using some scheme. The editor could display them on the screen immediately and use the next 20 seconds to prefetch characters 2001 through 4000. When the user moved the window down at the 40 second mark, those characters would now be present in
local memory. Thus, no delay would have been experienced during the above segment. On the other hand, if none of the characters 1 through 2000 had been present in the local processor at time = 20 seconds then the user would have experienced a total delay of 40 seconds for the above session segment (20 seconds for each group of 2000 characters).

Thus to determine the delay times for a particular editing session given the above data, it is necessary to start at the beginning of a session and then simulate the entire session assuming a particular rate of transmission and a particular prefetch scheme. At each point in the simulation, the knowledge of what text has been stored locally and what text this scheme will prefetch may be used to determine the communications time (delay) for displaying each group of characters.

3.3. The Ideal Prefetch Scheme

To evaluate the effectiveness of distributed editing architectures that prefetch text from the host, one could invent and implement many possible and different prefetching schemes. The best of these schemes could then be tested against traditional editing to measure the effectiveness of prefetching. However, this approach would lead to conclusions about the particular schemes chosen rather than a general statement about prefetching. For example, if the chosen schemes did not work well, it would not necessarily mean that all conceivable prefetching schemes would fail. Even if the chosen schemes worked fairly well, it would be useful to know how hard we should look to come up with better ones. These considerations motivate the following definition.

The ideal prefetch algorithm is a scheme for prefetching text that uses any “spare time” on the communications line to prefetch the next characters that will be needed from the host.

The ideal prefetch algorithm requires perfect knowledge of the future and is thus not feasible in practice. Its value derives from the property that it prefetches
precisely the needed text in the correct order and as soon as possible, guaranteeing that it will perform at least as well as any other prefetching method. Furthermore, it is easily simulated given data from editing sessions. Thus the ideal method serves as a metric against which all feasible methods may be compared for utility. This notion of an ideal but infeasible algorithm has previously been applied in the study of operating systems to benchmark practical page replacement schemes for virtual memory management [Shaw 74].

The existence of the ideal prefetch algorithm suggests an experiment for which data of the form described above are collected from actual editing sessions and used to simulate ideal prefetching. If it performs well then further simulations can be carried out using feasible methods, with the goal of finding a method that works almost as well as the theoretical ideal. If the ideal method performs poorly, on the other hand, then no feasible method can possibly work well given the typical editing behavior represented by the simulated sessions and the available communications bandwidth. In this case, we know not to bother looking. The following section describes the experiment in more detail.

3.4. Simulating Ideal Prefetching

Although the ideal prefetch algorithm requires precise a priori knowledge of the characters that will be referenced during an editing session and is thus not realizable in practice, one may easily simulate its effectiveness at particular transmission rates given the character reference data for a conventional editing session conducted at a much faster rate. The faster rate of the conventional session gives some assurance that the dominant factor in the pause between data entries corresponds to user "think time" rather than idle time while the user's screen is updated. We wish to determine how much communications delay would have been experienced by the user if the same editing session had been conducted at a slower communications rate but using ideal prefetching and local memory.

---

8The simulation method will be described in Section 3.4.
Let groups be an ordered list of contiguous character bound-pairs, and group\textsubscript{i} be a single bound pair (e.g. 1:2000). Let group.time\textsubscript{i} be the time at which the user requested that the characters in the bound pair be displayed. Let tcs be the time required to transmit one character over the communications line at the simulated rate, and tca be the analogous time for the actual session rate. The basic event to be simulated is the communications activity required for each group\textsubscript{i}, more specifically the number of characters not already present in local memory.

We keep track of the cumulative unused time on the communications channel at each point. If at time group.time\textsubscript{i} there is any cumulative communications time left to prefetch the characters in group\textsubscript{i} that are not already present locally, then the communications time (tcs*\text{nchars}) is subtracted from the cumulative total for as many of those characters as possible. The remaining characters in group\textsubscript{i} must be fetched after group.time\textsubscript{i}, resulting in a delay in processing the user's command.

The algorithm in Figure 3-2 describes the simulation process more precisely. It computes the cumulative delay for a given editing session when ideal prefetching is used. For comparison purposes, it also computes the cumulative delay for transmitting all of the characters in group\textsubscript{i} using conventional editing at the original bandwidth of the session, or tca * nchars(group\textsubscript{i}). The cumulative delay is only one possible way to summarize the delay intervals. Others, such as the interval histogram of the delay intervals, will be considered later.

IdealDelay is the total amount of time the user has to "wait" for characters to be transmitted from the host during editing because they were not prefetched or otherwise present in local memory at the time that they were needed locally\footnote{Note that the user might not actually be waiting during this time, as he might be reading characters that are already displayed.} using the ideal prefetch algorithm at the simulated bandwidth, which is slower than the actual bandwidth. ConventionalDelay represents the total amount of time required...
IdealDelay:=ConventionalDelay:=0;
oldTime:=0;
Memory:=phi;

for each group in groups do begin
  newChars:=length(group)-length(group intersect Memory);
  
  comment "prefetch" as many chars as possible:
  while group.time-oldTime>=tcs and newChars>0 do begin
    oldTime:=oldTime+tcs;
    newChars:=newChars-1;
  end;

  comment Fetch the rest and compute the delay;
  if newChars>0 then begin
    IdealDelay:=IdealDelay+tcs*newChars;
    oldTime:=group.time;
  end;

  comment Compute the delay for conventional editing;
  ConventionalDelay:=ConventionalDelay+tcs*length(group);
  Memory:=Memory union group;
end;

Figure 3-2: Algorithm to Simulate Ideal Prefetching

to transmit every character displayed, including instances when the same group of
caracters are displayed more than once. Thus ConventionalDelay is the delay time
associated with conventional editing at the higher actual bandwidth.

The ratio IdealDelay/ConventionalDelay (the delay ratio) is then one possible
measure of the delay experienced by the user using the ideal prefetch scheme at
the simulated bandwidth, relative to the delay actually experienced by the user during
conventional editing at the higher actual bandwidth. Values greater than 1 would
indicate that the ideal prefetch scheme at the lower, simulated bandwidth performed
worse than conventional editing at the higher bandwidth. Values below 1 would
indicate that the user would experience less total delay at the simulated bandwidth
with prefetching than at the original, higher bandwidth using conventional methods.
3.5. Data Analysis and Reduction

The delay ratio represents one possible measure of utility of a proposed prefetch algorithm. Other measures might be proposed as well. For example, one might measure the proportion of the time that requested characters were already present in local memory, the probability that the next screenful of characters will be prefetched, the maximum amount of delay time that a user experienced at any point during the editing session, or the frequency with which the prefetch delay exceeded the conventional delay. These measures differ primarily in the way that data for a particular simulated editing session are summarized or reduced to a linear measure. Given a particular choice, one must also decide how to summarize that measure across all of the simulated editing sessions. For example, all intervals for all sessions can be combined and treated equally, or the average interval for each session can be computed and these average intervals treated equally. The final results may depend on the chosen method of averaging.

Ideally one would like to find a measure of utility that corresponds closely with the subjective “feel” of the user about the “responsiveness” of the editor. Psychological analysis of various proposed schemes goes beyond the scope of this thesis; nevertheless, one must be careful to choose utility measures that correspond well to phenomena that are perceptible to the user. Since the primary purpose of this aspect of distributed editing is to increase the effective communications rate, these considerations suggest measurement and comparison of the distribution of delay times for the regular sessions and the simulations of various prefetch methods, as well as some summary mechanism such as the mean or median delay. For example, one can compute an interval histogram of the delay times.

Given that the delay is to be averaged in various ways and its distribution computed, there are additional choices for analysis that affect session-to-session variability. For example, it may be the case that some prefetching schemes do well for some types of editing and poorly for others, while other schemes perform
better for the latter than for the former. To discover patterns of this kind in the simulated data one must analyze individual editing sessions to isolate those sessions for which the simulated results were particularly good or particularly bad. Such patterns might suggest "hybrid" prefetch schemes that depend on some dynamic characterization of user editing or upon the nature of the file being edited.

The particular editing command language used introduces a potential source of variation as well. For example, editors that facilitate movement from one part of a file to another may produce different results from those that do not. Furthermore, the command language may provide a way to predict the appropriate part of the file to prefetch next. It may prove useful to add a few primitives to editors that allow the user to declare in some fashion which parts of the file are to be examined, preferably through a command sequence that corresponds naturally with the editing task rather than through an artificial mechanism whose only purpose is efficiency. One example of such a mechanism is a command that declares that the text surrounding every instance of a given string is to be edited in turn.

3.6. A Class of Practical Text Fetch Schemes

In this section I shall describe a class of practical text fetching schemes which includes conventional methods and some simple prefetching schemes. I chose to study these simple methods first on the general principle that when designing software, one should favor the simplest algorithms that perform the desired task well.¹⁰ As it turns out, these simple methods work so well that additional study of more complex general schemes does not seem warranted within the scope of the present work.

The methods are classified according to how much local memory is used, how

¹⁰This principle is the software analog of the principle of Occam's razor, which favors the simplest theory that explains experimental data.
the text to be prefetched is chosen, and the limit on the number of characters prefetched. The notation is as follows:

\[ M_n \]  
The local processor has n characters of memory available for storing text.

\[ N_n \]  
Up to n of the "next" characters (those forward of the present position in the file) will be prefetched whenever the communications channel would otherwise be idle.

\[ P_n \]  
Up to n of the "previous" characters (toward the beginning of the file from the present position) will be prefetched whenever the communications channel would otherwise be idle.

\[ D_n \]  
Up to n characters will be prefetched in a direction determined by the most recent editing motion within the file. For example, if the user just requested that lines previous to the current position in the file be displayed then the editor will use \( P_n \) prefetching. If the user next asks that a window forward of the current position be displayed then the editor will use \( N_n \) prefetching.

If both "N" and "P" prefetching are specified, the order determines which will be done first. Thus "N:500 P:500" specifies that up to 500 of the next characters in the file will be prefetched, and then if there is time, up to 500 of the previous characters will be prefetched. In the following subsections, the most interesting methods in this class are described.

3.6.1. Conventional Demand Method - M:0

Using conventional methods, the local processor has no storage other than the current screenful of text. Whenever the user moves to a different section of the file a new screenful of text must be transmitted, regardless of the previous editing history. The host sends text to the local terminal "on demand" by the user. This method requires a minimum of interaction between the host and local processors but requires high bandwidth communications to be effective.
3.6.2. Unlimited Memory - M:=

When considering the effectiveness of prefetching, one must separate the advantage gained simply from the addition of local memory from the effects of the chosen algorithm for guessing what text will be needed next. The "memory" algorithm simply remembers whatever has been displayed directly by user commands during the editing without attempting to prefetch. For example, if the user were to edit a section of the file initially, there would be a delay while that screenful was transmitted. Later, if he moved to the next screenful below the original one, there would be another delay. If he then moved back to the original screenful, there would be no additional delay because that screenful would have been stored in local memory.

While no actual processor has "infinite memory", the number of unique characters displayed during a typical editing session is, in practice, quite small even when editing large files. This result derives as a consequence of the limited rate at which characters can be transmitted, and the limited rate at which users can read and interpret text. In practice, M:10K performs about as well as M:= (where K is $2^{12}$). Therefore, the chosen schemes all assume that the memory is unbounded. If in fact the local processor exhausts its memory, a reasonable algorithm could be used to throw away information on the least recently referenced and most remote sections of the local copy of the file.

3.6.3. Unlimited Prefetch-Next - M:= N:=

Using the "prefetch-next" scheme, the local processor stores all previously transmitted text as with the "memory" scheme above, but also utilizes any spare time during which the communications channel is free to prefetch any text immediately following the currently displayed portion of the file that is not already in local memory. For example, when the user begins the editing, there is a delay when the first screenful of text is transmitted. While he is editing that text, the local
processor requests that the text following the current screenful be transmitted, as
much as can be in the time that the user spends editing the current screenful. If
the user then moves on to some of the prefetched text, there is no additional
delay. If he moves to a part of the file that has not be prefetched, though, he
experiences a delay. Note that if the user spends enough time on any screenful of
text, there may be enough time to prefetch the entire file, since there is no limit on
the number of characters prefetched.

3.6.4. Limited Prefetch-Next - M:= N:n

The limited "prefetch-next" scheme works like the unlimited version except
that the amount of text that is prefetched is limited. Once that amount of text has
been prefetched, no further prefetching is done until the user moves to a portion
of the file where the chosen number of lines or characters has not yet been stored
in local memory. If a limited prefetch-next scheme performs nearly as well as the
unlimited scheme, we can save on host computer loading and data transmission
costs by lowering the limit, n, as much as possible to reduce "wasted transmission"
of parts of the file. Limiting total output from the host may be critical in some
applications.

3.6.5. Direction-Dependent Prefetch - M:= D:n

The "prefetch-next" scheme depends on the assumption that the user is more
likely to move forward in the file than backward. In some cases it might be
advantageous to prefetch the previous screenful instead of the next one. One
obvious clue to the user's intentions is the direction in which he previously moved.
The "direction-dependent prefetch" scheme performs prefetching in the direction
that the most recent window motion occurred. For example, if the tenth window
of the file is displayed, followed by the ninth, this method will prefetch the eighth
window.
3.6.6. Limited Prefetch-Next-Previous - M: N i P: m

Once the limit has been reached on the "prefetch-next" scheme, it might be useful to start prefetching the previous screenfuls. This scheme could be varied in may different ways, for example by prefetching alternately from before and after the current position in the file. In the simplest case, up to n of the next characters are prefetched, and then up to m of the previous characters are prefetched.

3.7. Specialized Prefetching Schemes

Many specialized modifications to the simple schemes described above can be made. One can take advantage of the command language of a particular editor, or of the properties of a particular kind of data file, to increase the probability of prefetching the correct text. For example, in some situations, the user may search for selected substrings in the file and edit the file at or near those substrings. If a search command has been issued, it may be advantageous to begin prefetching text at the next occurrence of the substring rather than at the screenful following the current location, in case the user chooses to search for the same substring again. It might be useful to install special editor commands that allow the user to optionally declare his intent to search repeatedly in this fashion and thus allow the local editor to optimize prefetching.

The editor might have a variety of prefetching schemes available to it and choose from among them based on dynamic performance or some characteristic of the file, such as its type (e.g. ALGOL source, numeric data file, English document) or the type of editing (normal editing versus read-only examination of the file). Other specialized commands and prefetching schemes to handle them well might also be imagined. It is not possible to analyze within this discourse all such modifications that might be added to a good general prefetching scheme. Nevertheless, modifications could clearly benefit particular implementations and could be applied in addition to the simple but general prefetching schemes described in Section 3.6.
3.8. Summary

This chapter has introduced the notion of prefetching, defined the ideal prefetching algorithm, and has described a class of prefetch schemes. Chapter 5 describes the collection of empirical data from text editing sessions at Rutgers, and the simulation of these methods to determine their performance relative to the ideal prefetch method. Chapter 4 treats communication in the other direction, from the local system to the host.
CHAPTER 4

TRANSMISSION FROM LOCAL PROCESSOR TO HOST

The preceding chapter dealt with the transmission of parts of the file from the host system, which is the permanent repository of the file, to the local processor. Host-to-local transmission has been emphasized because many more characters are displayed on the user’s terminal during typical editing than are typed by the user. The actual changes made to a file generally require a small number of selected keystrokes made at a very low rate (e.g. 5 cps) compared to the rate at which the text is transmitted for viewing (e.g. 960 cps). Nevertheless, the design of a distributed editor must handle transmission of file changes back to the host, including both the method of representation of these changes and the implementation decision of how often to transmit stored up changes back to the host.

This chapter addresses the issues of local to host communication by analyzing methods of describing the difference between two strings. An algorithm that finds the minimal (most concise) difference description between a pair of strings, relative to a set of difference operators, is derived. This algorithm serves both as a potential method for local to host communication and as a benchmark with which simpler heuristic difference description methods may be compared.
4.1. Motivation for Studying File Difference Representations

When changes are made to the local representation of the file, they must eventually be communicated back to the host system for incorporation into the permanent copy. The maximum rate of data transmission required for describing the editing changes to the host is bounded by the rate at which the user types, since at worst the local processor could simply transmit each keystroke. One can in fact do much better than simply transmitting user keystrokes. Even in the case of inserting into an empty file, for which the transmission is entirely from the local processor to the host, the rate at which text must be sent to the host is still bounded by the rate at which the user can type. The 300 baud rate, the slowest commonly used phone line communications speed (see Figure 3-1), is sufficient to keep up with even the fastest typists.

Despite the comparatively smaller communications demand made by local to host transmissions, they cannot be ignored completely. Recall that one reason for studying distributed editor architectures is to reduce host interrupts. If every character, or almost every character, typed by the user is transmitted to the host immediately, the number of interrupts will not be reduced compared to conventional editing. Thus some means of summarizing local changes must be used to reduce the number of "packets" of characters sent to the host.

Some editing systems in use today employ a "smart terminal" that stores one or more screenfuls of text. When the user has finished editing a screenful, he hits a key that transmits the entire screen back to the host. This system avoids the interrupt problem at the expense of making the local-to-host transmissions as demanding of communication as their host-to-local counterparts. Others have tried to optimize this solution slightly by transmitting only the changed lines, along with a necessary line identification. Some implementors have suggested schemes for "saving up" user keystrokes that describe the editing changes, and transmitting them in a batch to the host. But how efficient are the user keystrokes at describing the
changes made to the file? Can they be modified to be more concise (or is it worth the effort)? Is it reasonable to compute the difference directly using the original and changed text? Choosing an appropriate distributed editor architecture depends on the answers to these questions.

As the above questions suggest, local to host transmissions depend on methods for describing changes to the file in a concise manner. This chapter will concentrate on developing the required analytic and algorithmic methods for describing the differences between the original file and the changed version, or more generally, the difference between any two arbitrary strings. Study of this problem will lead to a definition of the minimal difference description between two strings and an algorithm to compute it. Like the ideal prefetch algorithm, the minimal difference algorithm provides a benchmark against which competing methods can be judged.

4.2. A Class of Difference Problems

Given two strings s and t, what is the "best" way to describe the differences? We require that the difference description be generated by someone who knows both s and t. This person then communicates it to someone who knows only s, who applies it to his copy of s to reconstruct t. This question defines a class of difference problems, depending on the definition of the word "best" as well as on the chosen set of difference operators and their representation.

For our purposes, the "best" difference description takes the fewest bits of information to encode. The number of bits required to describe the difference between two strings depends critically on the choice of operators and their representations. To take an extreme case, suppose that s is President Reagan's Inauguration address and t is Lincoln's Gettysburg Address. If the two communicants plan to make this particular improvement to Reagan's speech on a regular basis, they could agree to represent it by a special Reagan operator that requires only one bit
to describe. On the other hand, if the only available operator is "replace s with the following" then it will be necessary to transmit the entire Gettysburg Address to describe the difference. Thus, depending on which operators are chosen, any particular difference may be represented using anywhere from one bit of information to the number of bits required to represent the second string in its entirety.

For the sake of generality, we will avoid specialized one or two bit operators such as Reagan above. Rather, we will choose a small set of general operators that are sufficient to describe string differences, usually requiring fewer bits than the transmission of all of t. In particular, the operations of skipping to a given location within the string and inserting, replacing, and deleting characters form such a set.

4.3. Previously Solved Related Problems

The problem of determining the difference between two strings has received much attention in the literature [Heckel 78, Hirschberg 75, Hirschberg 77, Hunt 77, Lowrance 75, Wagner 74]. Several algorithms for computing the maximal common subsequence of two strings have been presented recently in [Hirschberg 75, Hirschberg 77, Hunt 77, Lowrance 75, Wagner 74]. These papers have dealt with the edit distance between two strings, or the smallest number of insertions, deletions, and replacements required to change one string into the other. The best of these algorithms run in time proportional to $|s| \cdot |t|$ (worst case) and space proportional to $|s| + |t|$, where $|x|$ is the length of x. They provide single character insert, delete, and replace operators, each with a fixed cost, and find the sequence of operators to apply with the least total cost. They do not associate a cost with describing where an operator is to be applied, e.g. which character is to be deleted. Furthermore, they consider the cost of applying an operator n times in succession to be n times the cost of applying it once. These algorithms have been
applied to problems where the cost they describe is related to the probability that one string will be converted into another, for example due to typographical or spelling errors.

The insert, delete, and replace operations, supplemented by information about the characters to which they apply, may be thought of as an encoding of the difference between the two strings. Published algorithms have minimized the cost of performing the operations required to change one string into another, but have not minimized the cost of representing the difference.

To see how the two minimization problems differ, it is helpful to consider an example of two strings and the difference derived by Wagner's algorithm ([Wagner 74]) versus the most concise difference. Consider the strings

\[ s = "the history of this project" \]
\[ t = "this project" \]

The Wagner algorithm produces a difference encoding that deletes the characters with arrows under them below:

```
the history of this project

++++ +++++ +++++ +++++
```

Using any reasonable representation, a more optimal difference encoding would delete the first 15 characters of \( s \):

```
the history of this project

+++++++++++++
```

We see that the Wagner algorithm favors the shortest possible deletion because it stops deleting from \( s \) as soon as a character in \( s \) that matches the next character in \( t \) is found. On the other hand, in the following example, finding the shortest possible deletion produces a more optimal difference encoding than finding the longest.
s = "the second reading had a rather bad ending"
t = "the reading"

shortest: the second reading had a rather bad ending

longest: the second reading had a rather bad ending

In this case, the Wagner algorithm finds the optimal difference encoding. The fact that Wagner's algorithm does not necessarily produce the most concise difference description is no surprise because the algorithm considers every deletion to be of equal cost, regardless of where in the string it occurs.

4.4. Minimal String Difference Encodings

The existing algorithms for describing string differences do not produce the minimum cost difference description according to our definition of cost, that must account for the number of bits that are required to describe the difference. In this section, we will derive an algorithm for finding the minimal string difference encoding of two strings, or the difference description that can be represented using the fewest bits. Given one of the strings and the difference encoding, it must be possible to reconstruct the second string uniquely. We present an algorithm to compute the most efficient difference encoding given the operators insert, delete, replace, and skip. The approach differs from the usual file comparison schemes such as those described in [Heckel 78] that are not based on algorithms that produce minimal difference descriptions.

---

\[11\] This section is based on [Goldberg 82]
4.4.1. Problem Definition

Given two strings $s$ and $t$, and the notion of the current position within a string, a difference encoding is defined as an ordered sequence of the following operations that, when applied to $s$, yields $t$. The ordered sequence will be referred to as an OpSeq (pronounced op' seek).

**Delete** Deletes the current character in $s$, leaving the current position at the following character.

**Insert(c)** Inserts the character $c$ before the current character in $s$, leaving the current character unchanged.

**Replace(c)** Replaces the current character in $s$ with $c$, advancing the current character to the next one in $s$.

**Skip** Advances the current character to the next one in $s$.

The notion of the current position, coupled with the Skip operator, specifies the location at which each operator in OpSeq is to be applied within the modified version of $s$ being constructed. Some of the other operators change the current position as a side effect in addition to their primary purpose. At the beginning of the operator sequence the current position is at the first character of $s$.

The cost of an operator reflects the number of bits used to represent it. The cost of the entire sequence is the sum of the individual costs of the operators it contains, possibly with a "discount" for using the same operator more than once in succession. The operators Insert and Replace specify new characters so their representation includes enough bits to describe that character in addition to the bits required to select the operation. Since it is possible to save some bits by representing multiple occurrences of the same operator using a repeat count, we allow the cost of an operator to be a tuple, $\text{Cost}(op) = (\text{Cost1}(op), \text{Cost2}(op))$, where $\text{Cost1}(op)$ is the cost of applying the operator the first time and $\text{Cost2}(op)$ is the incremental cost of applying it successive times. The first cost must be positive and the second non-negative. For reasonability we assume that the second
cost is less than or equal to the first, and in the case of Insert and Replace that it is at least the cost of representing a character. The problem is to find an OpSeq such that Cost(OpSeq) is minimized.

As a concrete example, let OpSeq be represented by a string of 7-bit bytes (e.g. ASCII characters). A sequence of operations is represented by a byte to select the operator, followed by a second byte that is a repeat count n telling how many successive times the operator is to be applied. For Insert and Replace, the repeat count is followed by n bytes containing the new characters\(^{12}\). For this example, the cost of each operator (in bytes) is

\[
\begin{align*}
\text{Cost (Delete)} &= (2,0) \\
\text{Cost (Insert)} &= (3,1) \\
\text{Cost (Replace)} &= (3,1) \\
\text{Cost (Skip)} &= (2,0)
\end{align*}
\]

The cost of performing n inserts, for example, is \(3 + (n-1) \times 1\).

**4.4.2. Definitions and Notation**

An OpSeq is an ordered sequence of operators that changes string \(a\) to string \(t\). At the beginning of the sequence, the current position is at the first character of \(a\). Last(OpSeq) is the last operator in OpSeq. SeqSet[\(i,j,op\)] is the set of all OpSeq's that end with the operator \(op\) and that change \(a_{1:i} \) to \(t_{1:j}\), where \(x_{1:i}\) denotes the first \(i\) characters of the string \(x\). SeqSet[\(i,j,^\#\)] is the union of SeqSet[\(i,j,op\)] for all operators \(op\). In general, "^\#" as a subscript will denote all possible values of that subscript. Cost1(op) is the cost of representing the operator \(op\) once (in bits or bytes), and Cost2(op) is the incremental cost of representing a second or successive op. Cost(OpSeq) is the total cost of representing OpSeq, taking into account successive occurrences of the same

\(^{12}\)The optimal OpSeq is relative to a particular choice of operators and their representation. This particular choice of representation is convenient but is by no means the most economical possible. For example, the selection of the operator could be done using only two bits.
operator. \( \text{MinCost(SeqSet}[i,j,\text{op}]) \) is the cost of the minimum cost \( \text{OpSeq} \) in \( \text{SeqSet}[i,j,\text{op}] \). The operator "\(|\)" denotes concatenation. \( \text{OpSeq} \ || \ \text{op} \) represents the sequence formed by concatenating \( \text{op} \) to the end of \( \text{OpSeq} \). \( \text{SeqSet}[i,j,*] \ || \ \text{op} \) represents the set of sequences formed by concatenating \( \text{op} \) to the end of each \( \text{OpSeq} \) in \( \text{SeqSet}[i,j,*] \).

4.4.3. Algorithm

Given two strings \( s \) and \( t \) we will first find the minimal cost \( \text{OpSeq} \) using dynamic programming to consider the cost of every interesting sequence of operators that changes the substring \( s_{1;i} \) to \( t_{1;j} \), for \( i=0 \) to \( |s| \) and \( j=0 \) to \( |t| \). Next, a second algorithm will be employed to "walk backwards" through the array of interesting solution costs produced by the first algorithm to construct a minimal cost sequence that changes \( s \) to \( t \).

4.4.3.1. Computing the Cost Array, \( L \)

In the first step of the algorithm, we shall compute the entries of the array \( L \), where \( L[i,j,\text{op}] \) is defined as the cost of the minimum cost sequence that ends with the operator \( \text{op} \) and changes \( s_{1;i} \) to \( t_{1;j} \). Thus, by definition, \( L[i,j,\text{op}] = \text{MinCost(SeqSet}[i,j,\text{op}]) \). \( L \) may be thought of as a matrix in which the \( i,j \)th square contains four values, corresponding to the cost of the most efficient \( \text{OpSeq} \)s that end with each of the four possible operators, and that change \( s_{1;i} \) to \( t_{1;j} \). The minimum cost of an \( \text{OpSeq} \) that changes \( s \) to \( t \) is the smallest of the four values in \( L[|s|,|t|] \):

\[
\text{min}(L[|s|,|t|,\text{Delete}], L[|s|,|t|,\text{Insert}], L[|s|,|t|,\text{Replace}], L[|s|,|t|,\text{Skip}]).
\]

The entries \( L[*,0,*] \), \( L[0,*,*] \), and \( L[1,1,*] \) can be computed trivially. \( L[0,0,*] \) is the minimum cost of changing the null string to itself, which is zero. \( L[0,0,\text{Delete}] \) is the minimum cost of changing \( s_{1;i} \) to the null string with the last operator a Delete, which can be done only by 1 deletions in succession.
L[0,j, Insert] is the minimum cost of changing the null string to t_{1:j} with the last operator an Insert, which can be done only by performing j insertions in succession. All other entries L[0,*,*] and L[*,0,*] correspond to impossible OpSeq's so their minimum cost is undefined. For bookkeeping purposes, we can set these entries to infinity, which effectively eliminates them from consideration.

For each of the entries L[1,1,*] there is a single sequence that performs the required operations so its cost is easy to compute. L[1,1, Delete] and L[1,1, Insert] correspond to a single deletion and a single insertion, so they both have the same cost. Cost1(Delete) + Cost1(Insert). The only sequence associated with L[1,1,Replace] is a single Replace operator, whose cost is Cost1(Replace). Similarly, L[1,1, Skip] is Cost1(Skip) if s_j = t_j; otherwise it is infinite.

Once the initial values of L have been entered, we can iteratively determine the values of L[i,j, op], for i=1,2,...|s|, j=1,2,...|t|, i+j>2, by examining the previously computed values L[i,j-1,*], L[i-1,j,*], and L[i-1,j-1,*]. To see that L[i,j, op] depends only on these three predecessors, first consider computing Cost(OpSeq || op) for any non-empty OpSeq. The incremental cost of adding op depends only on whether last(OpSeq) is the same as op:

Cost(OpSeq || op) := if last(OpSeq) = op 
them Cost(OpSeq) + Cost2(op)
else Cost(OpSeq) + Cost1(op)

Next consider constructing the set of all possible OpSeq's, SeqSet[i,j, op]. From the definitions of the four operators and the fact that SeqSet[i,j, op] must end with operator op, we obtain the following set constructions:

SeqSet[i,j,Delete] = SeqSet[i-1,j, *] || Delete
SeqSet[i,j,Insert] = SeqSet[i,j-1,*] || Insert(t_j)
SeqSet[i,j,Replace] = SeqSet[i-1,j-1,*] || Replace(t_j)
SeqSet[i,j, Skip] = if s_j = t_j then SeqSet[i-1,j-1,*] || Skip
else EmptySet

Using the definition of L[i,j, op] and the formula for the incremental cost of
concatenating an operation to the end of a sequence, these constructions lead to
the following relationships in the array \( L \):

\[
L[i, j, \text{Delete}] = \min(L[i-1, j, \text{Delete}] + \text{Cost1(Delete)},
L[i-1, j, \text{Delete}] + \text{Cost2(Delete)})
\]

\[
L[i, j, \text{Insert}] = \min(L[i-1, j, \text{Insert}] + \text{Cost1(Insert)},
L[i, j-1, \text{Insert}] + \text{Cost2(Insert)})
\]

\[
L[i, j, \text{Replace}] = \min(L[i-1, j-1, \text{Replace}] + \text{Cost1(Replace)},
L[i-1, j-1, \text{Replace}] + \text{Cost2(Replace)})
\]

\[
L[i, j, \text{Skip}] = \min(L[i-1, j-1, \text{Skip}] + \text{Cost(Skip)},
L[i-1, j-1, \text{Skip}] + \text{Cost2(Skip)})
\]

These relationships are used by Procedures BuildL and Smallest, shown in Figure
4-1, to compute \( L[i, j, \ast] \) given its predecessors.

4.4.3.2. Procedure DeriveOpSeq

Procedure DeriveOpSeq in Figure 4-2 returns a minimal cost OpSeq given the
array \( L \) produced by procedure BuildL. It first determines the last operator by
examining \( L[s], [t], \ast \) to find the op for which \( L[s], [t], \text{op} \) is smallest. If there
is more than one such op, it chooses one of the possibilities arbitrarily. The choice
of \( \text{op} \) uniquely determines the predecessor entry \( L[m, n, \ast] \) to examine. The next-
to-last operator in the sequence is then the value of \( k \) for which \( L[s], [t], \text{op} = L[m, n, k] \) plus the incremental cost of performing \( \text{op} \) after \( k \). Again, if there is
more than one such \( k \), one is chosen arbitrarily. The choice of \( k \) uniquely
determines the predecessor entry of \( L \) to examine next. This process continues
until \( L[0, 0, \ast] \) has been reached, at which point the algorithm has produced an
OpSeq that changes \( s \) to \( t \) and whose cost is Smallest\( (s), (t) \).
procedure BuildL(s,t);
begin
  comment Step 1: Initialize known values of L;
  for op := Delete, Insert, Replace, Skip do L[0,0,op] := 0;
  for i := 1 upto length(s) do begin
    L[i,0,Delete] := Cost1(Delete) + (i-1)*Cost2(Delete);
    L[i,0,Insert] := L[i,0,Replace] := L[i,0,Skip] := @;
  end;
  for j := 1 upto length(t) do begin
    L[0,j,Insert] := Cost1(Insert) + (j-1)*Cost2(Insert);
    L[0,j,Delete] := L[0,j,Replace] := L[0,j,Skip] := @;
    L[1,1,Delete] := L[0,1,Insert] + Cost1(Delete);
    L[1,1,Insert] := L[1,0,Delete] + Cost1(Insert);
    L[1,1,Replace] := Cost1(Replace);
    L[1,1,Skip] := if s[1]=t[1] then Cost1(Skip) else @;
  end;
  comment Step 2: Fill in L[i,j] based on predecessors;
  for i := 1 upto length(s) do
    for j := 1 upto length(t) do
      if i+j > 2 then begin
        L[i,j,Delete] := min( Smallest(i-1,j) + Cost1(Delete),
                              L[i-1,j,Delete] + Cost2(Delete));
        L[i,j,Insert] := min( Smallest(i,j-1) + Cost1(Insert),
                              L[i,j-1,Insert] + Cost2(Insert));
        L[i,j,Replace] := min( Smallest(i-1,j-1) + Cost1(Replace),
                              L[i-1,j-1,Replace] + Cost2(Replace));
        L[i,j,Skip] := if s[i]=t[j]
                       then min( Smallest(i-1,j-1) + Cost1(Skip),
                              L[i-1,j-1,Skip] + Cost2(Skip))
                       else @;
      end;
  end;

integer procedure Smallest(i,j);
begin
  comment Return the minimum of L[i,j,\*];
  Smallest := min( L[i,j,Delete],
                   L[i,j,Insert],
                   L[i,j,Replace],
                   L[i,j,Skip] );
end;

Figure 4-1: Algorithm for Computing the Entries of L[i,j,op]
procedure DeriveOpSeq:
begin
  OpSeq := NIL;
i := length(s);
j := length(t);

  comment Find the least cost final operator;
opcost := -1;
  for k := Delete, Insert, Replace, Skip do
    if L[i,j,k] < opcost then begin
      opcost := L[i,j,k];
op := k;
    end;

  comment Trace back the sequence ending in it;
  while i>0 or j>0 do begin
    OpSeq := op || OpSeq;
    case op of begin
      [Delete] i:=i-1;
      [Insert] j:=j-1;
      [Replace] begin i:=i-1; j:=j-1; end;
      [Skip] begin i:=i-1; j:=j-1; end
    end;
    comment Find its predecessor;
    for k := Delete, Insert, Replace, Skip do
      if CumCost(i,j,k,op) = opcost then begin
        op := k;
opcost := L[i,j,op];
exti:op;
      end;
  end;
  return(OpSeq);
end;

Integer procedure CumCost(n,m,prevOp,op):
begin
  comment Return cumulative cost for applying op after L[n,m,prevOp];
  CumCost := if prevOp ≠ op or n+m = 0
             then L[n,m,prevOp] + Cost1(op)
             else L[n,m,prevOp] + Cost2(op);
end;

Figure 4-2: Procedure to Derive a Minimal Cost OpSeq Given Array L
4.4.3. Example

Figure 4-3 shows an example of the array L for the strings s = "ice cream" and t = "i scream". The four values in L[1,1] indicate the cost of changing s1 to t1, or the string "i" to itself, by various methods. The cheapest method is to use the skip operator, at a cost of 2 bytes. One could also delete the "i" in s and then insert the one in t (or vice versa) at a cost of 5, or replace the "i" in s with the one in t at a cost of 3. As an example of the need to store all four values in each cell of the matrix, note that the cost of getting to L[4,2,Insert] from L[4,1] is 6 as a result of L[4,1,Insert], even though L[4,1,Delete] is smaller.

The bottom right entry, L[9,8], tells us that the least cost OpSeq ends with a skip operator and has a cost of 9. The skip moved from L[8,7] to L[9,8]. Examining L[8,7], it is clear that the next-to-last operator must also have been a skip. In fact, the least cost OpSeq must have ended with a series of five skip operators moving from L[4,3] to L[9,8]. Examining the costs in L[4,3] we observe an ambiguity: the operator before the five skips could have been Insert, delete, or replace. Thus there is more than one least cost sequence. Arbitrarily, let us choose Insert. We now examine L[4,2] and determine that the previous two operators must also have been Insert, which brings us to L[4,0]. The only way to get from L[4,0] back to L[0,0] is by four delete operators. Thus the minimum cost OpSeq we have traced deletes the "ice " in "ice cream" and inserts "i s" in its place.

4.4.4. Complexity Analysis

The algorithm requires |s| · |t| · nops words of storage for the array L, where nops is the number of unique operators considered by the algorithm (nops = 4 in the algorithm as presented). Note that nops is actually a constant with respect to the problem the algorithm solves. The memory used for storing the derived OpSeq is O(|s| + |t|) in the worst case.
<table>
<thead>
<tr>
<th>i</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>j=0</td>
<td>S0</td>
<td>S1</td>
<td>S2</td>
<td>S3</td>
<td>S4</td>
<td>S5</td>
<td>S6</td>
<td>S7</td>
<td>S8</td>
</tr>
<tr>
<td>i=1</td>
<td>l0</td>
<td>l1</td>
<td>l2</td>
<td>l3</td>
<td>l4</td>
<td>l5</td>
<td>l6</td>
<td>l7</td>
<td>l8</td>
</tr>
<tr>
<td>j=2</td>
<td>d0</td>
<td>d1</td>
<td>d2</td>
<td>d3</td>
<td>d4</td>
<td>d5</td>
<td>d6</td>
<td>d7</td>
<td>d8</td>
</tr>
<tr>
<td>k=3</td>
<td>r0</td>
<td>r1</td>
<td>r2</td>
<td>r3</td>
<td>r4</td>
<td>r5</td>
<td>r6</td>
<td>r7</td>
<td>r8</td>
</tr>
<tr>
<td>c 2</td>
<td>s0</td>
<td>s1</td>
<td>s2</td>
<td>s3</td>
<td>s4</td>
<td>s5</td>
<td>s6</td>
<td>s7</td>
<td>s8</td>
</tr>
</tbody>
</table>

Figure 4-3: Example of the Minimum Cost Array \( L[i,j] \) for the strings \( s = "ice cream" \) and \( t = "i scream". One of the derived minimal difference encodings is delete(4), insert("i s").
Step 1 of Procedure BuildL has linear time complexity $O(|s|+|t|)$. Each iteration of step 2 examines $O(nops^2)$ locations in the array and the inner loop is executed $|s| \cdot |t|$ times, so the time complexity of step 2 is $O(|s| \cdot |t| \cdot nops^2)$. Procedure DeriveOpSeq executes its while loop $O(|s|+|t|)$ times and examines $nops$ locations in the array so its time complexity is $O((|s|+|t|) \cdot nops)$.

The overall complexity of procedures BuildL and DeriveOpSeq is $O(|s| \cdot |t| \cdot nops)$ in space and $O(|s| \cdot |t| \cdot nops^2)$ in time.

4.4.5. Comments on the Algorithm

Trailing Skip operators need not be included in the cost of the minimal difference description since they do not affect the resultant string. To take advantage of this optimization, procedure DeriveOpSeq can be modified to decrease $L[|s|, |t|, \text{Skip}]$ by the cost of the final series of $n$ Skip operators, $\text{Cost1(Skip)} + n \cdot \text{Cost2(Skip)}$, before selecting the lowest cost final operator. If skip is in fact selected as the final operator, it should begin producing OpSeq at the appropriate operator in $L[|s|-n, |t|, n, \ast]$, rather than at $L[|s|, |t|, \text{Skip}]$.

The same basic algorithm can be adapted to handle additional operators with related characteristics. For example, one can add operators analogous to the four that have been introduced above but that operate on a single character at a time, to save the cost of encoding the repeat count for single character operations. This addition will allow the algorithm to derive lower cost difference descriptions for strings that differ by single characters. Higher level operators, similar to those found in text editors, can also be handled by storing sufficient information in $L$ to allow the algorithm to determine the cost of applying each operator by examining a few predecessor entries. In this fashion the same basic algorithm may be adapted to handle many simple context independent operators, at some cost in time and space as the number of operators $nops$ grows.
The algorithm presented cannot reasonably be used to compute differences between entire files directly, due to its quadratic time and space requirements. For example, assuming that the two files were of length 100,000 characters, and that each step of the algorithm takes $10^{-6}$ second, the computation would require on the order of 10,000 seconds and $10^{11}$ bytes of memory for storage of the array $L$. The average time and space complexity of the maximal common subsequence problem has been reduced from the $|s| \cdot |t|$ of the original algorithm in [Wagner 74] to more linear behavior in [Hirschberg 75, Hirschberg 77, Hunt 77]. Similar techniques may yield average case improvements to the current algorithm as well. Despite its quadratic time and space dependence, however, the algorithm can be applied on a line by line basis as a user edits a file, encoding the changes "on the fly". Alternatively, it can be used to process the output of a line-by-line file comparison utility into a more concise form.

The operators considered in this derivation move forward through string $s$ to produce string $t$. While one could define more general operators that move backwards as well as forwards, these operators would not reduce the cost of the minimum cost difference encoding. The proof of this assertion is tedious and has been omitted. Basically, any operation applied through a sequence of operators in which the direction of motion changes can be applied just as cheaply in a sequence that moves in one direction only.

4.5. Heuristic Approximations to Minimal Difference Encodings

Is it in fact worth the time of the editor implementor to use the algorithm presented in the previous section? It is quite feasible for use in a distributed editor so long as it is used on a line by line basis as the user edits the file, but even so it uses a considerable amount of CPU time and is more complex to implement than other possible schemes. In this section, we will consider two simple heuristics that
can be used in place of the exact algorithm. In the following section, we will
describe ways that the minimal difference encoding algorithm itself can be used to
measure how closely each of these methods approximates the optimal difference
description.

4.5.1. Common Head and Tail Method

The editing changes that a user makes to a line of the file often fall into the
following categories:

- inserting or deleting a line of text
- Making a localized change within a given line, e.g. correcting a
  misspelling, adding or deleting a word or phrase

A difference encoding method that worked close to optimally in these two cases
might reasonably be expected to work well on the average, provided that these
types of changes predominate during typical editing sessions.

The common head and tail method determines the difference between two
strings $a$ and $t$ by determining the greatest common initial substring (the head) and
the greatest common final substring (the tail). It then considers what remains of
the two strings when the head and tail are removed. If there is nothing left in
either string, the two strings are the same. Otherwise, it generates a difference
encoding that replaces what is left of $a$ with what is left of $t$. An algorithm similar
to this one is used by ITS TECO to update the display screen whenever a line of
the edit buffer has been changed.

If a new line is inserted by the user, there is no common head or tail so the
algorithm generates an insert of the entire new line, that is in fact the minimal
difference encoding. If a single group of contiguous characters is inserted, deleted,
or replaced in the line, the algorithm will generate the same insertion, deletion, or
replacement. While not necessarily optimal, this encoding will often be close to
optimal. However, in the case that a small change is made to both the beginning
and end of a line, this algorithm will produce a very suboptimal encoding. For example, consider:

\[ s = "this string will not be changed much" \]
\[ t = "This string will not be changed much." \]

The common head and tail algorithm will not find anything in common, so it will delete all of \( s \) and insert all of \( t \), whereas the minimal difference algorithm will simply capitalize the initial letter of \( s \) and insert a "." at the end.

4.5.2. Optimization of User Commands

The commands typed by the user provide a good clue to the minimal difference description. While the user may not always choose the optimal set of commands to perform a given edit on the file, she will probably make a fairly concise choice to avoid unnecessary typing.

Using the optimization of user commands method, the editor notes the commands used to edit the file and optimizes them into a concise difference encoding. Most editors contain internal primitives for performing insertions and deletions in the file. These primitives are invoked by the procedures that process actual user commands, and can easily generate an OpSeq "on the fly." When the editor is ready to transmit the file changes to the host, it can use various heuristics to compress and optimize the OpSeq that has been generated. For example, successive deletions can be combined into a single delete if they act on adjacent characters of the file. Note that cursor and screen motion commands, which can comprise up to 50% of the commands issued during an editing session, do not cause the OpSeq to grow since they do not change the file.

As with the common head and tail method, this method may in some cases produce very poor results, particularly if the user makes changes to a line and then changes the changed characters. With additional effort, such redundant edits can be reduced by converting the OpSeq into a single, forward pass through the file.
Using additional heuristics, operators whose effect is cancelled later in the sequence can be detected and removed. Performing the linearization and removal of redundant operations perfectly may be as difficult as determining the most concise difference directly. Heuristics for this process may prove sufficient in most cases.

4.6. Need for Empirical Experiments

This chapter has defined the minimal difference encoding between two strings, which is the most concise way to describe the difference between them. An algorithm that computes the minimal difference encoding given two strings and that runs in time proportional to the product of the lengths of the two strings has been derived and presented. If tightly coded, this algorithm can be used in a distributed editor to transmit changes on a line-by-line basis, but would take too long to describe changes on a screen-by-screen basis. Several possible heuristics to produce close-to-minimal difference encodings have also been described. The degree to which these heuristics work depends on typical edits that users make while editing and thus cannot be predicted theoretically.

The minimal string difference algorithm has two significant impacts on this thesis. First of all, it provides a practical way to describe file changes to the host on a line-by-line basis. Secondly, it provides the theoretical lower bound against which more easily implemented approximate methods may be judged empirically. The next chapter provides empirical results that show that these heuristic methods provide difference encodings reasonably close to the theoretical ideal.
CHAPTER 5
EMPIRICAL STUDY

This chapter analyzes empirical data collected during in vivo conventional editing sessions for the purpose of evaluating the effectiveness of potential distributed editing architectures. The instrumentation of text editors used on the LCSR DECsystem-20 at Rutgers University is described and the methods of analysis and simulation of the collected data are presented. The simulations indicate that some simple prefetching schemes would significantly increase the effective communications bandwidth during typical editing sessions and thus reduce the delay time for transmitting parts of the file to the user's terminal. Some simple schemes for sending editing changes back to the host system work much better than retransmitting entire screens or lines of the file. Furthermore, the number of host interrupts can be reduced by an order of magnitude by using these schemes rather than conventional text editing methods. The results of this chapter are used to substantiate claims about the importance of various possible distributed editing architectures.

5.1. Instrumentation of the Editors

To evaluate the effectiveness of prefetching, the major text editors used on the LCSR DECsystem-20 at Rutgers University were modified to produce a data file containing information about the current editing session.¹³ These editors all use

¹³Charles Hedrick implemented the instrumentation code for TVEDIT and much of the EMACS code. Roy Marantz implemented some of the EMACS code. The author instrumented MEDIT and implemented all of the data reduction and simulation programs used to analyze the data collected from the editors.
conventional techniques for communication with the terminal, i.e. they send characters of the file to the terminal on demand whenever the user issues a command that moves the current editing window. The editors make an entry for each group of characters sent to the terminal as a result of user commands that move the current edit window.

Sessions using two conventional video text editors, TVEDIT and EMACS, were studied in detail. A third editor, MEDIT, that incorporates both a line editor (SOS) and a video editor, was also studied to see if there are major differences in performance of prefetching schemes between line editors and video editors.

5.1.1. Statistics File Entries

The file produced by each editor contains sufficient information to determine the sequential character numbers within the file that were displayed on the terminal and the elapsed time interval (in milliseconds) since the previous group of characters was displayed. This information suffices to simulate various prefetching schemes.

Although each editor uses a slightly different notation for the parts of the file that have been displayed, and each outputs somewhat different auxiliary information about the editing session, the statistics files produced by each have a similar format, consisting of single line entries. Each line begins with an alphabetic character describing the type of entry, followed by a time stamp giving the relative time of day in milliseconds since the editing session began, followed by entry-specific data. The entries have the form:

---

14 They also send characters to the terminal to update the current screenful as characters are inserted or deleted within the current window, but this communication was not recorded since a distributed editor would handle local screen updating without communication with the host system.
U stamp user-name editor-name term-number baud-rate date
R stamp input-file nbytes R|C|E
L stamp IDCcount chrpos1 chrpos2
S stamp xxx
W stamp output-file nbytes
E stamp tty-chars-input, tty-chars-output, runtm

and have the following meanings:

U  This entry indicates a user session with an editor that begins at the specified date and time. The editor makes this entry during its initialization at the time the statistics file is opened.

stamp  time in milliseconds relative to the time of day when the editor was first started. This is used to determine the elapsed time between entries. It is monotone increasing throughout any editing session.

user-name  name of the user, e.g. GOLDBERG
editor-name  MEDIT, TVEDIT, or EMACS
term-number  system code for the type of terminal that the user has
baud-rate  baud rate of the user’s terminal
date-time  date and time the session started

R  This entry is made whenever a file is selected for input by the editor. Nbytes is the length of the file in bytes. R|C|E is the type of editing session: R=ReadOnly, C=Creating file, E=normal Edit.

L  This entry is made whenever a line or group of lines is output to the terminal as the result of a window shift. The character positions include end of line characters. IDCcount is the number of characters inserted minus the number deleted since the previous IDCcount was output.

S  This entry is made whenever the user issues a search command of some kind. XXX is editor-dependent but should indicate if the user is repeating a previous search or specifying a new search pattern or patterns.

E  This entry is made at the end of a session

tty-Chars-input  the total number of characters read from the terminal (i.e. typed by the user) during the session.
tty-Chars-output  the total number of characters output to the terminal (including non-printing characters).
Runtm  the runtime of the editing session in milliseconds (to be used for system load determination)
5.1.2. Instrumentation of MEDIT

MEDIT is a hybrid of SOS (a line oriented hard copy editor) and TVEDIT (a video editor) written by the author. Some statistics collected during MEDIT sessions were analyzed using the ideal prefetch simulation to present a rough idea of how prefetching might apply to hard copy editors. While the editing sessions consisted of about 50% line-oriented command and 50% screen-oriented commands and thus do not reflect true hard copy editing, the results indicate that prefetching would be of value to hard copy editors.

For the purpose of collecting statistics, MEDIT was modified to output two data files for each user editing session. The first file contains entries describing the type of editing done, a description of the editing commands used, and entries to indicate precisely which lines of the file were displayed (or typed) on the user's terminal during the editing session. In addition to the standard entries, MEDIT produced an entry for each command typed in its display mode.

The second data file produced by MEDIT contains an entry for each line changed during the session. The entries consist of the original line, the changed line, and the editing primitives (insert, delete, replace) that the user invoked to change the line. Entries are made to this file whenever the user moves the cursor off a line that has been changed.

5.1.3. Instrumentation of TVEDIT

TVEDIT is Stanford's video editor. The version used at Rutgers was written by Pentti Kanerva for the TENEX/TOPS20 operating system. Due to the modularity of TVEDIT, and the fact that it is implemented in a high level language, SAIL, the instrumentation was carried out relatively easily. The only compromise made was that the L entries reflect the number of characters in a line, rather than the number
of characters that fit on a line of the terminal's screen. TVEDIT handles long lines by displaying only the initial part that fits on the screen, and by shifting the line if the user moves the cursor to a portion of the "virtual line" that is off the screen. For most files edited, lines fit on the terminal screen, but for those that have a significant number of long lines, the L entries will indicate that more characters have to be displayed than is, in fact, the case, causing the simulated prefetch schemes to perform poorly.

5.1.4. Instrumentation of EMACS

EMACS is MIT's video editor. It is actually an open-ended collection of macros used with MIT's video TECO editing language. The nature of the underlying TECO implementation (it is written in assembly language), the division of labor between hard-coded TECO primitives and TECO macros, and the open-endedness of the package, made EMACS the hardest editor to instrument. Some of the statistics collected for the other editors, such as the total number of characters output to the terminal, proved virtually impossible to collect in EMACS because so many separate routines were involved. In addition, the editor is so general that it is often impossible to tell at the level of the underlying TECO primitives just what is being done at the level of user editing.

Many compromises were made in the interest of getting useful data with reasonable effort. Heuristics were used to guess which editing buffers were actual files that the user was editing, rather than initialization files or temporary workspace buffers created by macros. The L entries gave the current screen bounds each time they were changed, rather than the actual characters that the editor sent to the terminal, because the screen update routines are so complex that more specific information could not be obtained easily. When the file size was changed due to offscreen modifications, a special C entry had to be made so that the simulation program could adjust its model of the local memory. However, so many small C entries were generated that the smaller ones had to be ignored to prevent the data
file from growing too large during typical editing. Offscreen changes are not possible in TVEDIT, and lines are identified by a unique index in MEDIT, so this problem did not arise for these editors. These compromises yielded enough useful statistics to verify that the performance of prefetching in typical EMACS sessions would be similar to the performance in TVEDIT sessions.

5.2. General Editing Statistics

The user community of the Rutgers LCSR DECsystem-20 consists primarily of research faculty and graduate students. About 50% of the editing sessions involved English text, most of which was input for various document compilers. 40% of the sessions involved editing computer programs in assorted languages, and the remaining 10% of the sessions were for miscellaneous or undetermined purposes.

Figures 5-1, 5-2, and 5-3 provide general statistics for 14,437 TVEDIT user editing sessions, 1,100 EMACS sessions, and 179 MEDIT sessions. The average TVEDIT and EMACS sessions have similar characteristics except that typical EMACS sessions use more CPU time. The distribution of most variables shown is closer to Poisson than Gaussian, since the session lengths have a Poisson-like distribution and the other variables are strongly dependent on session length. Figure 5-4 shows the distribution of session lengths from which the general statistics were compiled. Because the mean is not very satisfactory when describing a Poisson distribution, the median was also calculated. The median gives a better estimate of performance for the majority of sessions.

Examining Figure 5-1, we see that most of the editing sessions last only a few minutes and use a few CPU seconds of host processor time. Typically, the user types a few hundred characters during one of these sessions, and the host sends some thousands of characters in reply. Of these, about 50% represent local
<table>
<thead>
<tr>
<th>Statistic</th>
<th>Mean</th>
<th>Median</th>
</tr>
</thead>
<tbody>
<tr>
<td>Length of Session (minutes)</td>
<td>11.00</td>
<td>2.05</td>
</tr>
<tr>
<td>Host CPU seconds used</td>
<td>4.9</td>
<td>1.9</td>
</tr>
<tr>
<td>Disjoint regions edited</td>
<td>2.8</td>
<td>2.0</td>
</tr>
<tr>
<td>Characters typed by user</td>
<td>609</td>
<td>100</td>
</tr>
<tr>
<td>Characters output by host</td>
<td>13,166</td>
<td>3,887</td>
</tr>
<tr>
<td>Data Characters Output by Host</td>
<td>6,779</td>
<td>1,953</td>
</tr>
<tr>
<td>Unique Data Characters Output</td>
<td>4,302</td>
<td>1,518</td>
</tr>
<tr>
<td>Initial File Size (characters)</td>
<td>28,672</td>
<td>5,748</td>
</tr>
<tr>
<td>Change in File Size (characters)</td>
<td>-257</td>
<td>0</td>
</tr>
</tbody>
</table>

*Figure 5-1: General Editing Statistics for 14,437 TVEDIT Sessions*

<table>
<thead>
<tr>
<th>Statistic</th>
<th>Mean</th>
<th>Median</th>
</tr>
</thead>
<tbody>
<tr>
<td>Length of Session (minutes)</td>
<td>13.00</td>
<td>1.70</td>
</tr>
<tr>
<td>Host CPU seconds used</td>
<td>10.1</td>
<td>3.0</td>
</tr>
<tr>
<td>Disjoint regions edited</td>
<td>3.7</td>
<td>2.0</td>
</tr>
<tr>
<td>Number of Edit Buffers</td>
<td>1.2</td>
<td>1.0</td>
</tr>
<tr>
<td>Characters typed by user</td>
<td>691</td>
<td>215</td>
</tr>
<tr>
<td>Characters output by host</td>
<td>Unknown</td>
<td>Unknown</td>
</tr>
<tr>
<td>Data Characters Output by Host</td>
<td>17,551</td>
<td>2,075</td>
</tr>
<tr>
<td>Unique Data Characters Output</td>
<td>5,449</td>
<td>1,176</td>
</tr>
<tr>
<td>Initial File Size</td>
<td>Unknown</td>
<td>Unknown</td>
</tr>
<tr>
<td>Change in File Size (characters)</td>
<td>8,163</td>
<td>0</td>
</tr>
</tbody>
</table>

*Figure 5-2: General Editing Statistics for 1,100 EMACS Sessions*

The updating of the screen and cursor positioning, and the remainder represent actual data characters. Particularly for the longer sessions, the same screens of the file are transmitted multiple times. The number of unique data characters represents those data characters that are not part of screenfuls that have been transmitted previously during the same editing session.
<table>
<thead>
<tr>
<th>Statistic</th>
<th>Mean</th>
<th>Median</th>
</tr>
</thead>
<tbody>
<tr>
<td>Length of Session (minutes)</td>
<td>4.46</td>
<td>1.36</td>
</tr>
<tr>
<td>Characters typed by user</td>
<td>273</td>
<td>106</td>
</tr>
<tr>
<td>Characters output by host</td>
<td>6,379</td>
<td>1,714</td>
</tr>
<tr>
<td>Data Characters Output by Host</td>
<td>6,379</td>
<td>1,714</td>
</tr>
<tr>
<td>Unique Data Characters Output</td>
<td>4,136</td>
<td>1,400</td>
</tr>
<tr>
<td>Initial File Size (characters)</td>
<td>27,454</td>
<td>7,000</td>
</tr>
</tbody>
</table>

**Figure 5-3:** General Editing Statistics for 179 MEDIT Sessions

The files edited are typically rather small. Figure 5-5 shows the distribution of file sizes. Like the editing session lengths, they have a Poisson distribution.

Figure 5-1 indicates that users tend to look at about the same amount of text regardless of the size of the file. Small files will usually be examined in their entirety. As the file size grows, users become more selective, however. For larger files, only a fraction of the total text is displayed by the user. Figure 5-6 is a scatter plot of the relationship between file size and the percentage of the file that is displayed. For larger file sizes, the percentage of the file actually displayed drops off rapidly. For files larger than 200,000 characters, a very small percentage of the file is displayed. The limited rate at which characters in the file can be displayed and the short duration of most editing sessions explains this phenomenon.
Figure 5-4: Distribution of Editing Session Lengths (minutes)
Figure 5-5: Distribution of File Sizes Edited (characters)
Figure 5-6: Percent of File Displayed vs. File Size
5.3. Programs used to Simulate Prefetching

Several programs were written to read the session data files produced by each editor and simulate:

- Conventional Demand Editing (without timesharing delays): M:0
- Editing with Local Memory: M:∞
- Various Simple Prefetch Methods with Local Memory\textsuperscript{15}
  - N:n
  - N:n P:n
  - D:n
  - N:n except that after a search command, the location of the next occurrence of the search string is prefetched rather than the next screenful
- The Ideal Prefetch Method with Local Memory

Each time the window shifts, the editor makes an entry of the form

\[ \text{L \ msec \ IDcount \ chrpos1 \ chrpos2} \]

in the session data file, where msec is the time of day in milliseconds at which the window shift was requested by the user, IDcount is the incremental count of characters inserted on the current window (before the shift) minus the characters deleted, and chrpos1:chrpos2 are the sequential character numbers within the file that have been output to the terminal as a result of the window shift.

For each method and each baud rate simulated, the simulation program maintains a linked list of chunks describing a contiguous range of characters of the file that would be present in local memory if that method were used, as shown in Figure 5-7. To process an L entry, the program first adjusts all chunk boundaries from the current chunk forward by IDcount. Next it locates the new current chunk.

\textsuperscript{15}These methods are described in Section 3.6.
Figure 5-7: Linked List Structure Used by the Prefetch Simulation Program

for chrpos2, either extending the range of an existing chunk, inserting a new chunk into the list if this range is not adjacent to an existing chunk, or coalescing two chunks if this range bridges the gap between them. During this step, the number of new characters added to the list (those not already present in the local memory) is computed.

To simulate the M= scheme, the number of new characters is multiplied by the time to transmit a character at the baud rate being simulated, to give the delay for this L entry using the memory method. To simulate N= prefetching, the correct number of characters forward of the previous chunk position is prefetched, depending on the time that elapsed between the previous L entry and the current one. After the prefetched characters have been added to the linked list, the characters in the L entry are inserted and the number of new characters is used to compute the delay as for the M= method. The ideal prefetch algorithm is simulated by doing the prefetching based on chrpos2, and by maintaining a variable containing the cumulative unused communications time so far, as presented in Figure 3-2 in Chapter 3.

As the delays for each prefetch method at each simulated baud rate are calculated for an L entry, they are added to the total delay for that method and baud rate for eventual calculation of the total and mean delay time. They are also time-quantized and placed into bins to construct an interval histogram (probability
distribution] of the delays for each method and baud rate. The results of the simulations are presented in the following section.

5.4. Results of Prefetch Simulations

5.4.1. MEDIT Sessions

About 180 MEDIT sessions were conducted at 9600 baud and simulated at various bandwidths from 300 to 9600 baud. The delay ratio using the ideal prefetch algorithm for these sessions is graphed versus the communications speed in Figure 5-8. Recall that a delay ratio of unity means that the simulated session at a lower speed works as well on the average as the actual session at 9600 baud. According to the graph, a user of a distributed editor that employed the ideal prefetch scheme would wait on the average the same amount of time for text to be transmitted on a 1200 baud communications line that a user of a conventional editor would wait on a 9600 baud line. This result applies to editing sessions in which non-video editing was used for about 50% of the session. Simulations of prefetching using a much larger collection of editing sessions on the two most popular video editors at Rutgers, TVEDIT and EMACS, are presented in the next section.

5.4.2. TVEDIT and EMACS Sessions

5.4.2.1. Normal Editing

Figures 5-9 through 5-11 summarize the results of the prefetch simulations of about 14,000 TVEDIT sessions using various schemes at communications speeds from 110 to 9600 baud. The results for the 1,100 EMACS sessions are almost indistinguishable so they have not been included as separate figures. The results obtained when the simulation was restricted to sessions in which only one kind of file was edited (e.g. English Text, Fortran programs, Lisp programs, Algol programs, etc.) are also almost identical to the general results for TVEDIT.
Figure 5-8: Delay Ratio vs. Communications Speed (baud) using ideal Prefetch Simulation (178 MEDIT sessions)

Figure 5-9 shows the mean delay time averaged over all groups of characters and all editing sessions. As expected, the simulated demand method produces the longest mean delay, followed by the memory method, then the simple prefetch schemes. The ideal prefetch method produces the shortest mean delay time. The point at which the dashed horizontal line at the 0.8 second mark intersects each curve shows the communications speed at which each method produces the same average behavior as simulated demand editing at 9600 baud. Local memory requires only half the bandwidth (4800 vs. 9600 baud) to produce the same average behavior. The simple prefetching schemes reduce the communications requirement by a factor of 5, to about 2000 baud. The ideal prefetch algorithm provides the same performance at about 1000 baud. These results for the ideal prefetching scheme are quantitatively very similar to the results.
for MEDIT presented in the previous section. Thus, ideal prefetching at about 1200 baud gives the same average behavior as conventional editing at 9600 baud both for video and line editing. Notice that while the relative improvement for both types of editing is the same, the absolute delay times for line editing are lower than those for video editing due to the greater use of communications made by video editors.

The demand curve in Figure 5-9 represents the theoretical behavior of conventional editing assuming no timesharing delays and no communications overhead. Aside from timesharing delays that depend on machine load, two major factors reduce the effective communications rate achieved by the video editors on the Rutgers DECSystem-20. First of all, the terminal communications provided by the TOPS20 operating system cannot output characters to terminals at a sustained 9600 baud rate, when averaged over a typical screenful\textsuperscript{16} of characters. Experiments we have conducted indicate that on a 9600 baud line, the effective terminal output rate is about 7000 baud, even when the machine is lightly loaded. Thus, users who think they are editing at 9600 baud actually experience delays at the simulated 7000 baud rate or even less, depending on the system load. However, communications lines nominally rated at below 7000 baud do in fact deliver their nominal rates.

The demand method curve in Figure 5-9 also assumes that the editor sends only data characters from the file to the terminal. In fact, TVEDIT must send additional characters other than text, most notably the colon character it uses to mark the beginning and end of each file line on the screen. The average screenful of text thus contains about 500 data characters plus 200 additional characters required to position the cursor and output the colons. Using a distributed editor, of course, such screen manipulations would take place locally. Thus the effective bandwidth of conventional editing reduces further from 7000 to 5000 baud. The top dashed line at about 1.2 seconds in Figure 5-9 represents conventional 9600 baud.

\textsuperscript{16}Most of our terminals display 24 80-character lines
Figure 5-9: Mean Delay Time in Seconds vs. Baud Rate
baud editing with adjustments for these two factors. It intersects the simple prefetch method curve at about 1200 baud, indicating that simple prefetching methods provide the same average delay time at 1200 baud that conventional editing provides at 9600 baud on the DECsystem-20.

Notice that the top dashed line does not take machine load factors into account. As the response time of the timesharing host degrades, the delay for all methods increases above the simulated values, but since conventional demand editing requires host interaction for every keystroke while distributed editors handle all local editing primitives themselves, heavy system load impacts conventional editing much more seriously.

The mean delay time represent average behavior only. To get a better feel for the response one can expect from a distributed text editor, we examined the population of delay intervals from which the mean values were computed. To get a feel for the performance of each prefetching scheme we integrated the interval histogram of the delay times to get the probability function \( P(\text{Delay} \leq n \text{ sec}) \), the probability that a delay interval randomly selected from the population will be less than \( n \) seconds. Figure 5-10 shows this probability (labeled \( P(D \leq n) \)) for 1200 baud simulations and Figure 5-11 shows \( P(D \leq n) \) for each of the methods for communications speeds from 110 to 9600 baud.

Figure 5-10 is an enlarged version of Figure 5-11E which serves as a typical example of the prefetch simulations. The point at which each curve intersects the vertical axis gives the probability that there would be no delay associated with displaying a given group of characters. For the demand method, this probability is zero, since characters must always be transmitted to update the screen. For the memory method, this probability is about 52%, which means that over half of the time that the current editing window is changed, the new screen consists of characters that have previously been displayed during the editing session. For the
simple prefetch methods, the probability of zero delay increases to about 73%, and
the ideal prefetch method gives no delay 90% of the time. All of the curves reach
close to 100% after about 10 seconds, due to the fact that few screen updates
involve more than about 1000 characters. By 16 seconds, all curves must reach
100% since that time is sufficient to transmit a full screen at 1200 baud.

Figure 5-11 shows the results for all simulated baud rates. In Figure 5-11A
we see that at 110 baud, the probability of no delay is the same as at 1200 baud
for the demand and memory methods, and the shape of these curves is about the
same as in the 1200 baud simulation. Notice however that time scale of the
horizontal axis is 100 seconds rather than 10 seconds. The simple prefetch
schemes work only slightly better than local memory alone because at 110 baud
there is no time to prefetch, even using the ideal prefetch method.

At 300 baud (Figure 5-11B), prefetching begins to show improvement over
the memory method. For the simple prefetch methods, the probability of zero
delay increases to about 68%, and the ideal prefetch method gives no delay almost
80% of the time.

From 110 baud to 1200 baud (Figures 5-11A through 5-11E) the demand
and memory curves remain the same in shape, though the time scale of the
horizontal axis decreases from 100 seconds to 10 seconds due to the increase in
communications speed. The simple prefetch and ideal prefetch curves shift
upward, however. Using simple prefetching, the probability of no delay has risen
from 61% to 73%. For ideal prefetching the improvement is from 70% to 90%.

Figures 5-11F through 5-11K show the continued improvement of prefetching
schemes as the communications speed increases. At 4800 baud in Figure 5-11H,
ideal prefetching produces no delay almost 100% of the time, while the simple
prefetch schemes produce no delay up to 85% of the time.
Figure 5-10: 1200 Baud Prefetch Simulations of 14,000 TVEDIT Sessions
Figure 5-11: Prefetch Simulations of 14,000 TVEDIT Sessions
Figure 5-11M compares various prefetching schemes at 1200 baud with the simulated demand method at 4800 baud. The 4800 baud demand curve crosses the 1200 baud simple prefetch curve at about 1.5 seconds and 82%, indicating that simple prefetching at 1200 baud will produce shorter delays than conventional editing at the simulated 4800 baud rate 82% of the time; the remaining 18% of the time it will produce considerably longer delays. Due to various inefficiencies associated with DECsystem-20 terminal output and screen manipulation overhead, the simulated 4800 baud demand curve closely approximates the actual behavior of conventional 9600 baud editing. Thus for 82% of the intervals, simple prefetching at 1200 baud will perform better than conventional "9600 baud" editing on the DECsystem-20.

5.4.2.2. Read-Only Editing

Figures 5-9 through 5-11 give the results of simulating various schemes using editing sessions in which the user actually changed the file. Most editing sessions at Rutgers involve file changes, but sometimes people use editors to scan on-line documentation or other files in read-only mode. Figures 5-12 through 5-13 show the results of simulating read-only sessions.

Comparing Figures 5-9 and 5-12 we see that all of the methods (including conventional editing) produce longer delays in read-only mode. The performance spread between various simple prefetch schemes is wider in read-only editing, with direction dependent (Dm) prefetching performing better than the prefetch-next (Nn) method. Following the upper dashed horizontal line in Figure 5-12 we see that it intersects the simple prefetch schemes at around 2000 baud, compared to 1200 baud in Figure 5-9. While prefetching improves performance considerably, 1200 baud is not quite sufficient to match conventional "9600 baud" editing. Notice that for read-only editing, local memory by itself hardly improves performance at all.

At 300 baud (Figure 5-13B) the prefetching schemes perform considerably
Figure 5-12: Mean Delay Time in Seconds vs. Baud Rate for Read-Only Sessions
better than the *demand* method; the probability is almost 40% that there will be no delay using simple prefetching. However, even the *ideal* method cannot prefetch text sufficiently quickly to avoid delay because it takes up to 64 seconds to prefetch a screenful of the file.

At 1200 baud (Figure 5-13E) the situation improves considerably using prefetching, while the *demand* and *memory* methods continue to perform poorly. Using simple prefetching, the probability of no delay increases to about 58%.

At higher baud rates (Figures 5-13F through 5-13L), prefetching performs increasingly well. At 9600 baud (Figure 5-13L) simple prefetching provides a dramatic improvement over both conventional demand editing and distributed editing with local memory only.

The differing behavior of normal editing sessions from read-only sessions arises from the nature of the editing primitives invoked by the user. The editing typically starts at the beginning of the file and proceeds toward the end without "backing up" to previous screenfuls. Search commands play an important role, since the user often locates the desired section using keywords or phrases. In reading the file like a hard copy document, the user typically moves by screenfuls rather than moving the cursor beyond the bottom of the screen, which would cause the screen to scroll up only a few lines. Thus the editor must display more characters during an average screen shift than it does during a conventional editing session, which explains the decreased performance of the *demand* method. Once a screenful has been displayed it is not usually redisplayed later during the same editing session, which explains the decreased performance of the *memory* method. Both of these factors affect the prefetching schemes as well, but as the communications speed increases, they are able to make up for the increased communications demand by *anticipating* needed text, similarly to the way a good tennis player anticipates the area of the court to which her opponent will return the ball.
Figure 5-13: Prefetch Simulations for Read-Only TVEDIT Sessions
5.4.2.3. Unnecessary Host Output

The *ideal prefetch* method never prefetches unneeded characters, but the simple prefetching schemes we have examined have no such guarantee. To ensure that they do not buy decreased delay at the expense of greatly increased character output from the host system, we measured the amount of wasted prefetching using the simulated prefetching methods.

Figure 5-14 shows the number of characters prefetched unnecessarily, expressed as a percentage of the characters that are fetched or prefetched and subsequently displayed. For the various simple prefetching schemes and baud rates, the amount of waste is very low, between 10 and 30% of the necessary output from the host. If no limit is placed on the amount of prefetching (e.g. N= M=) the amount of waste increases to about 58%.

Figure 5-15 shows the unnecessary prefetching that occurs during read-only sessions. For the limited prefetching methods, the waste is about the same as in normal editing sessions. If no limit is placed on the amount of prefetching (N=) the amount of waste increases to about 75%.

To get a fair comparison of the transmission overhead required for prefetching, Figures 5-14 and 5-15 should be compared to conventional editing, which produces wasted character transmission for two major reasons: redisplay of the same file characters more than once during the editing session, and various miscellaneous cursor motion and screen manipulation functions. This waste in a conventional editor typically amounts to 160%. Thus, despite wasted prefetching, a distributed editor would reduce considerably the total transmission of characters from the host.
Figure 5-14: Wasted Prefetching as a Percentage of Necessary Output
Figure 5-15: Wasted Prefetching for Read-Only Sessions
5.5. Host Interrupts

Common wisdom has it that video editors consume more system resources than line editors. The data collected from MEDIT and TVEDIT back up this claim. First, more characters are output to the terminal during a typical TVEDIT session than during a typical MEDIT session (13,166 vs 6,379 characters). More significantly, TVEDIT must read all characters typed to it using a monitor call that performs single character input, while a line editor can use a single monitor call to read an entire command or data line. During input of text, this difference causes the TVEDIT process to block (relinquish control to the operating system) almost two orders of magnitude more often than a line editor.

Programs that do single character input impose an extra burden on the host operating system because they must be rescheduled to run and process each character typed by the user. Unlike many other large timeshared computers, the DECsystem-20 is designed to provide reasonably fast response to user programs that read characters from the terminal one at a time, but this form of input still has considerable overhead compared with line buffered terminal input. Averaged over an editing session, users type approximately one character per second, so 50 simultaneous video editor users cause 50 context switches per second due to their keystrokes alone, or one context switch every 20 milliseconds.

To get a rough idea of the inefficiency of single character terminal input on the DECsystem-20, an experiment was performed in which 800 characters were input using a monitor call that reads a line at a time from the terminal (RDTTY) and using 800 monitor calls that read a single character from the terminal (PBIN). The RDTTY call took 0.3 second of CPU time, versus 0.8 second for the 800 PBIN's.

\footnote{TVEDIT sends many more characters to the terminal to update the screen than a line editor, primarily because it must "repaint" the colon character at the end of the line after each character inserted by the user, and also due to other screen manipulation suboptimalities. Note also that MEDIT is not a pure line editor; the figure for a true line editor would probably be somewhat lower.}
Thus, for every 10 lines of text typed by the user, approximately half a second more CPU time is used by a video editor than for a line editor, just because the video editor must use PBIN rather than RDTTY. Note that this figure does not include the added processing that the video editor must necessarily perform to update and ensure the integrity of the screen. Typically, for example, TVEDIT uses about 2.0 seconds of CPU time to input 800 data characters on blank lines, and about 8.1 seconds to insert 800 data characters into lines already containing text.\(^\text{18}\)

These figures do not take the system scheduler overhead for 800 process blocks into account.

A suitably designed distributed editor can provide video editing without interrupting the host frequently because it can handle user commands directly and need not interact with the host on every keystroke. When it does communicate with the host, the distributed editor can use a protocol that allows the host program to invoke line-at-at-time monitor calls (such as RDTTY on the DECSystem-20). Thus, as far as the host system is concerned, a distributed editor would eliminate the efficiency difference between line editing and video editing. In fact, the host would support more distributed video editing users than conventional line editing users.

Using a distributed editor, the host program must be scheduled each time a packet is received from the local system. To see how much of an improvement one can expect for distributed video editing, the number of packets sent to the host was estimated during simulation of simple prefetching. On the average, roughly 60 packets would be exchanged with the host during a typical editing session using a distributed editor that is optimized for sending the fewest number of packets. This figure compares favorably with the average of 1200 packets sent between the

---

\(^{18}\) When inserting characters on lines containing text, TVEDIT must redraw the colon character it uses to mark the right hand edge of the screen. This operation typically requires transmitting 10 characters to the terminal for each character inserted on the screen.
host and terminal during a typical TVEDIT editing session (600 user commands plus 600 screen updates). Thus, with a suitably designed distributed editor, the number of host activations could be reduced by a factor of 10.¹⁹

Processing commands locally isolates the user from timesharing and communications network delays. From the user's point of view, the host interrupts due to prefetching do not matter because delays in processing them do not affect the response to keystrokes.²⁰ The user will be aware of timesharing and communications delays only when she issues a command that cannot be processed locally. Most such commands will occur when the screen must be shifted to a new location in the file, and a small number of them will occur when performing global operations such as searching the file. However, using the simple N:n prefetching, most such delays can be avoided. Figure 5-16 shows the average number of window motion commands per editing session during which the user might potentially observe a timesharing delay directly, as simulated for each baud rate from the collected TVEDIT sessions. The average number of window motion commands in a typical session is 16.8, so at 1200 baud, for example, potential timesharing delays would be observable by the user for only 30% of the host interactions that take place. Notice that the delays that result from search commands and other global operations will probably not be as objectionable, because the user will expect these commands to take time to process anyway.

¹⁹Not 20, because there is one TVEDIT activation per command, not one per packet.

²⁰Except to the small degree that they decrease the effective rate at which text is prefetched.
<table>
<thead>
<tr>
<th>Baud</th>
<th>Interactions/Session</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>7.4</td>
</tr>
<tr>
<td>300</td>
<td>6.5</td>
</tr>
<tr>
<td>600</td>
<td>5.8</td>
</tr>
<tr>
<td>900</td>
<td>5.4</td>
</tr>
<tr>
<td>1200</td>
<td>5.1</td>
</tr>
<tr>
<td>1800</td>
<td>4.7</td>
</tr>
<tr>
<td>2400</td>
<td>4.5</td>
</tr>
<tr>
<td>4800</td>
<td>3.9</td>
</tr>
<tr>
<td>6000</td>
<td>3.9</td>
</tr>
<tr>
<td>7000</td>
<td>3.8</td>
</tr>
<tr>
<td>8000</td>
<td>3.8</td>
</tr>
<tr>
<td>9600</td>
<td>3.8</td>
</tr>
</tbody>
</table>

Figure 5-16: User-Observable Host Interactions using Simple Prefetching, for 14,000 TVEDIT Sessions

5.6. Analysis of File Change Descriptions

To compare various methods that can be used to transmit file changes back to the host, one must study the nature of changes made by typical users during typical editing sessions. The MEDIT statistics files provide this information about the kinds of commands typed by the user during the editing session and the changes they make to the file.

During a typical session, cursor motion commands account for a substantial fraction of the total commands typed. Depending on the command language of the editor and the existence of automatic repeat features on the terminal's cursor motion keys, cursor motion can account for as many as half of the keystrokes typed. Substantial economy of bandwidth is achieved in distributed editing merely by processing commands locally, since cursor motion need not be transmitted directly to the host. This section is concerned with additional economies that may be achieved by optimizing the file modification commands typed by the user.

The second session file produced by MEDIT contains information about line-by-line changes made by the user, in particular the sequence of editing primitives
(insert, delete, replace, and skip) invoked to effect the changes along with the original and changed line. From this information, one can compare the number of bytes needed to transmit the entire changed line, the unoptimized primitives, a heuristically optimized version of the primitives, the common head and tail heuristic, and the minimal string difference produced by the algorithm in Chapter 4. For the purpose of this comparison, line changes caused by inserting a new line were excluded since this case can be detected quite easily and the optimal difference encoding can be transmitted regardless of the method used.

Figure 5-17 shows the results of the comparison. Encoding the difference by transmitting the entire changed line requires an average of 32.5 bytes, which reflects the average line length. The raw user editing primitives were generated on the fly by the internal MEDIT primitives for insertion, deletion, and character replacement, one primitive per user keystroke. During insertion of text, each user keystroke translates into a single character insertion operator that requires 3 bytes, making this representation more verbose than transmitting the entire changed line. However, by applying the simple heuristic optimization of combining all identical adjacent operators, the user primitives require only 18.3 bytes to represent, which is considerably less than the 32.5 bytes required to transmit the entire changed line. Additional heuristics beyond the simple one used would further reduce this figure. The common head and tail difference method described in Chapter 4 requires only 14.4 bytes on the average. Both the heuristically optimized primitives and the common head and tail method perform reasonably well compared to the theoretical minimum difference description, which requires 12.8 bytes on the average. Notice that on a 300 baud communications line, a full second would be required to transmit the changed line, while only about 1/3 to 1/2 second would be required using the more optimal methods. For longer average line lengths, the difference would be even greater.

21 The MEDIT sessions from which the line change data were extracted consisted primarily of source program editing, so the number of characters per line tends to be smaller than for sessions in which documents are edited.
<table>
<thead>
<tr>
<th>Difference Method</th>
<th>Mean Bytes to Encode</th>
</tr>
</thead>
<tbody>
<tr>
<td>Send Entire Line</td>
<td>32.5</td>
</tr>
<tr>
<td>Raw User Editing Primitives</td>
<td>40.8</td>
</tr>
<tr>
<td>Heuristically Optimized Primitives</td>
<td>16.3</td>
</tr>
<tr>
<td>Common Head/Tail Optimization</td>
<td>14.4</td>
</tr>
<tr>
<td>Minimal Difference Encoding</td>
<td>12.8</td>
</tr>
</tbody>
</table>

Figure 5-17: Comparison of Difference Descriptions for 8500 changed lines

5.7. Discussion

The results of the previous sections provide strong evidence that some simple and easily implemented prefetching and file difference methods would decrease the communications delay during interactive text editing. Using a distributed text editor at 1200 baud with the limited "prefetch-next" algorithm, for example, the user will often (e.g. 75% of the time) experience no delay when moving to a new position in the file. When a delay occurs, it will usually result from an editor command that displays a section of the file not yet examined during the current editing session, and not adjacent to the current position. At such times, the user may wish to read through the current screenful more carefully than otherwise, to establish the context of the paragraph he wishes to edit. Thus he might not care that the screen is being updated at the communications speed rather than instantaneously at such times.

To further explore the behavior of prefeteching and the feel of editors that employ it, a prototype distributed editor was implemented using the DECSytem-20 as the host and a PDP11-based microcomputer with a byte-mapped screen as the local processor. Implementation of a prototype also fleshes out important practical factors such as communications reliability and the ability to recover from both host and local system crashes. The prototype is described in Chapter 6.
CHAPTER 6

THE PROTOTYPE DISTRIBUTED EDITOR: DISTED

This chapter describes the prototype distributed editor that I implemented based on the ideas of the preceding chapters. The prototype represents only one of a class of distributed editors that one could implement based on the simulation results, depending on the relative costs of communication, host CPU usage, and user delay times in a given application. It uses the simplest prefetching technique that works well in the simulation, namely prefetching the next few screenfuls ahead of the current position in the file. It also uses the simplest method for determining file changes on a line by line basis, the common head and tail method. Despite the choice of easily implemented schemes, the prototype editor provides significant performance improvements over conventional editors while presenting a conventional command interface to the user.

Building a prototype serves several purposes: it verifies that a distributed editor can provide a comfortable user interface that is as friendly and easy to use as state-of-the-art conventional editors, while at the same time providing greatly increased communications speeds and decreased timesharing and network delays. It fleshes out some of the potential difficulties that arise in distributing the editor between two distinct machines, such as program complexity and the robustness of communications protocols. Finally, it demonstrates that a distributed editor is not so complex as to be of theoretical interest only: the fact that I implemented the prototype on a microcomputer system in just a few months' time demonstrates that building a distributed editor is quite within the capability of an industry software house, provided that they have the required knowledge to make the necessary design choices.
6.1. Hardware and Software Used

6.1.1. Hardware

The prototype runs on a timeshared DECsystem-20 host and a dedicated local Terak PDP11/03 with a byte-mapped screen and 64K bytes of memory. The Terak also has a floppy disk on which the editor code is stored when the Terak is powered down, but which is not required during editing. The local program occupies less than 16K bytes, leaving room for about 30 screenfuls of the file in memory. If the prototype were to be marketed, the disk could be replaced by ROM to store the local program, thus reducing the cost of the required hardware to the range $1,000 to $2,000. A Z80-based system would also provide adequate computing power and address space to serve as a local processor. Thus the cost of the required hardware is competitive with the prices of ordinary terminals currently on the market.

6.1.2. Software Characteristics

The program on the DECsystem-20 (the editor server) is written in SAIL and uses the disk in much the same way that a conventional editor does, that is, it contains software to allow insertion and deletion of characters at arbitrary points in the file. Since neither the disk hardware nor the operating system provides direct insertion or deletion within a file, a text editor must simulate this capability by some means. The prototype editor server simply reads the file into random access memory during the editing session and performs insertion and deletion there. At the conclusion of the editing session, it writes the current state of the file out to form a new version of the disk file. It uses standard operating system calls of TOPS20 version 4 to perform all input/output, both to the disk and the Terak: it communicates with the Terak on a conventional full-duplex serial terminal line at baud rates between 110 and 9600 using a packet protocol with error detection and recovery.
The program on the Terak (the local editor) is written in BLISS-11. It runs stand-alone without using any external operating system calls. It handles input from the keyboard and input/output to the host via software interrupts and queues. Output to the Terak screen is byte-mapped and simply involves writing the desired characters to the screen memory.

6.2. User Operation of the Prototype Editor

When the user is not editing a file, the local software simulates a Visual 200 display terminal. All characters typed by the user are sent directly to the DECSYSTEM-20, and all characters received from the DECSYSTEM-20 are displayed immediately on the screen, scrolling it if necessary. Furthermore, all special cursor and screen control operations of the Visual 200 are simulated correctly. Thus the user may log into the host, read mail, and run programs normally as if using a conventional interactive terminal.

To edit a file using the distributed editor, the user invokes the editor server on the host just as a conventional editor would be invoked, e.g.,

```
@disted myfile
```

The DISTED program on the host sends a special command sequence to the local editor running on the Terak that causes it to start the local editing session. From this point onward, until the user ends the editing session, all input from the keyboard is processed directly by the local editor running on the Terak. When the editing session is over, the local editor reverts to simulating a Visual 200 terminal connected to the DECSYSTEM-20. The fact that editing operations are processed locally is transparent to the user, except that there are many fewer delays in processing commands, and the screen is refreshed at a rate considerably faster than 9600 baud.

\footnote{The Visual 200 is an upward compatible version of the DEC VT52 terminal, that also supports insertion and deletion of characters and lines.}
The local editor remembers the name of the file last edited and the time of
day at which it was last changed. If the user decides to re-edit that same file and
it has not been changed by another program since the previous edit, the local editor
will pick up the edit at the place the previous editing left off, without bothering to
refetch needed screenfuls from the host if they are already present locally. This
optimization avoids delay in the common case that the user edits the same file twice
in a row, especially at lower communication speeds.

6.3. Overview of the Interprocessor Software Communication

Once an editing session has started, the local editor assumes control of the
editing process: it reads commands from the keyboard, does whatever is necessary
to process them, and waits for the user to type more commands. When the user
issues a command that requires interaction with the host, the local editor calls on
the host as a subroutine by sending it a command packet. It then reads the host’s
response and uses it to process the user’s command. For example, if the user asks
to see a part of the file that is not present locally, the local editor issues a
command to the host to transmit the desired characters of the file, which it then
reads into its memory and displays on the screen. Thus the host server is very
much like a line editor that reads a command line, processes it, and waits for
further commands, except that it reads commands generated by a program (the local
editor) instead of commands typed by a user.

The use of an error-detection packet protocol is crucial to the operation of
the distributed editor because no mistakes can be tolerated in communication
between the host and local processors. For example, if a mistransmission of a
packet causes the local processor to expect to receive a packet from the host
when the host is also expecting to receive a packet, the communication will become
out of phase and the editing session will have to be aborted. If the host and local
copies of the file become different due to communications error, all subsequent
editing operations performed by the user will not perform the desired operations on
the file, and the user will be misled about the actual contents of the file. Though terminal communications are reliable enough to display information to a user, they are not 100% reliable and thus the ability to detect missing, extraneous, and incorrectly transmitted characters is required if they are to be used for communication between computers.

6.3.1. Packet Protocol

A simple but robust packet protocol is used for communication between the host server and the local editor over the serial terminal line. This protocol was chosen to make the probability of an undetected error very low while adding minimal overhead to communications. Automatic error correction was not used since it was judged that the probability of error is quite low and the retransmission of occasional packets would be quite tolerable. It is helpful at this point to outline briefly how the communication takes place.

Two kinds of packets are used, data packets and confirmation packets. A data packet consists of a two-character header followed by the contents of the packet, a checksum byte, and a one character packet terminator:

```
| DLE | STX | Data Bytes | Checksum | ETX |
```

Confirmation packets are two characters long, and either confirm or deny correct receipt of a data packet. The two possible confirmation packets are:

```
| DLE | ACK |
| DLE | NAK |
```

The protocol for sending and receiving packets works as follows. If the local editor wishes to send a command to the host server, it sends a data packet whose data consists of the command. It then waits for the host server to return an acknowledgement packet. If a NAK packet is returned, it retransmits the data packet again until it receives an ACK from the host. When it gets the ACK, it knows that the host has correctly received the command. If the command asks the
host to transmit something back, it then waits to read a data packet from the host. If the data packet is correct (i.e. if it begins and ends with the correct control characters and it has the correct checksum) then the local editor replies with an ACK packet and proceeds. Otherwise, it sends a NAK and waits to read the retransmitted packet.

6.3.2. Special Considerations

Because one of the reasons for a distributed editor is to reduce host load, the communications with the host was planned very carefully. The TOPS20 operating system provides a terminal input routine, the TEXTI JSYS, that meets the needs of the distributed editor quite well because it allows an entire packet to be input before the host program is awakened to process it, thus avoiding unnecessary scheduling of the editor server merely to read the next character of a command. However, the TEXTI JSYS interprets certain control characters as editing commands and the TOPS20 EXEC and monitor interpret others as interrupt characters. To avoid difficulty, none of these special characters are allowed in packets. These special characters are:

- **NUL (ASCII 0)** This character is often ignored or generated spontaneously by the terminal communications software.

- **Control-C** This character causes an interrupt when sent to the DEC-20.

- **Control-O** When the DEC-20 receives a Control-O, it purges its output buffer.

- **Control-Q** Control-Q and Control-S resume and suspend output to the terminal.

- **Control-R** The TEXTI JSYS echoes the current input buffer when this character is received.

- **Control-S** See Control-Q above.

- **Control-T** The TOPS20 EXEC outputs a line of process status information when this character is received.
Control-U

The TEXTI JSYS clears the input buffer when this character is received.

Control-W

The TEXTI JSYS deletes the previous word of input when this character is received.

RUBOUT (Octal 177) The TEXTI JSYS deletes the previous character input when this character is received.

In addition, only 7-bit ASCII codes were used because the TEXTI JSYS does not read 8-bit characters and also because certain network software has difficulty passing all 8 bits. The characters used to delimit packets were chosen to avoid the "problem" control characters above, as follows:

DLE = Control-P (ASCII octal 20)
STX = Control-D (ASCII octal 4)
ETX = Control-Z (ASCII octal 32)
ACK = Control-A (ASCII octal 1)
NAK = Control-Y (ASCII octal 31)

The checksum had to be defined carefully so that it would never come out to be one of the special unusable characters or one of the packet delimiter characters.

The algorithm for computing the checksum solves the problem by forcing the result to come out as a non-control character (octal 40 ≤ checksum < octal 177), as follows:

```pascalg
integer procedure checksum(string s);
begin
  integer checksum, i;
  checksum := 0;
  for i := 1 step 1 until length(s)
    do checksum := checksum xor s[i];
  if checksum < '40 then checksum := checksum + '40
  else
    if checksum = '177 then checksum := '40;
  return checksum;
end;
```

In addition to all other restrictions, the front end PDP-11 computer used by the DECSYSTEM-20 cannot handle sustained transmission of characters from the terminal,
even at moderate speeds. To avoid crashing the front end, the size of the packets sent by the local editor to the host had to be restricted to no more than 128 characters.

6.4. File Representation

Since the host server and local editor must communicate intimately about the file, it is convenient to use the same uniform and consistent representation of the file in both programs.

6.4.1. The Chosen Representation for Files

In the prototype, the file is represented as an ordered list of lines, where a line is a series of characters in the file terminated by a linefeed character (ASCII octal 12). Lines in the file are identified by a monotone increasing consecutive integer that will be referred to as the lineID. A page mark is considered a special kind of line that is assigned a lineID like any other line (since it is displayed on a separate line of the screen in the local editor) but that consists of a formfeed character (ASCII octal 14) with no terminating linefeed. As lines are added or deleted, the lineID’s of the remaining lines change dynamically. The local editor often needs to access a screenful of lines starting at a given spot in the file, and since it uses one screen line per file line, it can easily determine the lines of the file that it must have in local memory before it can refresh the screen.

6.4.2. Other Possible File Representations

Some alternative file representation schemes that might also be used are discussed here, along with the implementation difficulties they present. Any of these representations would be a perfectly valid means for a user to communicate with the local editor about the file. Indeed, the prototype editor uses relative lines within pages to allow the user to jump to another part of the file. The discussion here is limited to the utility of various representations for communication between the local editor and the host editor server.
Relative lines within a page are not used for communication purposes because they do not allow the local processor to determine the existence of gaps in the local copy of the file. For example, suppose that the local processor has the following lines of the file stored locally:

Page 1
- Lines 1:40
- Lines 80:120

Page 2
- Lines 1:40

It cannot tell whether there are any lines beyond line 120 on page 1. However, using the chosen representation, holes in the local copy can be determined with certainty:

- Lines 1:40
- Lines 80:120
- Line 121 (the page mark)
- Line 122:161

Typical line editors such as Wylibur [NIH 80] and SOS [Weiser 70] assign a line number to each line of the file. This line number does not change during the editing session so it uniquely identifies a given line. This representation has some advantages for communication between a user and the computer, but not for communication between two computers. If line numbers are stored between edits in the host file system, the host must transmit them to the local computer along with the text, thus slowing down the effective communications rate. In addition, this kind of numbering system provides even less information to the local editor about possible holes in its file representation than the relative line number method above.

Representing the file as an arbitrary sequence of characters allows files of arbitrary format to be created and edited. However, this representation poses two inconveniences for the current implementation. First of all, the mapping between character number and screen position is not well defined. The only way the local editor can tell what characters of the file it needs to fill up its screen is to ask the host, thus increasing the complexity of interaction between the two systems as well as the computational burden put on the host.
The second inconvenience posed by the sequence of bytes representation pertains to the chosen local processor, the PDP11/03. This processor performs integer arithmetic on 16-bit integers, so the largest sequential character number that could be conveniently represented is about 32,000. This number would limit the size of the largest file that could be edited unless special methods were devised to represent and manipulate larger numbers within the local editor.

These disadvantages by no means rule out this kind of representation for distributed editors in general, but they were judged sufficiently bothersome to rule out this representation for the prototype. An implementation of a distributed EMACS editor might well use this representation, however.

6.5. The Host Editor Server

The host editor server is a SAIL program that manipulates the file to be edited in much the same way as a conventional editor. It reads commands from the local editor in much the same way that a conventional line editor would read commands typed by a user. The host editor server's command parser is considerably simpler than that of a typical line editor, however, because it provides a much more rigid command syntax than a program designed for use by people. Furthermore, only a few general commands are provided rather than the large variety necessary to provide a friendly user interface. Most commands cause it to send output back to the local editor. The commands are described below. The program is quite short because all of the user interaction is handled in the local editor.

The host editor server is invoked on the DECsystem-20 much as any editor would be:

```
@DISTED filename
```

or

```
@DISTED/CREATE filename
```
6.5.1. Editor Server Commands

The editor server accepts the following commands.

=N Send the following FileName Date<EOL>nlines, where
nlines is the number of lineID's in the file and Date is the
date the file was last changed, to the nearest millisecond.
This information allows the local editor to determine if
the file previously edited is the same as the one about to
be edited, in which case it can reuse the lines already
present locally.

P lineID:lineID Transmit one or more packets containing the indicated
lines. Page marks are sent as formfeeds at the beginning
of a line, with the line that follows (if any) immediately
following the formfeed. The last packet transmitted
begins with "L", and all others begin with "M" to indicate
that more packets follow. Lines are not broken across
packets. The maximum packet size, including the start and
stop bytes and checksum, is 255.

D lineID:lineID Delete the indicated lines. No data packets are sent to
the local editor in response.

I lineID,nlines Insert nlines blank lines after lineID. lineID may be 0 to
insert lines at the beginning of the file. No data packets
are sent to the local editor in response.

F lineID,curchar pattern Search for pattern starting at curchar in lineID. Return
the lineID and first byte positions within the line where it
is found, e.g. "102,5", or "?" if it is not found.

T page[,line] Translate the given line and page number to a lineID and
transmit that lineID back. This function is used by the
local editor when the user gives a ":t" command.

W[filename] Write out the current edits to filename (default is the
input file). The editing continues. Send back a packet
with the complete file name, date, and number of lines, in
the same format as the =N command.

E Exit to the operating system. The local editor sends this
command to the host when the user asks to end the
editing session. It then simulates a terminal so that the
user can communicate directly with the host computer.

A lineID alters Perform the modifications contained in alters on lineID.
Alters contains a concise notation for DELETE, INSERT, REPLACE, and location-setting commands, as follows:

\(<\text{rpt}\>\) Move forward \(<\text{rpt}\>\) characters in the current line.

\(<\text{rpt}\>\text{D}\) Delete the next \(<\text{rpt}\>\) characters in the line.

\(<\text{rpt}\>\text{xxx}\) Insert \(<\text{rpt}\>\) characters.

\(<\text{rpt}\>\text{Rxxx}\) Replace \(<\text{rpt}\>\) characters, i.e. same as \(<\text{rpt}\>\text{D}\)<\text{rpt}\>\text{xxx}\).

\(<\text{rpt}\>\) is represented in decimal.

6.5.2. Host Crash Recovery

No explicit crash recovery mechanism has been included in the prototype host editor server to protect against host crashes. In a production system, a mechanism such as found on conventional text editors should be supplied to save the file on disk after a certain number of changes have been made or a certain amount of time has passed. This problem has been solved in a variety of ways by existing conventional text editors and does not represent a problem peculiar to distributed editor servers.

6.6. The Local Editor

The local editor is a BLISS-11 program that provides editing capabilities similar to most video editors and word processors. It reads and processes all keystrokes made by the user on the keyboard, keeps the screen updated as necessary, and communicates with the host editor server when necessary to inform it of changes made to the file and to fetch or prefetch needed lines of the file into local memory. Since it must interact with the user, its command parser is much more complex than that of the server. It also provides many more commands.
8.6.1. Software Organization

The local editor is organized into the following distinct components:

- **Keyboard I/O**
  Reads characters from the keyboard at interrupt level and queues them to be returned to the main program when it asks for keyboard input.

- **Serial I/O**
  Reads characters from the serial input port from the host system (at interrupt level) and queues them for later access by the main program. Reads characters from the output queue and sends them to the host (at interrupt level) whenever the host's serial port will accept another character. These routines implement low level single character I/O and are used by the Packet Routines.

- **Packet Routines**
  These consist of two routines, SendPacket and ReadPacket. SendPacket takes a string as an argument and sends it encoded as a data packet to the host, making sure that it was received correctly before it returns. ReadPacket waits for a correctly sent packet from the host and returns the data it contains as a string.

- **V200 Simulator**
  Simulates a Visual 200 terminal whenever the user is connected to the host but is not running the distributed editor. The V200 simulator also provides screen manipulation functions (e.g., insert character, delete line) that are used by the Screen Editor to update the screen as necessary.

- **Screen Editor**
  Includes the command parser, screen update routines, local file representation, editing primitives, and procedures to implement each command recognized by the parser. Some of the editing primitives work by using the Packet Routines to communicate with the host. The screen editor is implemented the same way a conventional video editor would be, except that it communicates with the host instead of with disk files to get parts of the file not already present in memory. Also, it keeps careful track of what it has in memory so as not to request any lines that it has already. It represents lines of the file as a linked list of records, with each record containing the lineID of that line as well as the text.

- **Background Process**
  Performs prefetching of lines from the host, and sends local file changes back to the host, using free local processor time between user commands. Prefetching in the background allows the user to continue editing locally while the host is transmitting portions of the file. Sending local changes back in the background prevents delays when the user has made changes to the file and the host does not respond immediately with an acknowledgement of the changes.
All user commands that act on the current screen of text are processed locally. When a line has been changed and the user moves to another line, the editor compares the old version of the line with the edited version and sends a description of the difference to the host using the common head and tail heuristic. When the user moves the cursor to a screenful that is not in local memory, a command is sent to the host to fetch the required lines of the file.

6.6.2. Prefetching

Prefetching is done in the background as the editor runs. Any editing operation that requires communication with the host must first wait for any background prefetching in progress to terminate before it can use the communication line for its own needs. Therefore, prefetching is done one line at a time so that it will not monopolize the communications line for too long. This design choice increases the number of prefetch packets sent to the host considerably, however.

Whenever the editor asks for a character from the keyboard, prefetching is initiated if no character has been typed and some lines need to be prefetched, i.e. the next few screenfuls forward of the current location in the file are not already present locally. If prefetching is already going on and a packet has been received from the host, the packet will be read and processed at that time. Thus while the user is actively typing commands, the editor will process them first. As soon as he pauses, the editor will initiate or continue processing prefetching. If he types a command that requires communication with the host and prefetching is in progress, the prefetching will be completed before processing the command. This may cause a delay that is potentially as long as the time it takes to send one packet from the host to the local editor at the communications speed in effect (e.g. 0.7 second at 1200 baud). If the local editor is cleverly coded, this delay need only occur when the user asks to access parts of the file that have never been fetched, in which case the delay will be inconsequential compared to the time required to fetch the new screenful from the host.
6.8.3. Local Editor Commands

The commands provided by the local editor fall into two general categories, local editing commands and global access commands. Local editing commands include cursor motion on the current screen and commands that effect insertion and deletion of characters on the current screen. Global access commands include commands that shift the current screen to another part of the file, such as the next window and previous window commands, and commands that search for a given text string within the file. All of the local commands are implemented strictly in the local editor, without communicating with the host\textsuperscript{23}. Commands that search the file and that jump to a specified line or page within the file are implemented by communication with the host.

The user interface of the prototype is almost identical to the video editing mode of the MEDIT editor implemented by the author. An interface more like TVEDIT or the standard commands of EMACS could also have been chosen. Commands to the editor are normal alphabetic characters. To insert text, the "I" command is used to enter insert mode, and the ESCape key is used to return to command mode. Commands can be preceded by a numeric repeat count (-,0,9) or the "W" (word/window) modifier to cause them to act on the current word or window of the file.

<table>
<thead>
<tr>
<th>Command</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>WN</td>
<td>Move to Next window</td>
</tr>
<tr>
<td>WU</td>
<td>Move Up to previous window</td>
</tr>
<tr>
<td>SPACE</td>
<td>Move cursor forward a character</td>
</tr>
<tr>
<td>&lt;BS&gt;</td>
<td>Move cursor backward a character</td>
</tr>
<tr>
<td>N</td>
<td>Move cursor to Next line (&lt;CR&gt; and &lt;LF&gt; also work)</td>
</tr>
<tr>
<td>U</td>
<td>Move cursor Up to previous line</td>
</tr>
<tr>
<td>B</td>
<td>Move cursor to beginning of line</td>
</tr>
</tbody>
</table>

\textsuperscript{23}Except to inform the host of changes that have been made on the current line whenever the cursor is moved to a new line.
TAB     Move cursor to end of line
S       Skip to a specified character
I       Enter INSERT mode to insert text (ESC ends)
X       eXtend the line (move to end and enter INSERT mode)
D       Delete next character(s)
<DEL>   Delete previous character
↑W      Delete previous word
R       Replace character (Same as D followed by I)
C       Enter CHANGE mode to write over old contents (ESC ends)
K       Kill (delete) up to a specified character
Z       Zap (delete) to end of line and enter INSERT mode
T       Transpose characters
V       inVert case of character
↑D      Delete line
↑F      Find a string
↑L      Insert a line below current line
↑U      Restore current line to original
?       Enter HELP mode
.       Go to a specified page, line of the file
<       Convert character to UPPER case
>       Convert character to lower case
↑E      End edit and write file
-↑E     End edit without writing file

6.6.4. Local Crash Recovery

In the event of a local crash in the middle of an editing session, the user may reboot the Terak, run the local editor, and hit a special key sequence (Control-Shift-Null) to pick up the editing session from where it left off. Since all changes to the file are reported to the server on a line-by-line basis by the local editor, the newly started copy of the local editor simply asks the server to transmit the needed screenfuls of the file, and thus allows the user to continue editing the file.
6.7. Experience of Use

The distributed editor prototype was not developed into a production editor and thus has not been used extensively for daily editing by many users. For the sake of expediency, the string space garbage collector was not implemented, the maximum line length was limited to the screen width (80 characters), and control characters (other than standard carriage control) within the file were disallowed. While necessary for a production editor, these features do not add to the understanding of distributed editing. Despite these shortcomings, the prototype was developed sufficiently to be tested against conventional editors and to verify qualitatively that it significantly improves editing throughput, even when used on 1200 baud communications lines and over packet switched networks from one coast of the United States to the other.

Subjectively, the distributed editor feels much more responsive than a conventional editor. One notices immediately that all cursor motion commands and local changes cause instantaneous action on the screen, regardless of the load factor on the host machine. Thus, editing the current screen of the file becomes indistinguishable from conventional editing on a lightly loaded host at 9600 baud, even when accessing the host over a network connection.²⁴

Whenever the user moves beyond the end or beginning of the current screen, or issues a search or jump command to move to another part of the file, the screen must be shifted forward or backward to a different section of the file. In a conventional screen editor, the host must send the new lines directly to the terminal's screen. The characters appear on the screen at the speed of the communications link, which is perceptibly slower than immediate even at 9600 baud.

²⁴I experimented with DISTED running over long distance ARPANET connections that produced a considerable delay of character echo when using normal programs at the host machine. The improvement using DISTED due to the immediate echoing of characters makes a considerable difference in editing productivity.
and can take up to 20 seconds at 1200 baud on typical 80 character by 24 line screens. Furthermore, timesharing or communications network delays often cause the output to pause at irregular intervals (stuttering) that is not only annoying to watch but also increases the time to refresh the screen. When using the distributed editor, the host need not be invoked to shift the screen if the new screen has been previously displayed or prefetched. In either of these cases, the screen is updated instantaneously, regardless of the rate of communication between the host and local computers.\textsuperscript{25} According to the simulation results at 1200 baud, 75 to 80\% of new screenfuls will be in local memory. In practice, one notices quite dramatically that moving back to previously seen screenfuls and forward again can be accomplished very quickly. Furthermore, on a 1200 baud communications line, prefetching occurs fast enough that the next screen is usually prefetched by the time the user has read the current one. Thus when using the editor to read sequentially through an online document, the user rarely waits to see the next screen and never waits when going back to previous ones.

Before the screen refresh begins in the local editor, it first ensures that all lines that must be displayed are present locally. If not, needed lines are fetched from the host and the message “fetching from host” appears on the mode line of the screen. When all lines to be displayed are present locally, they are copied to the byte-mapped screen memory. This design choice isolates the user from stuttering and gives the illusion of rapid response, even when the required text takes a while to reach the local editor. Alternatively, the local editor could have been designed to display characters on the screen as soon as they are received, or to display complete lines as soon as they are received. The distributed editing architecture gives the local editor designer complete freedom in this regard, unlike conventional editor architectures that require display of characters received from the host as they arrive at the terminal.

\textsuperscript{25}On the Terak hardware, the effective rate of copying local memory to the screen exceeds 20,000 baud
6.8. Possible Design Improvements

A guiding principle in designing the local editor is to perform as much of the computation locally as possible. Besides reducing the host overhead, performing tasks locally reduces delays in responding to user commands due to timesharing and network overhead. Furthermore, the computing power of most microprocessors is quite sufficient to perform all common editing operations in real time so there is no particular reason to use a more powerful machine except when an operation must be performed on a part of the file that is not present in local memory.

The prototype local editor was coded to avoid interaction with the host in the most common cases but not in every case where it could have been. The major cause of delay in the prototype occurs when lines are inserted, deleted, or changed. The editor attempts to communicate the changes to the server synchronously and does not continue processing commands until the host server has confirmed the receipt of the packet. By queuing up the change packets and processing them in background mode similarly to the way prefetching is processed, these delays could be avoided and the user would experience no delays during local editing of the current screen.

The search commands also cause unnecessary delay in the case that the search string is present within the local memory. Currently, the editor always asks the host to perform the search when the user issues a search command. The work of the host can be decreased, and potential timesharing and communications delays avoided, if the local editor searches for the string as far forward in the file as it can until it either finds the string or else reaches a gap in its local copy of the file. If a gap is reached, it can then ask the host to continue the search and report the location of the next occurrence of the search string. Similar optimizations can be applied to the command that jumps to a given page of the file.

Many possible user interface modifications might be made to reduce the
probability that the user will have to wait for adjacent screenfuls to be displayed. For example, if the user issues a command to view the next screenful, the editor currently advances the screen 20 lines forward. If at that point only the next 16 lines have been prefetched, the user has to wait for the missing 4 lines to be fetched before the screen is displayed. Instead, the editor might advance the screen by only 16 lines to avoid the delay. Similar considerations apply to the action taken by the editor when the user issues cursor motion commands to move the cursor to the line beyond the beginning or end of the current screen. One might adopt a conservative screen shifting routine in this case to minimize the probability of making the user wait, depending on how many adjacent screen lines are present already in local memory.

Prefetching is done one line at a time to minimize the time it takes to stop prefetching when a user command requires interaction with the host. This design choice requires the local system to send 48 prefetch commands to prefetch the next few screens of the file, however. To reduce the number of prefetch packets, the prefetching command protocol can be modified so that the local system can issue a single command, following which the host system will begin to transmit text in reasonably small (e.g. 100 to 200 character) packets. If the local system responds with a NAK packet, the host will cease sending text packets. To reduce the number of packets even more, a special Interrupt character could be supported by the host program in conjunction with the use of very large text packets. To halt a prefetch operation, the local system would transmit the interrupt character and continue to read the prefetch packet. The host, meanwhile, would terminate the current packet prematurely (but with the correct checksum).
6.9. Conclusions

The prototype distributed editor has demonstrated that

1. A distributed editor can provide a user interface that is at least as
   friendly as a conventional editor.

2. The simulation results presented in Chapter 5 are qualitatively accurate,
   that is, there is indeed an effective speedup of communications due
   to local memory and prefetching.

3. Processing editing commands locally eliminates most of the response
   delays that result from heavily loaded timeshared hosts and network
   communications.

4. Reliable communication between the host and the local system can be
   accomplished with a relatively simple packet protocol with error
   detection and recovery.

5. The task of implementing a distributed editor is not gargantuan. The
   prototype was implemented by a single programmer (under
   considerably less than ideal circumstances) in a few months.

6. An LSI-11/03 CPU with 64K bytes of memory is sufficient for the
   local processor, and a Z80 system would provide similar capability.
   Thus the cost of the local editor hardware falls in the range of
   $1,000 to $2,000, or about the cost of an inexpensive terminal.
   Many terminals already contain the hardware required for a distributed
   editor, except that they lack sufficient random access memory.

There is potential for improved editing capabilities beyond what current editors
supply. Various status information can be provided on the screen, even if it
changes with every keystroke. For example, every keystroke could be echoed in a
special area of the screen (in addition to performing its desired function) to help the
beginner learn the command set. Current video editors avoid making changes to the
screen on each keystroke because slow communications lines and timesharing delays
make such changes asynchronous with the keystroke and hence distracting.

As they become available, more powerful microprocessors with larger address
spaces will provide the local processor with increased local memory (to store more
of the file) and additional computation power to provide increased editing
functionality that might be too expensive to provide in a conventional timeshared
editor. A system based on the Motorola-68000 CPU, with 256K bytes of RAM, a
10M byte hard disk, a floppy disk, and an 8 megahertz clock is currently available for about $8,500. This system without the disks would provide a surfeit of memory and CPU power at a price that would be competitive with the more expensive conventional terminals. Alternatively, parallel copies of the files could be maintained both on the local disk and on a timeshared host. The user would be able to edit documentation and other files shared in common through the host as if they were present locally. Any files created locally could optionally reside both locally and on the host, allowing the user to process them using local compilers while at the same time sharing them with others through the host's file system.
CHAPTER 7
CONCLUSIONS AND FUTURE WORK

This chapter reexamines the design decisions faced by the implementor of a distributed text editor in light of the results of the detailed quantitative study of editor usage by the Rutgers LCS/DECsystem-20 user community described in Chapter 5 and the experience of implementing the prototype distributed editor described in Chapter 6. While only one user community was studied, we have no a priori reason to suspect that other communities would exhibit radically different behavior. Such factors as the average file size probably depend somewhat on the user community but do not directly affect the results. Neither the editor command set nor the types of files edited critically affect the simulated sessions.

Throughout this thesis, the following scenario is assumed about the environment in which the distributed text editor is used:

- The file resides at a "host" computer and must be updated there when the editing session is over. Computing time on the host is "cheap". Also, interrupting the host (e.g. by sending it input) is costly.

- The user performs editing at a dedicated "local" processor where computation is "free".

- The "local" and "host" systems are linked via a communications line that has a low bandwidth compared to the speed of either processor (e.g. 300 to 9600 baud) but high compared to the rate at which the user types. The user can perceive the delay in displaying a screenful of text if the text is being passed over the line. The cost of the line may depend on usage time, number of packets transmitted, bandwidth, and other factors.

However, many of the results obtained should prove useful even in potential distributed editing applications for which some or all of the above assumptions do not hold.
7.1. Design Choices

Because the architecture of a distributed text editor depends on many factors, including the constraints of existing hardware and communications facilities as well as the mix of editing tasks for which the system will be used, presentation of a single design would not be appropriate. However, the space of possible designs can be narrowed significantly in light of the empirical study and experience with the prototype distributed editor.

7.1.1. Transmission of Text

A simple text prefetching scheme and a modest amount of local memory provide greatly increased editing performance over conventional editing using low bandwidth communications lines. Simply prefetching the next few screens of text when the communication line would otherwise be idle works almost as well as the theoretical ideal prefetch algorithm. While more complex predictive and adaptive methods of prefetching might improve performance slightly, the effort involved to implement them would probably not justify the performance improvement. However, as discussed in Chapter 6, it may be desirable to adjust the windowing algorithms of the editor slightly to take maximum advantage of the lines currently in local memory and thus reduce the probability that the user will have to wait for text to be transmitted.

While much less critical than transmission of text from the host to the local system, transmission of file changes from the local processor to the host will impact editor performance and should be minimized. Some simple heuristics for determining file differences provide encodings that are almost as good as the minimal difference description. The simplest heuristic, the common head and tail method, typically uses only 12% more characters than the minimal description, while transmitting each changed line in its entirety uses 150% more characters. However, the common head and tail method would not work well to describe certain special
local editing changes, such as justification of an entire paragraph or screen. In this case, the heuristic of transmitting the actual editing primitives used by the local editor to perform the justification would provide significantly more optimal results. The computational overhead for both simple heuristics is so low that both could be carried out and the best result used to transmit the difference to the host.

7.1.2. Local Memory

According to the empirical data, the user displays about 5,000 unique file characters (roughly 10 average screenfuls) of the edited file during a typical session. Thus, a small amount of local memory would be sufficient for many editing sessions. For longer editing sessions, during which more of the file may be displayed, some mechanism for throwing out portions of local memory that are least likely to be redisplayed will have to be implemented. For example, the portions furthest away from the current location in the file could be eliminated. Portions earlier in the file should probably be thrown out before those closer to the end because users move forward through the file more often than backward. Additional local memory (e.g. 30 screenfuls) would make the decision of which portions to throw out less frequent and less critical to the performance of the editor.

If the local processor serves as both a distributed editor and a general purpose work station, its disk can be used for additional memory, both during an editing session and between sessions. By retaining the local copy of files edited on its disk, the editor would almost always have needed screenfuls locally and could thus provide very fast response even over 300 baud lines. Even without a disk, the local copy of the most recently edited file can be stored in random access memory between editing sessions. If the user edits the same file twice in a row, a frequent occurrence according to the collected statistics, the parts examined in the previous session will be available locally for immediate viewing.
7.1.3. Where should each editing primitive be processed?

Computing time on the microprocessor is "free" in that there is no need to limit it except as it impacts response time to user commands. Furthermore, inexpensive microprocessors have more than ample power to perform local editing functions. One of the goals of distributed editing is to reduce the number of host interrupts and the computational load on the host. These considerations lead to the following general principle of distributed editing: All editing primitives that can easily be performed directly by the local computer without communicating with the host should be processed locally.

The editing primitives provided by typical text editors fall roughly into three categories:

- **Local editing primitives** that effect changes to the current screen.
- **Global search and random access** that let the user move to a specified line or character directly without displaying the text between the present and desired locations.
- **Global changes** applied to most or all of the file without displaying all changed lines.

In typical editing sessions, the vast majority of commands issued by the user invoke local editing primitives to change the current screenful of text, and of these, a large fraction simply move the cursor. There is no reason to invoke the host system to process these primitives.

The second category of editing primitives, those that perform global search and random access to the file, cannot always be performed locally. For example, a search for a specified pattern in the file may require that the entire file be searched. The local processor can start searching until it either finds the pattern in the local copy of the file or it finds a gap. If it finds a gap, the host can be invoked to continue the search on any parts of the file that are not present locally since transmitting the entire file to the local system takes too long.

---

26 Except to describe file updates from time to time.
The third category of primitives must almost always be processed at the host since they involve access to most of the file but do not require that the file be displayed on the screen. Because the host may change parts of the file that are present locally, some method must be used to ensure local integrity of the file. Depending on the nature of the changes, the host might transmit a summary of changes made to the portions of the file stored locally, similarly to the way the local processor sends a summary of editing changes to the host. If the change descriptions are too complex, it might be better for the local system simply to forget its local copy of the file following the global change.

7.1.4. How should the processors communicate?

Due to the close interaction between the two systems, no undetected errors can be tolerated during communications. Terminal communications lines designed for use by people usually do not meet this requirement so if they are to be used, software error checking and recovery must be included in the host and local software.

Experience with the prototype editor described in Chapter 6 indicates that a simple packet communications method provides the needed error recovery with a relatively low communications overhead. Alternatively, if the host system provides hardwired processor networking capabilities and no phone line connections are used, the communications within the distributed editor can be simplified by removal of the packet protocol at that level. In either case, care must be taken to handle non-standard control characters that may appear in packets, since transmission of these characters may not be allowed or may be interpreted as special commands or interrupts by the host.
7.2. Summary of Hardware Requirements

The basic hardware required for the distributed text editor is currently available at moderate cost. The required hardware consists of at least:

- A dedicated local microprocessor with roughly the capabilities of a PDP11/03 or Zilog 80 and sufficient high speed memory to store considerably more than a screenful of text in addition to the editing program. Many editing sessions require only about 5000 characters of memory, or roughly 10 average screenfuls of text. An 8-bit microprocessor with 64K bytes of memory will accommodate about 20 to 30 screenfuls of text in addition to the editor program, assuming a 24 line screen with 80 characters per line. More powerful processors with larger address spaces such as the Zilog 8000 or Motorola 68000 are not needed to drive a distributed editor, though the larger address space and added CPU power would facilitate the implementation of fancier local editors.

- A video terminal (screen and keyboard) connected to the microprocessor. The communication interface between the terminal and microprocessor should ideally be limited only by the ability of the microprocessor to compute, as provided for example by a memory mapped display. A typical screen is currently 24 lines, 80 characters per line, but larger screens will likely become standard in the future. The need for distributed text editing will grow as the screen sizes become larger because it will take longer to update a larger screen over low bandwidth communications lines using conventional methods. Similar considerations hold if the set of possible characters that can be displayed increases significantly.

- A host computer with file storage devices.

- A two-way communications link between the host and the local processor. Phone line data communications links of 1200 baud, currently widely available, can provide excellent response time in many editing situations. 9600 baud links can provide essentially instantaneous response without communications network or host timesharing delays.

Microprocessor systems with memory, video monitors, and keyboards are currently available as units in the $1000 to $2000 range. At the current cost of semiconductor memory, about $200 for 64,000 characters, and with the projected decrease in memory cost, it is clear that providing a local processor with enough memory to store every character of the file that the user wishes to view during an editing session will not be prohibitively expensive in the very near future.
7.3. Summary of Software Requirements

A distributed editor requires at least two distinct software programs, a local editor program that runs on the local processor and interacts with the user, and an editor server that runs on the host and interacts with the local editor over the communications line. The local editor has most of the features of an ordinary text editor, except that it does not work directly with a disk file. Instead, it calls the host editor server somewhat like a subroutine to fetch needed parts of the file and to perform editing primitives that involve the entire file (such as searching for patterns). The host editor server consists of a disk file manager and a very limited command interface that responds to commands given by the local editor. Both the local and host programs must have a packet sending and a packet receiving subroutine to encode and decode packets and handle errors in transmission.

7.4. Summary of Performance Enhancements

The distributed editor provides increased effective communications bandwidth between the user and the file, isolates the user from most of the host timesharing and communications network delays, and significantly lessens the communications and computational load on the host CPU. Using a 1200 baud communications line, the user of a distributed editor would wait the same amount of time for the screen to be written as he would at 9600 baud using a conventional editor. Observation of the prototype indicates that the distribution of delays would be more predictable and acceptable to the user of a distributed editor, moreover. Furthermore, the number of times the user would have to wait directly for host response (and thus would notice a timesharing pause or network delay) is reduced by considerably more than a factor of 10, since the number of host interruptions to process input is reduced up to an order of magnitude compared to conventional video editing, and most of the remaining interruptions occur in the background when prefetching packets are sent. Host delays responding to prefetching packets would not be observed by the user.
7.5. Future Research and Development

Distributed editors have great potential in the coming decade. The number of home and business computer systems grows rapidly and the need for communication and sharing of information becomes more acute with each new system installed. A potentially large market exists for a simple but powerful distributed editor. The host server is simple enough that it could be implemented in a transportable language to run on all the popular large timesharing systems. Local editors could run on existing microprocessors or on networks of personal workstations. Alternatively, they could be implemented on specially designed hardware and marketed as replacements for conventional terminals. Such “super-intelligent” terminals would enable large mainframe equipment manufacturers to offer state of the art text processing facilities, even on half duplex systems.

The same host server could be used to support a wide variety of local editors with completely different user interfaces. This architecture would remove the burden of supporting many different editors from the staff of the large systems, while at the same time providing increased choice of editor command languages to users. Furthermore, with a standard host editor server, the same editor would be available to a user on every computer system, obviating the need to learn a different editor for each host machine.

As dedicated workstations become more prevalent, the sharing of data that we currently take for granted on timeshared hosts will become more difficult. Methods of shared file access in networks have been and are being investigated [Segall 76, Crocker 78], including the use of network file servers. The host editor server could be incorporated into a network file server, thus providing distributed editing capabilities so that any node can edit files on any other node. Further research is needed to explore distributed editor design issues in networks with multiple nodes,
particularly when multiple copies of the same file may reside at more than one node or a single copy of a file may be split between nodes. In one simple case, the user of a personal workstation may wish to maintain parallel copies of the same file on his local disk and on a centrally accessed file system, and also have the capability to examine files on the central system. A distributed editor architecture could be used to automatically ensure that the central system's copy of locally edited files are updated as necessary.

The general problem of assuring that multiple copies of the same file at different nodes in the network are identical will probably become important. Hashing techniques have been suggested to quickly check that two files at remote sites are identical, but if they are not the same, the differences must still be determined. To this end, further investigation of better time and space performance of minimal string difference encodings, as suggested in Chapter 4, would be useful. If a minimal string difference encoding algorithm marginally efficient to work on small files were developed, it could be used to benchmark various heuristics for encoding file differences to determine which ones are best suited for the kinds of file changes that users make in practice. The resultant algorithms would also facilitate keeping online edit histories of shared files automatically.

The trend in text processing systems is toward increasing the available character set beyond the 128 ASCII characters, enlarging the size of editing screens, and providing graphics capabilities. These applications will require higher communications bandwidth, increasing the potential payoff of distributed editing. A study of distributed graphics and text processing usage should prove interesting, judging by the results obtained in this thesis about distributed editing of ASCII text.

A potential performance enhancement exists beyond the prefetch methods described, using various coding techniques to encode textual material into a more compact form. This enhancement is orthogonal to the methods described in that it
can be applied to the text being prefetched rather than in place of prefetching. For example, Huffman coding techniques can be used to reduce the number of bits needed to encode the most common characters.

Another data compression technique that would be very effective for encoding recognizable English text would provide short abbreviations for the most common and frequently occurring words in the text. A classical study of the frequency distribution of words in written English documents, [Kucera 67], has shown that a small set of words occur with very high frequency in most written English documents. Furthermore, there is considerable overlap of the set of most common words among different modern language authors.

Zubkoff [Zubkoff 81] at Carnegie Mellon University analyzed over 13 million words in on line documents and built a dictionary of the most common 1024 words. He modified the software in an existing terminal\textsuperscript{27} to use single control characters to encode the 32 most common words (e.g. "the") and a two character sequence to transmit the remaining common words to achieve further compression. In addition, he kept a dynamic dictionary of the most recently transmitted tokens. By using a special editor at the host that knows how to take advantage of this special terminal software, Zubkoff claims to achieve typical compression ratios of 2.5 to 1 using this scheme, and as high as 4.0 to 1 when editing BLISS source code, presumably because of the large number of occurrences of a restricted set of programming language keywords. Applying this 4 to 1 compression on top of the 1200 baud to 9600 baud improvement using prefetching suggests that a system that combined a distributed editor with a text compression method could potentially deliver 9600 baud performance on a 300 baud phone line!

The methodology used throughout this thesis to explore software design and

\textsuperscript{27}The Concept 100
architecture produced results, both empirical and analytical, that led to a better understanding of the design decisions inherent in the construction of one kind of distributed text editor. Beyond its impact on distributed editing, this thesis serves as an example of what can be achieved in computer science by combining carefully planned and carried out experiments with analytical and algorithmic analysis of a problem area, with neither aspect totally dominating the other. Computer science is probably too young and unexplored a field for researchers to form a strong dichotomy between theory and experimentation.
References


*Computing and communications – A perspective of the evolving environment.*

*Studies in the Psychology of Computer Text-Editing Systems.*

*The Keystroke-Level Model for User Performance Time with Interactive Systems.*

*Models of Optimal File Allocation in a Distributed Data Base: A Survey.*


[Fraser 80] C. Fraser.
*A Generalized Text Editor.*

[Frazer 79] W. D. Frazer.
Potential technology implications for computers and telecommunications in the 1980's.


*Minimal String Difference Encodings.*
*Journal of Algorithms*, Accepted for publication in 1982.

[Heckel 78] P. Heckel.
*A Technique for Isolating Differences Between Files.*
*CACM* 21(4), April, 1978.
[Hirschberg 75] D. S. Hirschberg.
A Linear Space Algorithm for Computing Maximal Common
Subsequences. 

[Hirschberg 77] D. S. Hirschberg.
Algorithms for the Longest Common Subsequence Problem.

A Fast Algorithm for Computing Longest Common Subsequences.

A Case Study in Distributed Processing.
In Proceedings of CompCon Fall 1978, pages 399–402. IEEE

A User's Guide to TEC/DATAMEDIA TV-EDIT, IMSSS A20-0110
Stanford University, 1980.

[Kucera 67] H. Kucera and W. N. Francis.
Computational Analysis of Present-Day American English.

Virtual terminal management in a multiple process environment.
In Proceedings of the Seventh Symposium on Operating

An Extension of the String-to-String Correction Problem.

Thor - a display based timesharing system.
In AFIPS Conference Proceedings, pages 623–633. AFIPS.
Spring, 1967.

[NIH 80] NIH DCRT CCB.
Wyliibur Fundamentals

TVEDIT Manual
Stanford University, IMSSS, 1970.

[Roberts 79] T. Roberts.
Evaluation of Computer Text Editors.

[Segall 76] A. Segall.
Dynamic File Assignment in a Computer Network.
[Shaw 74] A. C. Shaw.
The Logical Design of Operating Systems.
Prentice-Hall, Englewood Cliffs, N.J., 1974, pages 141-144
chapter 5.

EMACS Manual for twenex users, AI Memo 554
MIT, 1980.

EMACS, the extensible, customizable self-documenting display
editor.
In Proceedings of the ACM SIGPLAN SIGOA Symposium on
Text Manipulation, pages 147-156. ACM, Portland, Oregon,
June, 1981.

The Cornell Program Synthesizer: A Syntax-Directed
Programming Environment.
CACM 24(9):563-573, September, 1981.

The String-to-String Correction Problem.

Son of Stopgap, SAILON-50.3
Stanford Artificial Intelligence Laboratory, 1970.

New Concept-100/104 Software.
1981.
ARPANET Message sent to INFO-TERMS at MIT-MC.
Robert N. Goldberg

1953
Born February 16 in Long Branch, New Jersey.

1971
Graduated from Northwood High School, Silver Spring, Maryland.

1971-75
Attended the University of Maryland, College Park, Maryland; Majored in Physics;

1975
B.S. with High Honors and Physics Honors, University of Maryland

1975-76
Graduate work in Physics and Computer Science at the University of Pennsylvania, Philadelphia.

1976-81
Graduate work in Computer Science, Rutgers University, New Brunswick, New Jersey.

1976-1979
Research Assistant, Department of Computer Science.

1978
M.S. in Computer Science.

1976-81
Systems work with Laboratory for Computer Science Research, Rutgers University.

1979-81
Special University Fellowship, Rutgers University.

1981
Article entitled "Minimal String Difference Encodings" accepted for publication in the Journal of Algorithms.

1981
Ph.D. in Computer Science.

1982