An Automatic News Article Editor

by
Robert G. Cote

Submitted to the Department of Electrical Engineering and Computer Science in Partial Fulfillment of the Requirements for the Degree of

Bachelor of Science in Computer Science and Engineering

at the

Massachusetts Institute of Technology
May, 1987

Signature redacted

Signature of Author .....

Department of Electrical Engineering and Computer Science
May 15, 1987

Signature redacted

Certified by .....

David K. Gifford
Thesis Supervisor

Signature redacted

Accepted by .....

Leonard Gould
Chairman, Department Committee

An Automatic News Article Editor

by

Robert G. Cote

Submitted to the
Department of Electrical Engineering and Computer Science
on May 16, 1987 in partial fulfillment of the requirements
for the Degree of Bachelor of Science in Computer Science and Engineering.

Abstract

The management of data from a common news wire can easily be automated. The design and implementation of an automatic news wire editor is presented, with emphasis placed on the design goals and how well they were achieved. The operational experience with the resulting system is also presented with respect to both speed an reliability. Lastly, final conclusions are are stated, including discussion of areas of further work.

Thesis Supervisor: David K. Gifford
Title: Associate Professor of
Computer Science and Engineering

Acknowledgements

I would like to thank Prof. David Gifford for all of his support and encouragement throughout this project and over the past two years. I have learned just as much while working in the Programming Systems Research Group as I have during my four years of school.

I would also like to thank Jim O'Toole and Nathan Glasser for answering my annoyingly steady stream of questions about C and Unix.

I would especially like to thank my parents who always believed in me more than I believed in myself. Thanks for all the encouragement and love.

Table of Contents

Table of Figures

Table of Tables

Chapter One

Introduction

The purpose of this project is to design and implement an automatic news article editor in an attempt to automate some of the more mechanical tasks performed by a human news editor. The automatic system will receive news articles from a standard news wire and process the articles as they are received, in much the same way a human news editor does.

1.1 Problem Description

A human news editor performs a number of important tasks during the preparation of news articles. One of these tasks is arranging the lay out of an article to make it more readable. This is particularly necessary for articles obtained from a news wire because some of the more widely used news wires do not send their news information pre-formatted. There are a number of cryptic abbreviations and control codes imbedded in the news wire articles which are meant to direct experienced editors as to how a story is to be handled. A human editor may have to interpret these character sequences and act on them as he sees fit, such as expanding an abbreviation into readable text or, based on a cryptic category designation, direct the article to a different editor who handles that particular category of news. Some of these special characters are also meant to direct electronic devices which preprocess the information before the editor sees it.

Another task which a human news editor performs is to reassemble articles which a news service may have transmitted as more than one piece. If an article is long enough, a news service may divide it into several pieces which are transmitted separately. Also, if an article is unfinished or if a fast breaking story should arise, then an article may not be completed quickly enough and those pieces of the article which are ready for transmission

may be sent over the news wire, with the remaining pieces being transmitted as they are completed. Additionally, as stories develop, optional additions to the previously transmitted story may also be transmitted. A human editor needs to manage the fragments of articles transmitted in such a way, deciding whether or not to hold a story pending the reception of additional text.

Since news stories are constantly changing as more is learned about a given situation, it is common for a news service to send multiple versions of a story over its news wire, with newer versions meant to replace older versions of the same story. So, an additional function of a news editor is to manage versions of stories, being careful to print the latest news available to him.

Lastly, a human editor must filter out various data items which are sent over a news wire and are of little interest to his readers. His job is to print items of greatest interest to his reading audience Some news wires transmit an immense amount of information, far more than any one newspaper can print. A news editor needs to filter through this large amount of data, gathering from it the articles he thinks will interest his readers the most. Some types of information are obviously not intended for readers of newspapers, such as advisories to editors about material being transmitted. A news editor should also filter out this type of material.

It is the goal of this project to construct a system which can perform most, if not all, of these functions. An automatic news editor should be able to properly transform raw news data received from a news wire into a format which is easily readable and aesthetically pleasing. Such a system should also be able to manage fragmented articles, piecing them together as needed. It is also necessary for an automatic system to manage story updates, notifying its readers of the changes as they occur. Lastly, it is also useful for an automatic system to be able to filter a news wire so that various types of information are sent to people best suited to receive them. For instance, advisories from the news wire service should be sent to an editor or system administrator, while actual stories should be sent to readers serviced by the automatic news editing system.

1.2 News Wire Specification

The immediate goal of this project was to produce a program to edit two particular news wire services: the *New York Times* news wire and the *Associated Press* news wire. Both of these news wires follow standardized transmission guidelines published by the American Newspaper Publishers Association Research Institute. The transmission guidelines were first published on June 1, 1976 in the ANPA R.I. Bulletin 1228. The guidelines were revised and published again on February 1, 1979 in ANPA R.I. Bulletin 1312: High-speed wire service transmission guidelines [1].

The guidelines specify that the articles should include, among other things, a priority, a category, a keyword and a date. In addition to these fields, the article text also usually contains an author and a title. The articles that the automatic editing system produces will have a separate field for each of these items, in addition to a field specifying which news wire the article originated from.

1.3 System Integration

The editing system that resulted from this project is actually just one stage in a much larger information system, known as the Community Information Systems. Figure 1-1 is a diagram of the various information system components which comprise the Community Information Systems.

- One of the systems is known as the Boston CommInS project [4]. This system is an experiment in distributing large amounts of information to a metropolitan population in a cost effective way. Basically, the news articles are transmitted over an FM sub-carrier and received by suitably equipped IBM PC's which filter the incoming articles according to the user's interests.

- A second system is the Walter program. Walter allows its user to interactively query the database of news articles for articles of interest to the user. The Walter program can reside on any host with access to the Arpanet.[1] When Walter is run, it makes a network connection to the news article database and sends user commands to the database where they are processed.

[1] Currently, Walter is only available for DEC VAXes, IBM PC-RT's and Sun Microsystems workstations.

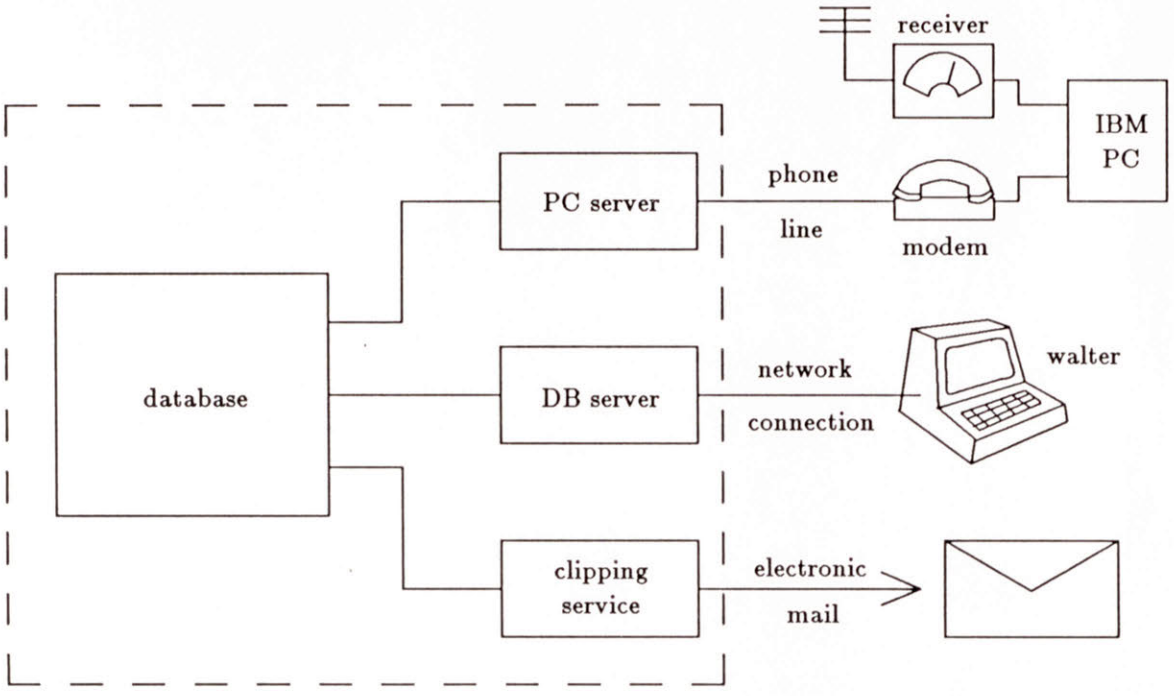

Figure 1-1: The Community Information System

Articles resulting from a query are returned to Walter via the network connection.

- A third system is the Clipping Service. The Clipping Service stores user interest profiles and uses electronic mail to send each user any incoming articles which match his or her interest profile.

Figure 1-2 is a diagram of all the stages involved in producing and maintaining a database of news articles within the Community Information System. The articles are first received by a wire input program which monitors a news wire service line. Each time an article arrives, the wire input program places it into a file in an input directory. The automatic editing system takes articles from the input directory and processes them,

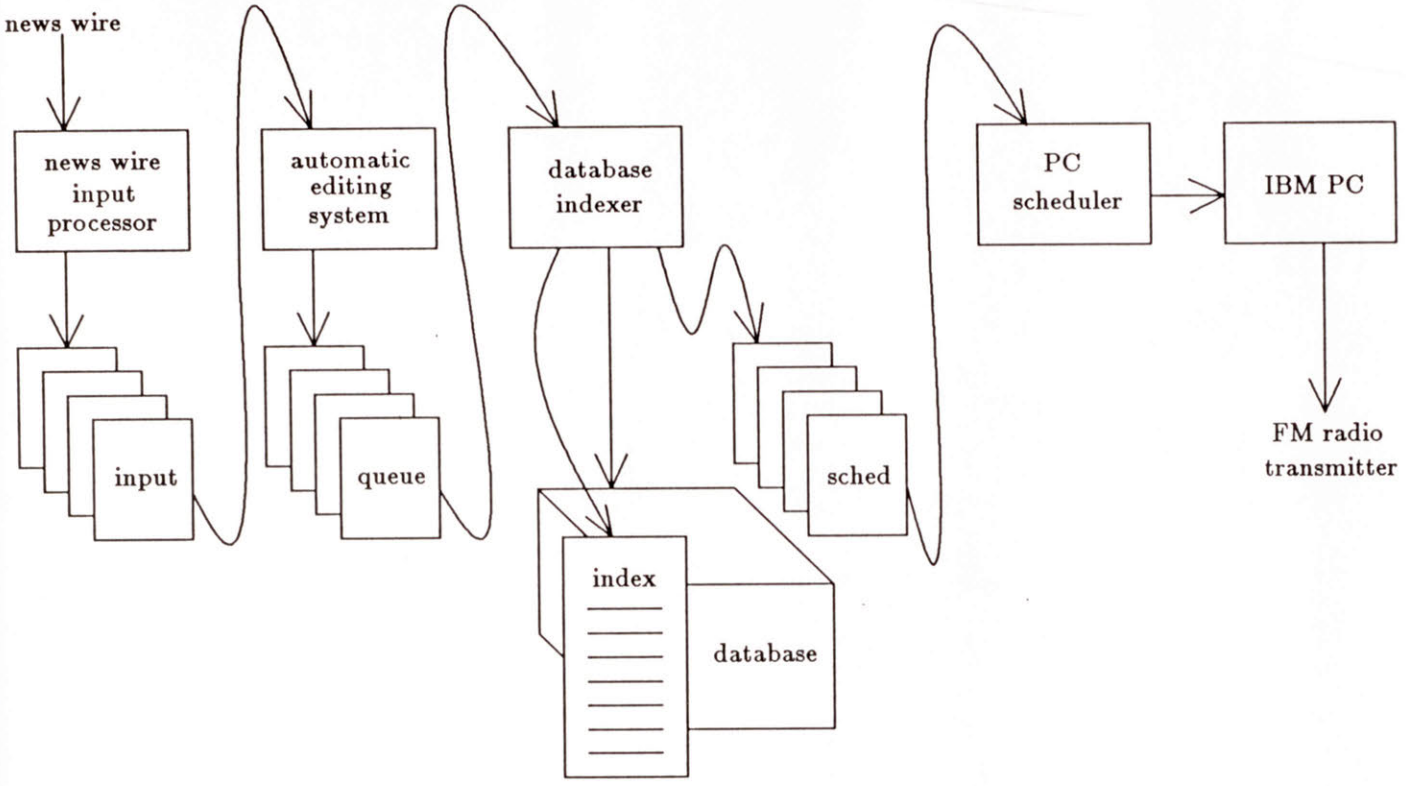

Figure 1-2:A Community Information System Database

placing the finished articles in the queue directory. The indexer takes articles from the queue directory and places them into the database. Depending on what type of news wire the indexer is processing, it may also create a word index for that article, thereby allowing efficient processing of database queries. The indexer also creates a scheduling file and places it into a scheduler directory. The PC scheduler handles sending articles to an IBM PC which then sends them, via a modem, to the radio station which broadcasts them to the Boston CommInS users.

Because the editing system is an integral part of a much used system, its reliability is of utmost importance. If the editing system breaks down, the entire Community Information System will not receive new information. For this reason, I

regarded robustness of the system as the primary goal, with efficiency of space and time as secondary goals.

The article's received from the news wire are processed by performing a number of different functions. The database is designed so that these functions are performed as physically different processes within the same machine. This is an important feature with respect to reliability. If one stage of the article processing should fail, the input to that stage is buffered in a directory. This means that if the editing system should happen to exit abnormally, its input is saved in the input directory and no data is lost. Also, the editing system does not need to be concerned about whether the next stage in the process is currently processing the artciles that the editing system is placing in its own output directory. If the indexer, which uses the output articles of the editing sytem for its input, is not running, the articles will remain in that directory until it is running again. Therefore, we will not *lose* data if the editing system exits abnormally, but the data will be delayed.

The remainder of this paper discusses the problems involved in building such an editing system, some solutions for those problems and how those solutions were applied to a real system. Chapter 2 outlines the editing functions the system needs to perform and the algorithms proposed for doing them. Chapter 3 details the design and implementation of the system. Chapter 4 discusses the operational experience with the resulting system, including its reliability and performance in carrying out the previously mentioned editing tasks. Chapter 5 presents a summary of the results of this project as well as some suggestions for further work.

The automatic editor developed through this project has been in production for 24 hours per day, for 3 months. The system described in this document incorporates much of the practical experienced gained during this time, as discussed in Chapter 4.

Chapter Two

Editing Algorithms

This chapter discusses the algorithms used to perform the editing functions of article reformatting, article reassembling, version updating and article filtering. In deriving these algorithms, emphasis was placed first on robustness, second on efficiency of space and third on efficiency of computation. Robustness is most important because the editing system is intended to be an integral part of an existing information system where reliability is very important.

2.1 Reformatting of Raw Articles

There are three stages to reformatting the raw article. The first stage is the parsing of the header. Most of the header is abbreviated, cryptic information that needs to be parsed and expanded. The second stage is recognizing special fields in the text header. The news wire specifications do not describe where items such as the author and the title should be placed in an article. The *New York Times* and *Associated Press* news wires choose to place these at the beginning of the article text. These special fields need to be recognized and extracted. The third stage is the reformatting of the text. The actual text of the article contains some typesetting characters which need to be recognized and either discarded or used to enhance the appearance of the article.

2.1.1 Parsing the Article Header

The first stage of reformatting the raw data is to parse and format the header of the article. The header contains descriptive information about the article, such as the keyword, the priority, the date and the category. Some of this information is in a cryptic or abbreviated form, so that during this stage of reformatting the information will be

parsed and expanded to make it more readable. Figure 2-1 shows a typical article header.

```
A1016^_tab-z
r i^S^Q BC-ROMANIA      05-10 0460
^B^BC-ROMANIA<
ROMAINS WANT TO REMAIN 'MOST FAVORED'<
By IRVIN MOLOTSKY=
c.1987 N.Y. Times News Service=
@
       WASHINGTON _ In recent days, the Romanian Embassy has brought
to Washington a chorus of 40 Jewish boys and girls who sang Yiddish
songs and a delegation of 12 religious leaders of various
denominations. These were not so much a part of a cultural exchange
or to discuss comparative religion as a question of trade policy.
```

Figure 2-1: Unformatted Article Header

As an example of abbreviations, the category and priority are encoded as single characters and need to be expanded into full words or phrases. Figure 2-2 shows the header after reformatting has been performed, with the category and the priority fully expanded.

The keyword of the article usually needs to be modified before it can be used as a subject field in the final article. According to the specifications, spaces are not allowed in the transmitted keyword so dashes are used to separate words, as can be seen in figure 2-1. The transmitted keyword usually has some version information appended to it as well.[2] So, some of the work needed for preparing keywords is to remove dashes, replacing them with spaces, and to strip the keyword of any version information to make it more readable.

---

[2]This is actually not true. The specifications claim that the keyword and the version information are two separate items within the header of the article. In practice, they are difficult to distinguish. I have chosen to treat the entire portion of the header as a keyword during the header parsing and strip off the known version words afterwards.

```
type: NYT (Copyright 1987 The New York Times)
priority: Regular
date: 05-10-87 0912EDT
category: International News
subject: BC ROMANIA
title: ROMAINS WANT TO REMAIN 'MOST FAVORED'
author:  IRVIN MOLOTSKY
text:

      WASHINGTON -- In recent days, the Romanian Embassy has brought
to Washington a chorus of 40 Jewish boys and girls who sang Yiddish
songs and a delegation of 12 religious leaders of various
denominations. These were not so much a part of a cultural exchange
or to discuss comparative religion as a question of trade policy.
```

Figure 2-2: Formatted Article Header

2.1.2 Recognizing Special Fields

Once all of the header information is parsed, the second stage of reformatting is to extract special fields from the beginning of the text of the article. The news wire specification does not specify where the author, title, copyright and various other pieces of information should be placed within the transmitted article or how these items should appear. In fact, the specifications do not even require that these items exist at all. There are no rules for placement of the these items within the article. This makes the job of reformatting the text difficult since we have to be able to distinguish between a line of article text and, say, the title of the story.

There are six pieces of information, or fields, which may be found in the text, other than the text itself. They are:

- the copyright notice
- the author
- the title
- parenthesized notations

- article glue
- notes to editors

There are a few pieces of information we know that will help us properly recognize these fields. The formal specification of the article text is all of the characters between the special `text_begin` and `text_end` characters, then we at least know that those fields, if they exist at all, will be found between those two characters. We also know, by observation, that all of these fields are found after the special `text_begin` character and before the actual text begins. In figure 2-1, the '^B' at the beginning of the third line is the `text_begin` character. Notice that after this character, we find a copy of the keyword, a title, an author and a copyright. Following this information is the actual text.

The first two fields listed above are the easiest to find. A copyright notice is a line which begins with the characters 'c.' or the string 'copyright'. An author is a line which begins with the string 'By'. It is possible that a line of text might begin with either of the strings 'By' or 'copyright' but not be an author or a copyright notice. However, for this to happen, that line would have to be the first line of actual text, because once we recognize the first line of actual text, we no longer search for any of the special fields listed above. The first line of actual text usually begins with a tab followed by the geographic location where the article was written, such as 'WASHINGTON' on the first line of text in figure 2-1. It is highly unlikely, then, that the first line of text line will begin with either of those strings.

The title is probably the most difficult of these fields to recognize. There are no characters at the beginning of the title which distinguish it from any other line of text and, to compound the problem, a title looks very much like a normal line of text. Additionally, not all articles have a title. If articles were required to have a title, then it would be much easier to distinguish between the first line of text and a title. Suppose this requirement did exist and there was a line which could be either a title or the first line of text. If we have not yet found a title, then obviously the line is a title. If we have found a title then the line is the first line of text. However, as already mentioned, articles are not required to have a title. A title *does* have a trailing quad character, used

by typesetters to position the title on a line. The first line of text is not known to have such a trailing quad character. The criterion, then, that we will use for recognizing a title is that a line of text is a title if it ends with a quad character and does not meet the criterion for the other five fields listed above.

Parenthesized notations can often be found at the beginning of the text, sometimes specifying which section of the newspaper the article should appear in or possibly some other note meant for an editor. We can easily notice these lines since they begin with an open parenthesis. Also, a parenthesized notation, like a title, usually ends with a quad character. Sometimes a parenthesized notation extends past one line, into two or three lines, so we need to be careful to capture all of it.

When an article is divided in pieces and transmitted as a lead with corresponding adds, the transmitting news service will sometimes include in the header of the text of the add some information which helps in reassembling the full article. I refer to this information as 'article glue' since it helps to 'glue' fragmented articles back together. This information is usually the first two and last two words of the article piece which precedes the current piece in the total sequence of the article's pieces. This information can be used to make sure that the end of one piece and the beginning of another piece are properly 'glued' together. This information is easy to distinguish and can be purged from the text.

Notes to editors can also be found at the beginning of the text. These notes usually point out such things as corrections or scheduling information for articles. We can recognize these lines because they start with either of the two strings 'Eds.' or 'Editors'.

2.1.3 Reformatting the Actual Text

Most of the actual article text is in an acceptable form when it is received, except for a few typesetting characters that the originating news source inserts into the text to direct editors or machines receiving the text. There are four types of typesetting characters that are commonly found in the transmitted articles: quad characters, rail

characters, tabbing characters and em dashes. The last of these, an em dash, simply translates into a pair of adjacent hyphens.

Quad characters indicate that portions of text should be centered, flushed right or flushed left on a line. For purposes of this project, quad characters are ignored. Quad characters are usually used within a particular line to position portions of text to enhance their readability, such as centering the title. Most of the time, the portions of text which are positioned by quad characters are portions of text that we would like to treat specially. For instance, in figure 2-1, the title is followed by a quad character, but in figure 2-2 the title is a separate field. Since we want to format these kinds of fields ourselves, the quad characters serve no useful purpose. Sometimes quad characters appear within the *text* of the article and can actually be used to perform some useful formatting. However, I have chosen not to use the information provided by these characters, because I don't believe enough would be added to the presentation of the text, which is usually displayed on a standard computer screen, to justify the overhead involved in processing these characters.

Rail characters indicate where a bar, or rail, is suggested to divide sections of text. This is commonly used in newspapers to separate titles and subtitles from one another. Figure 2-2 shows how rails appear in the input article. In the figure, '^' is an upper rail and '@' is a lower rail. Figure 2-3 shows how the rails appear in the output article. Since the resulting text of this system will not be displayed on a bitmapped computer screen, we do not have enough resolution to distinguish between an upper rail and a lower rail (i.e. a rail meant to appear directly above a line of text and a rail meant to appear directly below a line of text). Therefore, we can easily display either type of rail as simply a line of dashes.

One difficulty encountered in processing rails is that they are sometimes used to designate italicized or emphasized words within text. We will need to distinguish between this use of rails and rails used to set off a portion of text. We can do this by only using rails which are outside of a paragraph of text. Through much observation, the most common and useful function of rails is to 'frame' a heading or title, which usually

```
The New York Times News Summary for Monday, May 11, 1987:

^The World<
@
        PARIS _ The trial of Klaus Barbie in France is due to begin.
The trial, taking place more than 40 years after his alleged crimes
as a Gestapo lieutenant, has absorbed the French like few other
recent events .... BARBIE.

^The Nation<
@
        WASHINGTON _ Increasing voluntary AIDS testing has been
recommended by federal health officials, in a confidential new
report. But they opposed mandatory testing and called for laws to
protect the secrecy of test results ... AIDS.
```

Figure 2-3: Unformatted Text Containing Rail Characters

```
The New York Times News Summary for Monday, May 11, 1987:

------------------------------------------------------------------------
The World
------------------------------------------------------------------------

        PARIS -- The trial of Klaus Barbie in France is due to begin.
The trial, taking place more than 40 years after his alleged crimes
as a Gestapo lieutenant, has absorbed the French like few other
recent events .... BARBIE.

------------------------------------------------------------------------
The Nation
------------------------------------------------------------------------

        WASHINGTON -- Increasing voluntary AIDS testing has been
recommended by federal health officials, in a confidential new
report. But they opposed mandatory testing and called for laws to
protect the secrecy of test results ... AIDS.
```

Figure 2-4: Formatted Text Containing Rails

each appear on their own line in the input article, not within a paragraph of text. Therefore, only using rails which appear outside of paragraphs will intentionally exclude rails for italicized or emphasized words while retaining rails for framing headings.

Tabbing characters are used to direct the formatting of tabular or columnar data, such as stock reports or sports box scores. When a news wire service intends for a line to be typeset into a number of columns, it will send a special tab line indicator character at the beginning of the line. This character indicates that other formatting characters will appear within the line, directing how much spacing should appear between columns so that columns will line up properly. However, the tabbing information that is sent is intended for typesetting done by newspapers, where various spacing characters are used to compensate for the fact that all characters in newsprint are not the same width. For a computer display, where all characters *are* of equal width, this information is unnecessary. Therefore, once we notice a tab line indicator, we will need to calculate the tab spacing ourselves based on the size of the text comprising each column and the total allowable length of a line of text. The one problem we need to be aware of is that the size of a column may change within a table. For instance, the figures appearing in the beginning entries of the column may all be less than one hundred, i.e. at most two digits wide, while figures appearing later in the column may be greater than one hundred, i.e. three or more digits wide. This causes a problem because, if we have calculated each column to be two characters wide and adjusted the spacing accordingly, then the amount of spacing we are using will not properly line up columns which are three characters wide. We can usually overcome this problem by building a certain amount of tolerance into a column width, say 3 characters, to account for a reasonable range of figures.

2.2 Reassembly of Articles

It is common for a news wire service to divide a long or unfinished article into pieces and transmit the pieces separately. There are three types of article pieces, other than whole articles, which are commonly transmitted:

1. A lead which is the very first piece of an article, usually containing enough information to predict how many other pieces will be transmitted and what type of pieces they will be.

2. An addition, or simply an add, which is a piece meant to be appended to either an article lead or another article addition.

3. An append, otherwise known as an add-at-end, which is also meant to be appended to an article lead or an article addition, but many times as an 'afterthought' or as an optional add.

An append itself can sometimes be transmitted as a number of separate pieces, divided into a lead and a number of adds. However, it is useful to think of the entire append as one type of article piece.

When an article is received, information which designates what type of an article piece it is can usually be found in the header when it is parsed. However, the header doesn't always contain the information it should about what type of article piece we are dealing with. In some cases, we can determine from the text itself that more pieces of the article are to be expected. Usually, either the string '(MORE)' or the string 'nn' can be found at the end of the text if an add or an append is expected for this piece. So, we can look for either of these strings at the end of the text, in addition to information gleaned from the header, to help us determine if more article pieces will arrive when that information is lacking.

We can act upon the article piece based on what type of piece we believe it to be. If the article is a lead, we will store it in a temporary place, pending reception of all of its additions and appends. If the article is an addition or an append, we will try to find its lead and store the addition with the lead. If this addition or append completes the article, we will output the entire article.

If we can find no information in the article to designate that it is a lead, an addition or an append, then we will assume it is a whole article, ready to be output as complete. However, a small percentage of the time, an append will arrive for an article that was assumed to be whole. This is a problem because the article is no longer in temporary storage and therefore cannot be used to reassemble the append with its lead, which we assumed was a whole article. We can solve this problem by using the 'replaces' field of an article. The replaces field is mainly used in version handling. It specifies that the newly received article containing the replaces field should replace any articles whose identification numbers are listed in the replaces field. We can still output an article we suspect is complete, but we will save a copy of it in temporary storage in case an append

does arrive for it. If the append should arrive, then we can assemble the fully received article, adding a replaces field to indicate that this newly completed article should replace the article which we falsely believed was the completed article.

The wire services are not always reliable, and therefore a lead may wait forever for its additions which, because of unreliable wire transmission, never arrive. To mitigate this problem, we will apply a timeout constraint on article pieces in temporary storage. If an article has been waiting long enough, say thirty minutes, then we will assume that its additions and appends are not going to arrive. We will then assemble the article to the best of our ability, using whatever pieces we have, and ouput it as complete. The justification for this decision is that a partial article is still worth outputting since article pieces are large enough pieces of text that the article contains useful information. Additionally, articles are often divided so that each piece contains a complete thought or complete facet of the total story. So an article composed of only a few of its original pieces still contains a number of complete ideas. The timeout value should be based on observed behavior of the news service. A value on the order of thirty minutes seems sufficient for the *New York Times*.

In some cases, an addition or an append may arrive before their corresponding leads. Obviously, we will not be able to find the lead in temporary storage because it has not yet arrived. We will need to store these 'orphaned' additions in a separate storage place, referred to as the orphanage. If an article lead which is waiting for its additions has timed out and we are attempting to reassemble the few pieces we have, we may search the orphanage for one of its additions which may have arrived before the lead.

2.3 Version Updating and Duplicate Detection

It is not uncommon for a news wire service to send a new version of a previously transmitted story. Sometimes a story needs to be rewritten because the facts involved have changed or it has been decided by the author (or his editor) that the story needs restructuring. When a story is an intentional repeat or rewrite of a previously transmitted story, it is usually indicated as such in the header of the transmitted article.

We want our system to be able to recognize this information and indicate in the latest article that the former story should be replaced by the latest one.

In order to perform the task of version updating and replacing, we will need to remember some amount of information about previously received articles. If an article is received and it claims that it is a rewrite or correction of a previously received article, then we can search a table of previously received articles to first determine if we have received the original version. If we have, we can determine the original's unique identification number and include a replaces field in the new version, stating that it replaces the article whose unique identification number we looked up. If we have not received a previous version, then we will treat the current one as the original version.

We can also use the information in this table to help trap duplicate articles. A news service may send out an article that is identical to a previously transmitted article. Sometimes this is done for the benefit of a customer who did not properly receive the original, or sometimes the article is repeated if the news service suspected that it wasn't sent properly.

What type of information do we want to store about previously received articles? Clearly, we do not want to save complete articles since the work involved in searching through all previously received articles for one particular article is substantial. We want to be able to determine two facts for each incoming article: whether a previous version of the article was transmitted and what the article identification number of that previous version is. This information will be used to help eliminate duplicate articles and replace old versions of articles with newer versions. We want to save enough information to uniquely identify every article we have previously received while using a minimum amount of space.

The wire services guarantee that the keyword of each article uniquely identifies it. In actuality, two different articles may have the same keyword. For instance, one type of item transmitted is a budget of articles, which is a list of articles that the news service expects to transmit. Most budget articles have the same keyword, e.g. NYT-BUDGET,

so that we need more information than just the keyword to differentiate between various budget articles.

Other candidates for stored information are the date, the priority, the title or the author of the article, but none of these serve to uniquely identify an article. It is possible for a revised article to be transmitted the day after the original or even at a different priority, depending on how urgent the story has become. Not all articles arrive with a title and an author so that neither of these items may be used to reliably identify articles.

The category is an additional piece of information necessary for uniquely identifying each article. Users of the Community Information Systems should be able to retrieve articles by specifying that the articles should have a particular category designation. Suppose an article is transmitted with the category 'Entertainment and Culture' and the same article is later transmitted with category 'Sports'. Even though the text of the article is the same, we would like to treat these two articles as different articles, because we would like to provide this article to users who request articles with the category 'Entertainment and Culture' and also to users who request articles with the category 'Sports'. We don't want to eliminate the second article as a duplicate just because the text is the same. Therefore, the category of the article will be part of the information used to uniquely distinguish it.

In addition to the keyword and category, we might consider storing the text of the article. This is clearly more information than we want to save, but we can use the text to generate a string to uniquely identify each article. We can create an 'article stamp' from the text by taking the first character from the first 5 lines of the text of the article. A small amount of testing was performed to verify that in fact these article stamps do uniquely identify each article.

We can safely assume that no two articles will arrive with the same keyword, category and article stamp. For purposes of version updating, we actually do not need to store the article stamp. It is actually a likely occurrence that two different versions of an article will have different article stamps. Article stamps are used for duplicate detection.

2.4 Article Filtering

We would like to be able to filter out articles of little or no interest from the steady stream of articles we will make available to our users. A portion of the news wire is meant to be read just by editors, such as budgets for the days articles or advisories about what artwork or photographs are available for particular articles.

Because we want the system to be efficient in both space and time, we will avoid trying to perform reasoning on the content of the text to determine whether an article should be included in the final database. There are a number of much easier checks we can make to determine which articles to save and which to discard.

One check, and probably the most useful, is to use the article's keyword. Artwork advisories, for instance, have a keyword like ADVISORY-PICTURES. This type of article, among others, is of little interest to general readers, especially since we do not receive and distribute to our readers the artwork mentioned in these advisories. A reasonable strategy would be to store a table of keywords we want to act on in a special way. In this table, we can store all keywords for articles that we know ahead of time we do not want to save. We would like this table to be dynamically configurable. In other words, we would like to change which keywords are in the table without having to recompile the program. This would probably mean storing the table in a file, so that we can edit the file to change the table and cause the program to re-read the file while it is executing.

Chapter Three

System Design and Implementation

This chapter will discuss the design and implementation of the system. There are five design goals which helped shape the design:

- functional modularity, i.e. modularization of portions of code which serve a similar purpose
- data abstraction using data object modules with clearly defined interface boundaries
- fault tolerance or robustness in the face of poorly formatted input or a program/computer crash
- code generality so that portions of code can be re-used to implement parsers of other data types
- dynamic configuration of system parameters so that the program can easily adapt to subtle changes in the input articles

These concepts and their effects on the design of the system will be presented in the following sections.

3.1 Modularity

This section discusses how the program is modularized, what functions each module performs and how the modules interact. The primary motivation for modularizing the program is to 'divide and conquer'. It proved useful during the implementation phase of the project to implement and test modules separately.

Modularization also helped support some of the other design goals. For instance, meeting the design goal of code generality was greatly aided by enforcing functional modularity. General purpose source code is created so that it can be easily shared

between programs which perform similar tasks. If this code is separated in functional modules as well, i.e. each module performs a distinct, useful function, then a programmer can simply 'plug' an existing module into his program. Likewise, modularization supports data abstraction by helping to clearly separate and group those functions which belong to a particular data abstraction.

Figure 3-1 is the module dependency diagram for the program, showing the major and minor modules in the program and how they depend upon each other.

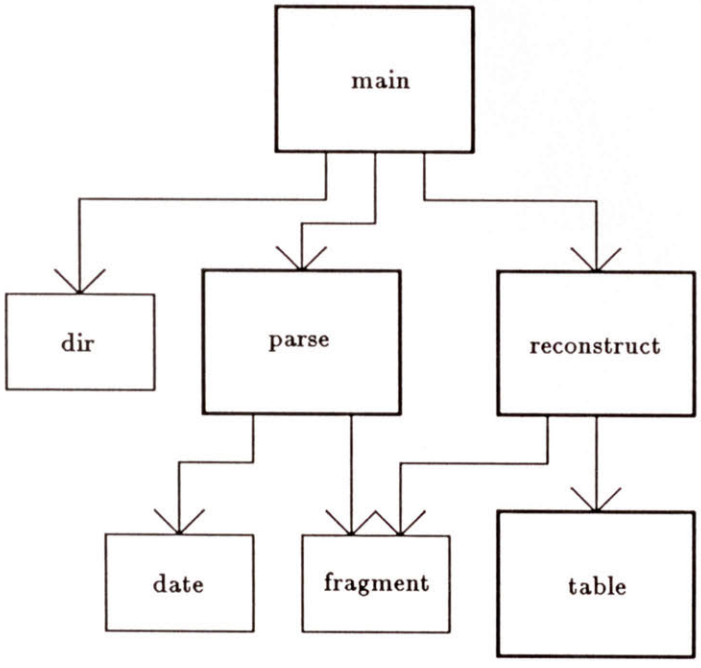

Figure 3-1: Module Dependency Diagram

The four major modules are the main module, the parse module, the reconstruct module and the table module.

3.1.1 The Main Module

The main module is the driver loop of the program. Its function is to open the input directory and pass the files in the input directory to the other modules for processing. After the article is processed, the main module will pass the next article in the input directory to the appropriate modules for processing. The main module will continue processing articles in the input directory in the same fashion until the directory is empty. After it empties the directory, it sleeps for a period of time and begins the loop over again, processing input articles if they exist.

3.1.2 The Parse Module

The parse module looks for pre-defined fields within the article, such as the author, the title, the category, the date and, of course, the text. This particular module was written to parse input articles which conform to the high-speed wire service transmission guidelines published by the American Newspaper Publishers Association.

The parse module creates a parse object containing all the information it was able to gain from the input article. If the parse module was unable to parse the article *at all* (i.e. the input article was incorrectly formatted) then it returns a null-valued object and the main module saves the unparseable article in a bad articles directory for perusal by a system administrator. An article may be unintelligible because of line noise which caused the article to be garbled or because the originating source did not properly follow the guidelines for the wire service.

The parse module was implemented with one main function as its external interface. This function is `parse_article` which takes a character buffer containing the raw article as input and returns a parse object containing the parsed article. Parsing is done in a top down manner. There are a handful of other functions which are part of the interface to the parse module. They are mainly used for creating and writing to a file a readable, text representation of the parse object. These routines are used to create the output article file from the parse object.

The internal representation of a parse object is a large structure with a separate

field for each piece of information we would like to find in the input article. Most of these fields are just strings that will be filled in as we parse the article. A few of the fields are pointers to other structures, such as a data object or a fragment object. The parse object also has a pointer to the character buffer containing the input article text. Therefore, we don't need to explicitly make the input article buffer available to procedures. Passing the parse object is sufficient.

3.1.3 The Reconstruct Module

If parsing succeeds, then the main module passes the resulting parse object to the reconstruct module. It is the job of the reconstruct module to reassemble articles which arrive in pieces.

It is common for a news service to send articles in pieces if the article is long or incomplete. Most of the time it is possible to anticipate how many total pieces are expected to comprise a total article after having received just the lead piece. This information is usually contained within the header of the lead article. The reconstruct module takes this information from the parse object it was passed and decides one of three things: if the article should be held pending future additions, if the article is an addition to a previously received add or if the article should be output as complete. If it decides that the article is complete, it places it in the output directory.

The reconstruct module has a simple external interface: the procedure `decide_reconstruct`. This procedure takes as arguments a parse object and a reconstruct object. It decides, based on information found within the parse object, how to handle the current input article. One field within the reconstruct object is a table object where all the information about what articles have arrived is stored. The `decide_reconstruct` routine interacts with the table module. It provides information about the current article and responds to results from the table module, such as outputting the articles that the table claims are complete.

3.1.4 The Table Module

If the reconstruct module decides that the article is not complete, it passes the article fragment to the table module. The table module stores article fragments along with how many pieces are expected and how many pieces have arrived for each article. Since all article fragments are guaranteed to have the same keyword as the keyword of the lead article fragment to which they belong, articles are stored in the table by the keyword of the lead article fragment. As additions to articles are inserted into the table, the table module will notify the reconstruct module whether or not the entire article has arrived. If it has arrived, the table module provides a reassemble routine that the reconstruct module can call. The reassemble routine puts all of the article pieces into one file and returns the resulting filename, which the reconstruct module can then place in the output directory.

If an article addition is given to the table module but an article lead cannot be found for it, then the addition is saved in an orphans directory. If the table module tries to reassemble an article but can't find one of the pieces, it can search for that piece in the orphans directory, referred to as the 'orphanage'. The names of the orphans are stored in a hash table to allow the table module to quickly determine whether or not an orphan exists for a given lead.

The article fragment table is implemented as a circular array of entries. The index pointer into the table points to the head of the table, which always houses the most recently inserted entry. Entries are always inserted at the head of the table, and the head pointer is incremented with each insert. To find a particular entry in the table, we start at the head and compare the keyword of the article at that entry with the keyword of the article we are looking for. We continue in this manner, searching linearly through the table, until we match the keyword or reach the end of the table or an empty entry. Figure 3-1 shows the basic structure of the table object, how entries are inserted and how entries are searched for. The table is optimized to find the most recently inserted entries fastest, since we are simply performing a linear search starting at the head of the table. This turns out to be a good strategy since additions to article leads usually arrive soon after the article lead. We can safely assume that most of the time an article lead for a newly received addition is close to the top of the table.

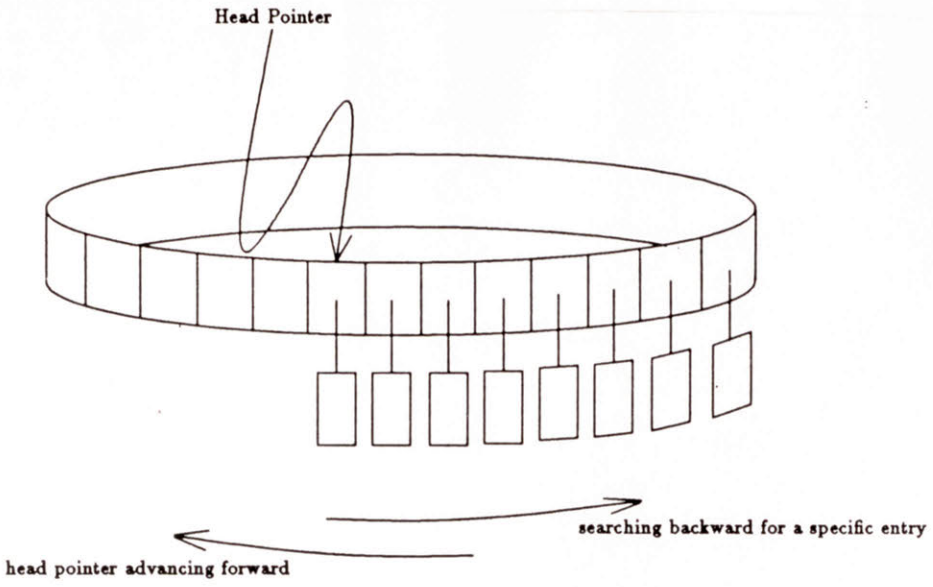

Figure 3-2: The Internal Structure of a Table Object

As discussed in Chapter 2, we do not want the table to retain partially received articles indefinitely, pending the reception of all of its pieces. This would mean that if a piece of an article did not arrive for some reason, then the article would never be placed in the output directory. Therefore, the table stores a time stamp with each entry inserted, designating when the entry was inserted. We can use this time stamp to determine if an entry has been waiting too long and should be reconstructed and placed in the output directory, even though some of the expected pieces are missing.

To insert a new entry into the table, we advance the head pointer forward one location and place the new entry at that location. Since the table is a circular array, the entry we find after advancing the head pointer should be empty if the table is not full. If we advance the head pointer and find an occupied location, then the table is full and we need to create some free space. We can safely assume that all of the articles in the table are not yet complete since articles are flushed from the table and placed in the output directory as soon as they are complete. Since there are not completed entries in the table,

we can't simply clear out a completed entry to make room for a new entry.[3]

The first strategy we will try for creating free space is to look for entries in the table which have been waiting longer than a timeout threshold for all of their pieces to arrive. These entries are referred to as 'oldtimers'. An oldtimer needs to be forceably reassembled and placed in the output directory, even though all the pieces for it have not yet arrived. If there are still no empty entries in the table after we have checked for oldtimers, then we will force out the oldest entry and use the resulting empty space for the new entry. Each time we remove an entry from the table, we need to compact the table so that no holes exist. However, we are inserting entries in order of their arrival. Therefore we do not need to compact the table when either clearing out oldtimers or forcing out the oldest entry since both of those types of entries are guaranteed to be the last entries in the table.

There are a number of functions which form the external interface to the table module. The first of these is `create_table` which allocates space for and returns a new table object. There is also a `free_table` function for freeing memory allocated for a table object. There are three insert functions, `insert_lead`, `insert_add` and `insert_append`, one for each type of article fragment which can be inserted. There is also a `reassemble_entry` routine which can be called if one of the three insert functions returns a value indicating that the article piece just inserted has completed the article it belongs to. If the table is full when we try to insert, then there is a function called `search_for_oldtimers` which searches in the table for oldtimers. If it finds one, it forceably reassembles it and returns the resulting filename. If none are found it returns a null value, so that all oldtimers can be removed from a table by iteratively calling `search_for_oldtimers` until it returns null value. If there were no oldtimers in the table, then `force_out_entry` can be called which will force the oldest entry to be reassembled, making room for a new entry. There is one last function in the external interface called `crash_recovery`. This function can be used to recover the state of the

[3]Articles are also removed from the table when they time out and are forceably reassembled. However, the argument still holds.

table if the program should unexpectedly crash. It takes as an argument the root directory of the table module and returns a table object containing whatever information it could find in that directory.

3.2 Data Abstraction

This section discusses how the ideas of data abstraction affected the design of the program. Various data abstractions which arose from the design are described, as well as what rules were used to enforce data abstraction within the program.

The motivations for using data abstraction are similar to those for using functional modularity when dividing up the program. Both concepts help to cleanly separate out important portions of the program so as to reduce the complexity inherent in implementing and debugging a large program. Once the complexity within a module or data abstraction has been dealt with and resolved it can be forgotten as the rest of the programming task deals only with the clearly defined interface that results.

The effects of data abstraction are closely related to the effects of modularization on the system design, because most of the major modules were implemented as data abstractions. In creating the data abstractions, I tried to follow the rules enforced by the programming language CLU [3]:

- A data abstraction is a separate module.
- The representation used for the new data type is internal to the abstraction and not visible to users of the abstraction.
- Interactions with the module and operations to be performed on data objects created by the module are the external interface to the module. This external interface is well-defined.

Because I used C to implement the system, there was no way to use the programming language to *strictly enforce* this discipline. Instead, I defined a methodology for using data abstractions within the framework of C and adhered to it while programming [2].

3.2.1 Implementing Data Abstractions Using C

Ultimately, we would like a programmer to be able to use a data abstraction we create by using only the specifications of the abstraction and the compiled data abstraction module. Using C, we can easily give the programmer an object code file containing the routines contained in the data abstraction module, but we also need to provide some specifications of the data abstraction. In CLU, this is accomplished by using spec files generated by the compiler. The spec files specify the interface to a module by listing the procedures which constitute the external interface. The compiler can perform any type checking, argument count checking and return value checking by using just the spec file of a module. C allows the programmer to provide module specifications by using header files. In my discipline the header file for each module contains external definitions for all functions which the programmer of the module intended to be part of the external interface. Each external function definition also contains comments about what arguments the function is expecting. By using the header file and the object files of the module, a programmer can be separated from the internal details of the module.

In C, new data types can be formed from existing data types by using a typedef statement. A typedef allows you to take an internal representation and give it an external name in much the same way that CLU takes an internal representation and creates a new kind of object. In CLU, the internal representation of the data abstraction is kept completely hidden from users of the module. However, in C, the actual representation cannot be kept completely hidden from the programmer. The typedef statement must be known at compile time by any module using objects of that type. This implies that it must appear in the module's header file, which the programmer must look at in order to use the abstraction.

A flaw, then, in the discipline is that the representation of the object is exposed and the abstraction can therefore be broken by the programmer. As stated earlier, this discipline cannot be enforced by the programming language. The programmer using the data abstraction must decide whether to adhere to the abstraction as given or to risk manipulating the data object directly.

3.2.2 Other Data Abstractions

As mentioned above, each major module, with the exception of the main module, was implemented as a data abstraction. In addition to having parse, reconstruct and table objects, there were other data types which proved useful. The article fragment information contained within an article, such as whether the article is a lead, an addition or a complete article, is information used by both the parse module and the reconstruct module. It was convenient to define a new data type, called a fragment object, that could be passed between the two modules thereby efficiently getting the necessary information from one place to the other. Other data types which proved useful were a date data type for storing information about an article's arrival date (i.e. day, month, year, hour and minutes) and a directory data type for opening, closing and reading directories.

3.3 Fault Tolerance and Robustness

This section discusses how the program was designed to behave well in the face of unexpected input or unexpected operating system performance. I expended a good deal of effort in minimizing the amount of data that can actually be lost. I regard minimal loss of data as a primary requirement of the editing system. Loss of data can occur in two ways: an unexpected system crash or a poorly formatted input article.

3.3.1 Recovery After A System Crash

The first way data can be lost is for the entire computer system or the article editor program to crash without notice. The danger is that any article currently being processed or any article stored in an internal table might be lost. The first problem is easily solved by leaving an article in the input directory until it is completely processed, which means it has either been successfully placed in the output directory or it has been moved to another safe directory. The second problem is solved by not storing the entries of the internal tables in main memory. Instead, articles stored in an internal table are actually stored in files residing in a directory known to the table. It is the filename of the article file which constitutes an entry in the table stored in in the program's run time memory.

A related problem is that once the program crashes, the table, meaning the collection of filenames in main memory, is lost and must somehow be reconstructed from the files it has saved in the known directory. The technique used to solve this is to use a filenaming discipline which puts enough information in the name of the file so that a crash recovering routine can determine which entry that file was in the table and all of the state which it had that table entry had.

For example, a typical filename generated by the table might be `L_BC-ARMS-TALKS_5`. The leading `L` indicates this particular file is a lead article piece. If this piece were an addition or an append, then the leading character would be a `P` and an `X` respectively. The string `BC-ARMS-TALKS` is the keyword which the originating news service gave the article. The trailing digit `5` indicates that this article is composed of a total of five pieces, including this one. If this piece were an addition or an append, the trailing number would indicate which add number it is within the sequence of adds or which append number it is in the sequence of article appends.

Using these naming conventions, the table can be recovered during start up by scanning the directory, parsing the filenames as they are encountered and inserting them into the table based on the information found within the filename. The alphabetizing ensures that all the leads will be recovered before the adds which will all be recovered before the appends. Also, the trailing numbers ensure that lower numbered adds will be recovered before higher numbered adds, which is the order in which they appear in the completed article.

3.3.2 Recovery From Poorly Formatted Input

For the most part, the articles which are transmitted over the news wire are constructed and formatted by human beings and not by computer. Even though the editing system was designed to parse and process articles which are expected to conform to some published specifications, the system needs to be able to properly process articles which deviate in minor ways from the standard. The system also needs to be able to gracefully recover from articles which deviate in major ways from the specifications.

For portions of the specification, it is easy to allow a tolerance for small deviations. For instance, according to the specifications, the keyword should be a maximum of 24 characters long and contain no spaces. The *New York Times* often does not follow this standard. However, we know from the specification that format identifier characters precede the keyword and that a filing date follows it. So we define the keyword to be all the text text between these two fields. If the filing date should is missing from an article, then we can use the end of the line as an ending delimiter instead. These types of errors are considered minor deviations from the specification since, for the most part, the necessary information can still be extracted from the article.

A major deviation occurs when an important item is left out of the article. For instance, sometimes an entire line of header information is lost because of electrical noise on the news wire. This usually results in an article with half a header field which abruptly ends, with the article resuming in the middle of the article text. Such an article cannot be processed to any useful degree and must be discarded. It is important that the editing system be able to recognize when an article is severely garbled. If it cannot find a required field that it is looking for within the header, then it assumes that the article is garbled and stores it in a directory of unparseable articles.

3.4 Generality of Code

A very important criterion for the design of this program was generality of source code so that certain modules could be easily re-used to construct automatic editors for other news wires. In particular, both the *New York Times* and the *Associated Press* news wires follow the same transmission guidelines, each with their own idiosynchrasies. We would like to be able to easily construct an *Associated Press* editing system by simply modifying small portions of the existing *New York Times* editing system.

In practice, the real differences between news wire services are in the way the transmitted data is arranged. The parse module is the only module which needs to know details about these arrangements. Once the articles are parsed and a standard parse object is produced, the rest of the code should be able to properly handle the parse object

without much modification. In other words, the news wire specific details are only contained in the parse module. The reconstruct and table modules are designed to handle any type of articles which arrive either complete or in any number of pieces. The table is able to store any type of articles, provided that a unique identifier can be supplied with it. The same holds true for the duplicate detection mechanism. The version updating is also general purpose in that it allows an article to be specified as either an original, an addendum or a replacement. An editor created for any news service should be able to use these modules effectively.

An added advantage of not having to rewrite a lot of source code is that debugging becomes less time consuming. Once the first editing system is written, a second editing system can be constructed using much of the source code from the first. Naturally, some of the procedures from the first editor might need to be rewritten for the second. These procedures should be placed in service specific files. This results in two editing systems which share large amounts of code. This means that there is much less code to be tested. We will only have to debug the few service specific files.[4]

An example of how generality is useful is the creation of an Associated Press editing system using the existing *New York Times* system as a base. The type of header information contained in both *New York Times* and *Associated Press* input articles is, for the most part, the same. However, the arrangement of that information in the article varies in ways between the two services, even though they both claim to follow a published standard. As work towards converting the *New York Times* editor into an *Associated Press* editor begins, it seems that the reconstruct and the table modules will be useable as is. Because both of these news services use the same transmission specification, much of the parse module is useable as well. Parsing procedures which need to be different are stored in separate files. The editor for each service will use its own version of the procedures.

---

[4]In practice, it may be the case that certain idiosynchrasies of the new news wire could cause other portions of code to misbehave. Such effects should be very limited and are probably bugs which previous news wires did not aggravate. In other words, this misbehaving is actually beneficial in that it points out bugs that we were not yet aware of.

3.5 Dynamic Configuration

As we observe the data transmitted by the news wire services, we learn more about how to better process it. As we learn more, We would like to be able to modify various parameters of the program in order to further optimize its performance. An example of this type of parameter is how long an article should be retained, pending the arrival of its adds. If after this timeout period, all of its adds have not arrived, then we will assume that they are not going to arrive and force the article to be output. Another example is logging level variables for enabling and disabling various log messages. The program has the ability to log messages describing all of its actions to allow a system maintainer to monitor its run-time behavior. The logging has been arranged in a series of levels, such as common information, uncommon information, common errors and uncommon errors. This allows us to log lots of information during the debugging phase by enabling all of the logging while just logging important errors when the program is running in production by enabling just uncommon error messages.

We would like to be able to vary parameters and configuration variables without recompiling and reinstalling the program. For instance, if we notice the program behaving unexpectedly, we may want to turn on all of the logging for a short period of time to better understand what the program is doing. Or, suppose that we notice articles are not waiting long enough for their additions to arrive. We would like to be able to increase the timeout parameter without having to stop the program, edit the source code, recompile the source code and reinstall the program.

The editing system is designed to dynamically read commands out of a configuration file. At any point during the execution of the program, a signal can be sent to it that will make it re-read the configuration file, setting its parameters as specified in the file and then continue processing where it left off. Now, to change a system parameter, we simply edit the file and send the proper interrupt to the program. This feature is implemented in a very general way so that we can easily add parameters to the list of dynamically changeable parameters. The program does need to be re-compiled to do this. The implementer simply needs to supply the name of the parameter and the handler for changing the parameter.

Chapter Four

Operational Experience

The work for this thesis has resulted in an automatic editing system for the *New York Times* news wire service. This system has been in production operation for approximately three months. This section will discuss the measured performance and the observed reliability of the system.

4.1 System Performance

I performed a series of tests to determine how much CPU time the editing system requires to process each article. I performed these tests on a DEC MicroVAX II/GPX running the Ultrix 1.1 operating system. The MicroVAX has 11 Mbytes of main memory and is not used much by other users or daemons.

Normally, the editing system has to process only about 140 articles each day. These articles arrive sporadically throughout the day so that the editing system processes an article every few minutes. Also, the editing system is continually looping. Either it is processing articles in the input directory or it is waiting for articles to arrive. For my timing tests, I configured the editing system to start up, process all the articles in the input directory and then exit. I then arranged for the shell to keep track of how much CPU time the editing system spent processing the entire directory of input articles. I divided this number by the total number of articles processed, which gave me an average time spent processing each article. The times which resulted from the tests were divided between time spent processing system software and time spent processing user software. The results presented in the tables below give both of these times as well as the total times.

Initially, I suspected that the number of articles in the input directory might have an effect on the average time spent processing each article, so I ran timing tests for a wide range of input article quantities. My hypothesis was that there was probably an optimal number of articles which the editor could process fastest per article.

Table 4-1 shows the time spent processing user software for a wide range of article quantities. For each quantity, I performed three trials. The average of these three trials is given, as well as the average time divided by the number of articles to give the processing time per article. Table 4-2 shows the same statistics for the system software processing time; table 4-3 shows the total of the two. All times are given in seconds.

Articles	Trial 1	Trial 2	Trial 3	Average	Per Article
5	1.6	1.4	1.5	1.5	0.30
10	3.1	3.1	2.9	3.03	0.30
25	8.1	7.6	8.2	7.97	0.32
50	15.1	16.2	15.4	15.6	0.31
100	31.7	33.1	31.7	32.2	0.32
150	48.0	48.3	48.0	48.1	0.32
200	63.7	66.0	63.9	64.5	0.32
250	80.8	81.6	80.7	81.0	0.32
300	97.4	98.0	96.5	97.3	0.32
350	113.8	113.7	113.5	113.7	0.33
400	130.4	130.4	128.6	129.8	0.33
450	148.7	148.3	145.5	147.5	0.33
500	165.4	163.5	165.6	164.8	0.33
1000	322.2	322.2	321.7	322.0	0.32

Table 4-1: User Processing Times

As can be seen from the tables, the differences in user software processing times are small. The differences in system software processing times are more noticeable. My initial expectation was that the processing time for each article would be larger with a smaller number of articles than with a larger number of articles. Based on table 4-3, this

Articles	Trial 1	Trial 2	Trial 3	Average	Per Article
5	0.6	0.7	0.7	0.67	0.134
10	0.9	0.9	1.1	0.97	0.097
25	2.1	2.3	1.9	2.1	0.084
50	4.0	4.5	3.8	4.1	0.082
100	10.9	10.2	8.4	9.8	0.098
150	14.5	14.5	14.3	14.4	0.096
200	19.6	20.1	19.4	19.7	0.099
250	25.8	26.2	26.8	26.3	0.105
300	35.5	33.5	34.7	34.6	0.115
350	44.7	41.5	44.4	43.5	0.124
400	51.1	52.8	53.3	52.4	0.131
450	62.5	60.6	60.8	61.3	0.136
500	67.6	68.9	66.8	67.8	0.136
1000	184.2	212.6	182.8	193.2	0.193

Table 4-2: System Processing Times

turns out to be true, to a point. The figures between 5 and 50 articles decrease as the number of articles increase. I attribute this to the fact that there is a constant amount of initialization that occurs, which naturally has less of an impact on the total processing time as the number of articles increases.

Interestingly, the processing times for article quantities greater than 50 increases as the number of articles increases. By comparing table 4-3 with tables 4-1 and 4-2, we see that the growth in processing time occurs in the system software processing time. The user software processing time remains, for the most part, constant. I suspect that processing more articles requires the editing system to make memory allocation calls. Additionally, processing a greater number of articles also means that more main memory is being used, which can lead to a greater number of page faults. Both of these factors would increase the amount of time spent processing system software.

Articles	Trial 1	Trial 2	Trial 3	Average	Per Article
5	2.2	2.1	2.2	2.2	0.43
10	4.0	4.0	4.0	4.0	0.40
25	10.2	9.9	10.1	10.1	0.40
50	19.1	20.7	19.2	19.7	0.39
100	42.6	43.3	40.1	42.0	0.42
150	62.5	62.8	62.3	62.5	0.42
200	83.3	86.1	83.3	84.2	0.42
250	106.6	107.8	107.5	107.3	0.43
300	132.9	131.5	131.2	131.9	0.44
350	158.5	155.2	157.9	157.2	0.45
400	181.5	183.2	181.9	182.2	0.46
450	211.2	208.9	206.3	208.8	0.46
500	233.0	232.4	232.4	232.6	0.47
1000	506.4	534.8	504.5	515.2	.52

Table 4-3: Total Processing Times

4.2 System Reliability

The editing system correctly processes, on average, about 95 % percent of the total articles processed. Most articles which are incorrectly processed can still be output as readable articles. The number of articles which are completely lost because of processing errors is less than 5 total articles over the two month test period. I have divided article processing errors into three classes:

- program error
- fixable service errors
- unfixable service errors

A *program error*, or simply a bug, is an error made by the program when it incorrectly performs a task that it is expected to correctly perform. For instance, if an article arrives that meets the transmission specifications, within acceptable bounds, and the program

does not properly process it, we classify this type of error as a program error. A *fixable service error* is an error made by the news service which the program can be modified to correctly process. For example, a valid version field for an article is '2takes', meaning that this article is really the first piece of two pieces which comprise the whole article. On the other hand, the string '2tks' is not a string the editor was written to understand, because the string '2takes' is the proper way to indicate such information. If an article arrives with a version field of '2tks' instead, then we will not expect the editor to properly handle it.[5] However, we can easily modify the system to properly handle such cases and, therefore, this is a fixable error. An *unfixable service error* occurs when the news service transmits an article that deviates from the specification in major ways, such that the program *cannot* be modified to properly process such deviations. For instance, if an add to an article is not transmitted with a keyword identical to the correspoding lead, then the program cannot be expected to properly reassemble the article.

Below are three tables of statistics about the number of processing errors made by the program during a two week period. These figures are for the version of the editing system that is running as of the writing of this paper. Table 4-4 shows how many articles were processed each day and how many of those articles were incorrectly processed. For the total number of processing errors a breakdown is given showing how many errors of each type occured. Table 4-5 is similar to table 4-4 but instead shows for each type of error what percentage of the total number of articles processed were processed with that error. Table 4-6 shows for each type of error what percentage of the total number of processing errors were of that type.

From the data we see that approximately 97% of the articles are properly processed. Of the 2.73% of articles which are incorrectly processed, about 1.19% of the errors are program bugs. We can also see from the table that if we eliminate known program bugs and modify the program to properly handle the fixable service errors, we can reduce the number of improperly processed articles to about 1%.

---

[5]Since the transmitted articles are human generated and, for the most part, human read, an editor might use a fragment description of '2tks', even though it violates the standard, because he knows that any other editor seeing that string will know what it means.

Date	Articles	Bugs	Fixable	Unfixable	Total
4/24	140	0	0	1	1
4/25	107	0	1	1	2
4/26	116	2	0	1	3
4/27	157	1	2	2	5
4/28	178	1	1	0	2
4/29	153	1	1	1	3
4/30	140	2	0	2	4
5/1	143	3	0	0	3
5/2	115	4	2	2	8
5/3	124	3	1	1	5
5/4	184	2	1	4	7
5/5	184	3	0	4	7
5/6	147	1	3	0	4
5/7	128	1	0	0	1
Total:	2016	24	12	19	55
Average:	144	1.64	1.14	1.14	3.93

Table 4-4: Number of Processing Errors

Date	Articles	Bugs	Fixable	Unfixable	Total
4/24	140	0	0	0.7	0.7
4/25	107	0	0.9	0.9	1.9
4/26	116	1.7	0	0.9	2.6
4/27	157	6.4	1.3	1.3	3.2
4/28	178	0.6	0.6	0	1.1
4/29	153	0.7	0.7	0.7	2
4/30	140	1.4	0	1.4	2.9
5/1	143	2.1	0	0	2.1
5/2	115	3.5	1.7	1.7	7.0
5/3	124	2.4	0.8	0.8	4.0
5/4	184	1.1	0.5	2.2	3.8
5/5	184	1.6	0	2.2	3.8
5/6	147	0.7	2.0	0	2.7
5/7	128	0.8	0	0	0.8
Average:	144	1.19	0.6	0.94	2.73

Table 4-5: Percentage of Errors Types for Total Articles

Date	Articles	Bugs	Fixable	Unfixable
4/24	140	0	0	100
4/25	107	0	50	50
4/26	116	66.7	0	33.3
4/27	157	20	40	40
4/28	178	50	50	0
4/29	153	33.3	33.3	33.3
4/30	140	50	0	50
5/1	143	100	0	0
5/2	115	50	25	25
5/3	124	60	20	20
5/4	184	28.6	14.3	57.1
5/5	184	42.9	0	57.1
5/6	147	25	75	0
5/7	128	100	0	0
Average:	144	43.6	21.8	34.5

Table 4-6: Percentage of Error Types for Total Errors

Chapter Five

Summary of Results

5.1 Unsolved Problems and Further Work

There are a few major tasks which the existing editing system does not perform. The *New York Times* news wire very rarely sends a duplicate article or a revision of a previous article. Since the current system was built with the *New York Times* in mind, it does not perform version updating or duplicate detection. The design did incorporate these functions, but they are not yet implemented.

Currently, work is being done to create an automatic editor for the *Associated Press* news wire. This will be done using the existing editing system, modifying the parse module as necessary. The *Associated Press* news wire tends to send articles which are not as 'polished' as the *New York Times* articles. The *Associated Press* editor will need to perform version updating and duplicate detection. Much of the supporting code for these functions has been written, such as data structures and modifications to existing code, but the new code has not yet been integrated into the editing system.

The existing editing system also does not perform any filtering of the incoming news articles. All articles which arrive are passed through to the final database. It is not clear whether filtering at this level is desired. It would be easy to have the editing system perform some filtering, but the usefulness of filtering has not been determined.

5.2 Conclusions

The current system reliably performs the tasks of data formatting and article reassembly on an average of 145 articles each day, with an 2.7 % error rate. It requires on average 0.40 seconds of CPU time to process each article. Approximately 250 users are serviced by the news articles processed by this system.

It was my intent to design and build a system which adhered to good design rules, performed reliably and provided for easy adaptation to other news wire services. The system which resulted from this project is well designed with respect to modularity and data abstraction. It is also robust in the face of a system crash, able to recover all of its state. Additionally, it is easy to 'fine tune' certain parameters that may need changing to adapt to changes within the news wire service. Lastly, all of the details about the *New York Times* news wire have been restricted to the parse module. Any future news wire editing systems can be built by just modifying the parse module as necessary.

References

[1] Sandra L. Puncekar (editor).
High-speed wire service transmission guidelines.
American Newspaper Publisher's Association Research Institute, 1979.

[2] Kernighan, Brian W. and Ritchie, Dennis M.
The C Programming Language
1979.

[3] Liskov, Barbara, et al.
CLU Reference Manual
1981.

[4] Gifford, David K., et al.
Boston Community Information System User's Manual.
Technical Report TR-373, The Laboratory for Computer Science, Massachusetts of Technology, 545 Technology Square, Cambridge, Massachusetts 02139, September, 1986.

[5] Ross N. Dreyer.
Automatic Editing of Newspaper Articles for Electronic Publishing.
Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, June, 1986.