An Interactive Program to Access Remotely Held Weather Data

Weather version 4.0

AMY WONG
SUMMER EMPLOYMENT PROGRAM

UNIVERSITY CORPORATION FOR ATMOSPHERIC RESEARCH
BOULDER, COLORADO
An Interactive Program to Access Remotely Held Weather Data

Amy Wong
Unidata
University Corporation for Atmospheric Research
Summer Employment Program 1991
INTRODUCTION

Weather v.3, which accesses the most recent weather forecast was revised and updated so it may be incorporated into a larger end-user package in the future, allowing the user access to a variety of atmospheric information such as storm warnings, satellite images, and earthquake information. Users are allowed to specify a city in the United States and type of weather report they desire by inputting command line arguments when they invoke the program.

The revised weather access program, weather v.4, has a graphical user interface using the OpenLook format. The non-technical user will find this program simpler to learn and use than weather v.3. Users are not required to know the report codes as they were with the previous program. Instead, they are able to select the report type they desire by scrolling down a list. Users are also allowed access to the most recent weather information outside of the United States. Weather v.4 is much more versatile than weather v.3. It saves the user a considerable amount of system time when asking for large amounts of data.
MATERIALS AND METHODS

The materials I used to develop my weather access project include: manuals describing remote procedure calls (RPC), manuals describing external data representation, and the local data manager database query version 3 (Ldmdbq3) program showing the use of the SENDME procedure and RPC calls. I prepared to program the graphical user interface of my program by referring to the xview toolkit programming manual. I developed my project on a Sun workstation in a UNIX environment with a C compiler and vi editor. Debugging was done with dbx and revisions of the program were registered with the remote control system (RCS) for reference purposes.

The method I used involves the principles of top-down design, coding, and testing. However, a bottom-up approach was used during the beginning stages of development for the lower level modules. For an overall view of the stages of development (Figure 1) I designed a structure chart showing all procedure calls made from the main procedure and the order in which the calls were made.

The structure chart helps map out the program in a "Divide and Conquer" manner in which the division of modules is roughly equal in size and complexity. This helps in employing good software engineering techniques. This technique made it simple to hide assumptions in order to limit their impact on the rest of the program if changes were ever needed for these modules.

I developed data flow diagrams (Figures 2a,b,c) to illustrate the flow of data from one process to the next. This helped me, as the programmer, to visualize how each module needed to be developed in order to handle the conversion of data from one form to the next. The data flow diagram also helped my supervisor see if I had a complete understanding of the project, thus he could provide me with valuable feedback. The data dictionary, which accompanies the data flow diagram, defines the terms used in the data flow diagram and in the processing narratives, which describe each functional primitive of the data flow diagram. The structure chart and data flow diagrams are shown in the following pages.

All the methods mentioned above were used in order to follow good software engineering techniques and to aid in the design of the program. These methods allowed more time to be spent actually coding rather than fixing code.

Since my program, weather v.4, is a revision of weather v.3, I obtained a performance analysis between the two programs to compare the amount of system time the programs were using and to compare the efficiency of the different algorithms they used. I used the C tools, time and gprof, to perform the analysis. The results of the analysis are discussed in the Performance Analysis section. The results, displayed in Tables 1 and 2, are both interesting and informative.
PROCESSING NARRATIVES

Processing Narratives explain the function of the procedures diagrammed in the structure chart. The processing narratives refer to weather v.4 without the inclusion of the code for the graphical user interface.

2.0 Establish Service

This procedure creates a TCP/IP based RPC service transport associated with the socket. If the socket is bound already a new socket is created, otherwise the socket is bound to an arbitrary port. A transient service is then invoked to register the program on the local machine's portmap service. Establishing a transient service is important because this is the channel for communication between the program and the server. Two transient services are established between the program and the server. The request channel is used for sending requests to the server and the callback channel is used for listening for the server's response.

2.1.1 Parse Command Line

This procedure takes the command line as input and parses it into the separate pieces of information it contains. The command line indicates a specific report type and station to use to access weather information for the user. If either the report type or station is not indicated, the default values are used. The default report type is the metro report, FQUS01, and the default station is Denver, KDEN. The report type and station are used to form the pattern string to be used in accessing the server for weather data. If an error occurs in the command line, it will be detected and the correct usage of the command line is printed out.

2.1.2 Sendme

This procedure takes in as input information needed to fill the fields of the args data structure, and the hostname of the server currently being used. The args data structure contains information such as the program number, pattern string, number of matches, and the start and stop times to look for weather data, which sendme sends on to the server. RPC calls associated with the specified program number, version number, and procedure number on the machine host are invoked to determine the RPC status of the program. If the status indicates a success, then it is possible to access the server to locate the desired weather data. Sendme then makes a request to the server for the weather data.

2.1.3 Svc_run

This procedure waits for a response from the server. When a response has been made, sink_3 is called.

2.1.4 Sink_3

This procedure receives and processes the response svc_run sends it. If the server returns a successful find of information then sink_3 receives the arguments and the formatted output stream from the server and prints out the weather information. If the server returns that all available data have been found within the time range, then sink_3 acknowledges this and frees the port, thus terminating the program successfully. If the server returns an error, then sink_3 prints out an error message and control returns to the calling procedure.
DATA DICTIONARY

The data dictionary is the notation for describing the contents of information as if flows through a system. It contains the definitions of terms used in the processing narratives and the data flow diagram. The data dictionary is composed of composite information, which contains terms further defined within the dictionary, as well as elementary data, which needs no further explanation.

default hostname is unidata.ucar.edu.

default report type is FQUS01.
   * the metro report *

time sequence is composed of an automatically generated sequence of dates and times,
   starting from the current date and time and computing
   the dates and times of the last 12 hours.

default station is KDEN.
   * Denver, CO, USA *

input line is composed of options for quiet mode
   or hostname
   or number of reports
   or report type
   or station.

   * if one or none of the options are specified, the default values are used *

output stream is composed of the requested weather information.

pattern string is composed of the report type and station.

port is composed of the communication line taken up by the program.

prognunm is composed of the number returned from the registration process.
   * The prognunm gives an id for the program so that communication can take place between
     the server and the calling procedures *

quiet mode is suppression of verbose mode.

report type is composed of a six character sequence identifying the type of weather report
   desired.

   * examples include FQUS01 for the metro report and FEUS01 for the extended report *

RPC status is composed of a boolean specifying the ability to make RPC calls to the server
   or not.

socket is the communication line established between the server and the program.

station is composed of a four character sequence identifying the city desired for weather data.
examples include KDEN, KLAX, and MMEX. The first character indicates in which country the city resides and the following three characters are based on the city’s airport code.

TCP/IP based RPC service transport is the type of service transport established.

usage of input line is composed of all available options offered to users to specify what type of data they desire.

verbose mode is the printing of the report type and station being accessed.

weather information is composed of data corresponding to the specified report type and station and within the time range.
Figure 1

STRUCTURE CHART

Weather v.4
- Main()
- Get report type and station
- Establish communication lines
- Wait for response from server
- Fill args data structure and send request to server
- Process server's answer
- Print weather
- No data found
- return to Sync_run()
DATA FLOW DIAGRAM
Level 1

1.0
Weather access Program version 4

command line

weather data for specified report type, station, and time range

Figure 2a

Level 2

2.0
Establish Service

rendezvous point

2.1
Send Request to Server

weather data

Figure 2b
DATA FLOW DIAGRAM
Expansion of Bubble 2.1

command line

2.1.1
Parse Command Line

prognum, num reports, pattern string

2.1.2
Sendme

RPC status, args

Server

weather data

2.1.4
Sink_3

response

2.1.3
Svc_run

response

Weather Data

Figure 2c
PERFORMANCE ANALYSIS

Both weather v.3 and v.4 programs access the same data, but they differ in the algorithms they use to access this data.

The weather v.3 program implements the algorithm of multiple dbget calls. An overview of the program is displayed in Figure 3. The user identifies the report type and station code as command line arguments. A sequence of dates and times, starting from the most current date and time and going back 12 hours, is then generated. A communications channel between the program and the server is then established. The dbget procedure makes a call to the server sending the server the report type, station code, and one date and time pair via this channel. The server then searches for weather data matching this pattern. When it successfully finds a match the weather data is sent back, otherwise a negative response is returned. After one product is found the data is printed and the program terminates. With a negative response a new pattern string is formed consisting of the same report type, station, and the next date and time pair and the pattern sent to the server again. The server is continually accessed in this way until a match has been found or until the maximum number of tries allowed has been reached.

In contrast, the weather v.4 program establishes two communications channels. All requests for weather data are sent to the server at once instead of one by one as with weather v.3. The server then begins searching for weather data matching the pattern string. It continues searching until it finds a match. Once a match has been found then the server sends the data back to the calling program to be printed out and then the program will again wait for a response from the server. The server continues searching and sending back reports until the number of desired weather reports have been found or until it reaches the end of the time range. The server then indicates that no more data will be sent and the connection line between the server and the program is severed. If no forecasts had been available, then a message indicating so would have been printed and the program would have exited immediately.

Making only one network transaction can increase the apparent speed of a program since there is always a time delay with every network transaction due to multiple users accessing the network. The fewer calls a program needs to make to the network, the lower will be the running time of the program. This can make a significant difference when the network is especially overloaded with users. The tradeoff is that the server may take longer to process this more general request.

I used performance analysis tools to test the difference between the algorithms. The first tool I used was the time command which is built in to the C shell. The command displays the statistics about how a program uses the system resources as well as the run time of the program. The average results from the time command for weather v.3 over five runs is:

<table>
<thead>
<tr>
<th>user time</th>
<th>system time</th>
<th>elapsed time</th>
<th>percent of user and system time as fraction of the elapsed time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.02u</td>
<td>0.14s</td>
<td>0:00</td>
<td>52%</td>
</tr>
</tbody>
</table>

And the average results for weather v.4 over five runs is:

<table>
<thead>
<tr>
<th>user time</th>
<th>system time</th>
<th>elapsed time</th>
<th>percent of user and system time as fraction of the elapsed time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0u</td>
<td>0.3s</td>
<td>0:03</td>
<td>10.4%</td>
</tr>
</tbody>
</table>

These results are rounded off to the nearest half second. Looking at these results, we see that weather v.3 takes more user time, but weather v.4 spends more time in the operating system and
has a longer elapsed time, although the difference is quite minimal. This is interesting because one would expect a new version of a program to perform better. Notice, however, the difference in the percentage of user and system time as a fraction of the elapsed time, that is the CPU time. The 52% from weather v.3 indicates that the program does not have to wait long for an external reply or for system resources. Weather v.4, however, only gets 10.4% of CPU time. That is to say, the program is blocked while waiting for the server’s response. The time command does not tell much else about what is causing the differences in elapsed time or CPU time. The time command is useful only for obtaining an overview of how long a program takes to run.

For a more detailed level of analysis I generated a call graph profile of both versions of weather using the C shell command: gprof. The call graph profile assists in indicating procedures which, if revised, can optimize the performance of a program. The results of a call graph displays a list of which modules are called by other modules, and which modules call other modules. The output is really voluminous so I will only include the most relevant portions of the results.

The headings of the call graph profile results are explained below:

function entries:

%time: the percentage of the total time of the program accounted for by this function and its descendents.

called: the number of times this function is called (other than recursive calls).

name: the name of the function, with an indication of its membership in a cycle, if any.

parent listings:

called the number of times this function is called by this parent. This is the numerator of the fraction which divides up the function's time to its parents.
total the number of times this function was called by all of its parents. This is the denominator of the propagation fraction.

parents the name of this parent, with an indication of the parent's membership in a cycle, if any.

children listings:

called the number of times this child is called by this function. This is the numerator of the propagation fraction for this child.
total the number of times this child is called by all functions. This is the denominator of the propagation fraction.

children the name of this child, and an indication of its membership in a cycle, if any.

The results from weather v.4 are listed in Table 1.

In examining the results, I noticed that for both versions, the procedures which account for most of the running time of the program are made by system calls, which are out of my control in terms of being able to revise them to optimize the performance of the program. Running time refers to user time and system time together. In weather v.4 calls to malloc (memory allocation) and its descendents account for 90.6% of the running time. The number of calls I make to malloc
directly is 1 out of 42, in ParseCommandLine. There is not much I can do to cut down on the calls to malloc to improve the running time of the program. The first procedure I encountered I which could possibly revised is the call to gethostbyname (Table 2) which makes a call to get the host name on which the program is running. Gethostbyname and its descendants take up 30.0% of the program's running time. Instead of making a gethostbyname call I could do a client broadcast call for any host which is available to answer, but that is not part of the specifications of the program. The program was designed to use the default host, unidata.ucar.edu, or a host specified by the user.

Looking at the flat profile results (Table 3), which gives the list of procedures which should be revised to optimize the performance of weather v.4, at the end of the call graph we see that for weather v.4, the 33.3% of the running time of the program is taken up by malloc, 33.3% is taken up by a system call, brk, which is called by malloc, and 33.3% is taken up by a variable mcount. The interesting thing is that mcount is from the gprof tool, itself. This tells us that the complexity of the program isn't significant enough to really benefit from performing a call graph analysis. The first procedure which is not a system call is the ParseCommandLine procedure, but this procedure takes up no registered running time and is thus considered insignificant in causing any time delays. Therefore, the call graph analysis did not give any insight of ways to optimize performance, but I am assured that I am not using an expensive algorithm for weather v.4.

Knowing this, I must analyze why the running time for weather v.4 came out longer than weather v.3. I have eliminated the possibility of it being from the algorithm implemented. And so now I look toward the difference in the way the server is accessed as an explanation. In weather v.3 one communications channel is opened to connect the program with the server. This is the channel used to send requests to the server and it is also the channel through which the server responds to the program. Recall that the dbget procedure makes multiple calls via this channel. The calls are short through and keep the systems attention to a minimum. In contrast, weather v.4 establishes two communications channels between the program and the server. The request channel is for making requests to the server and the callback channel is for listening to the server's responses. Establishing two channels is more expensive from a system point of view because more system calls are made and more of the system resources are used. Thus, this partly explains the difference between the running times of the two programs.

The question then arises as to why are we implement the program this way if it is more costly. The answer lies in the versatility that weather v.4 now offers. Consider the case of requesting large amounts of data. Weather v.4 is able to handle regular expressions so if the user wanted to place a wild card in the command line, it is perfectly acceptable. The users can also expand on the range of times searched and look for multiple reports at the same time. In weather v.4 the user may also specify the number of reports desired. Weather v.4 is capable of handling such versatility in requests with ease. The expression is sent to the server and the server does all the work in searching for the range of data. In comparison, if the user wanted this to be done with weather v.3, things would not be so easy. Weather v.3 can not handle regular expressions and so the inclusion of wild cards would cause an error in the program. To make the inclusion of regular expressions acceptable in weather v.3, the entire expression would have to be expanded to all possible cases and requests sent to the server one at a time as usual. For example, if the user requested a search for two different report types, but the same station, and date and time sequence, in weather v.3, a pattern string would first have to be formed for the first report type, station, and the first date and time pair, and the request sent to the server. In the worst case, the server would have to be accessed 12 times for the first report type, one for each of the date and time pairs. Afterward, another pattern string would have to be formed with the second report type, the same station, and the first date and time pair, and a request sent to the server. In the worst case, the server would have to be accessed another 12 times for the second report type. This is a total of 24 calls to the server as compared to 1 weather v.4 would have to make to the server with the same request. Just imagine the number of calls which would have to be made if a wild card were
included in the regular expression. The amount of calls could easily increase exponentially. Things get very complicated at this point and extremely time consuming.

And so we see that weather v.4 has a longer running time than weather v.3 because it establishes two communications channels and is more complex, in terms of what it is able to handle and perform. I then thought that perhaps I could optimize the performance of weather v.4 by establishing only one communications channel as in weather v.3 and make weather v.4 even more appealing. Observe the set up of having only one communications channel, as in weather v.3:

![Diagram](chart)

**Figure 4**

We see that the client side, which refers to the program, calls the server and the server grabs hold of the client while it is searching for a match. The server waits for a match to be found until sending the data and control back to the client.

In contrast, the set up for callback, as in weather v.4, is as follows:

![Diagram](chart)

**Figure 5**

The client sends a request to the server and the server acknowledges receipt of the request and immediately returns control to the client, which proceeds to wait for a response at the rendezvous point. The server in the meantime has forked, severing all communication with the program until it
finds a match. The server searches its database for data, and once found the server finds the
client waiting at the rendezvous point for a response. The server sends the response to be
processed and then once again returns to start searching for another match. This continues until the
number of reports requested has been found or the end of the time range has been reached.

For small requests, both designs would have a quick response, although a quicker one for
Figure 4, as we have already discussed. For large requests, though, the design in Figure 4 will
hold on to the client until all data has been found. The client sits there while the server collects
data. If an especially large request is made, the client will timeout because it assumes an error has
occurred if the client is not returned within a reasonable amount of time. Figure 5, however, does
not have this problem because matches are returned to the program to be processed as they are
found so the client does not have to wait for extensive periods of time. Weather v.4 will wait for
as long as needed to gather all the data. Establishing two communications channels was meant to
benefit accessing large amounts of data.

Therefore, weather v.4 with two communications channels is a benefit rather than a
hindrance to the program. The greater running time for small requests in weather v.4 can be
forgiven because the difference is minimal and in the long run the two communications channels
plus the additional features offered is superior to weather v.3.
Figure 3

* Level 1 is the same as Figure 2a
CALL GRAPH RESULTS for Weather v.4

<table>
<thead>
<tr>
<th>%time</th>
<th>called/total</th>
<th>name</th>
<th>children</th>
</tr>
</thead>
<tbody>
<tr>
<td>1/42</td>
<td>_ParseCommandLine</td>
<td>_findbuf</td>
<td></td>
</tr>
<tr>
<td>1/42</td>
<td></td>
<td>_on_exit</td>
<td></td>
</tr>
<tr>
<td>1/42</td>
<td></td>
<td>_callrpc</td>
<td></td>
</tr>
<tr>
<td>1/42</td>
<td></td>
<td>_xprt_register</td>
<td></td>
</tr>
<tr>
<td>1/42</td>
<td></td>
<td>_svc_register</td>
<td></td>
</tr>
<tr>
<td>1/42</td>
<td></td>
<td>_xdr_string</td>
<td></td>
</tr>
<tr>
<td>1/42</td>
<td></td>
<td>_default_domain</td>
<td></td>
</tr>
<tr>
<td>1/42</td>
<td></td>
<td>_load_dom_binding</td>
<td></td>
</tr>
<tr>
<td>1/42</td>
<td></td>
<td>_v2domatch</td>
<td></td>
</tr>
<tr>
<td>2/42</td>
<td></td>
<td>_xdr_bytes</td>
<td></td>
</tr>
<tr>
<td>2/42</td>
<td></td>
<td>_svctcp_create</td>
<td></td>
</tr>
<tr>
<td>2/42</td>
<td></td>
<td>_makefd_xpirt</td>
<td></td>
</tr>
<tr>
<td>2/42</td>
<td></td>
<td>_xdrrec_create</td>
<td></td>
</tr>
<tr>
<td>5/42</td>
<td></td>
<td>_makenode</td>
<td></td>
</tr>
<tr>
<td>7/42</td>
<td></td>
<td>_calloc</td>
<td></td>
</tr>
<tr>
<td>12/42</td>
<td></td>
<td>_clntudp_bufcreate</td>
<td></td>
</tr>
</tbody>
</table>

90.6  42     _malloc
      3/3     _morecore
      42/42   _demote

Table 1
CALL GRAPH RESULTS for Weather v.4 con't

<table>
<thead>
<tr>
<th>%time</th>
<th>called/total</th>
<th>parents</th>
<th>name</th>
<th>children</th>
</tr>
</thead>
<tbody>
<tr>
<td>30.0</td>
<td>1/1</td>
<td>_callrpc</td>
<td>_gethostbyname</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td></td>
<td>_yp_match</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1/3</td>
<td></td>
<td>_hostdata</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1/1</td>
<td></td>
<td>_sethostent</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1/1</td>
<td></td>
<td>_interpret</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1/15</td>
<td></td>
<td>_free</td>
<td></td>
</tr>
</tbody>
</table>

Table 2

FLAT PROFILE RESULTS for Weather v.4

<table>
<thead>
<tr>
<th>%time</th>
<th>cumulative seconds</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>33.3</td>
<td>0.02</td>
<td>_malloc</td>
</tr>
<tr>
<td>33.3</td>
<td>0.04</td>
<td>_sbrk</td>
</tr>
<tr>
<td>33.3</td>
<td>0.06</td>
<td>mcountr</td>
</tr>
<tr>
<td>0.0</td>
<td>0.06</td>
<td>_xdr_u_long</td>
</tr>
<tr>
<td>0.0</td>
<td>0.06</td>
<td>_xdrmem_putlong</td>
</tr>
<tr>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>0.0</td>
<td>0.06</td>
<td>_ParseCommandLine</td>
</tr>
</tbody>
</table>

Table 3
CONCLUSION

Weather v.4 may have more overhead in system design, but in the long run it is minimal compared to the overhead associated with weather v.3 and is actually a benefit when requesting a large amount of data. Weather v.4 is capable of handling a higher volume of data and can satisfy more varied requests than weather v.3 while only making one network transaction. The implementation of a graphical user interface in weather v.4 is also an added benefit which users will find helpful and user-friendly. The program will appeal to many users because of its ease in usage and because it provides data for something which everyone encounters everyday—weather.

ACKNOWLEDGEMENTS

Many thanks to Glenn Davis and Mitch Baltuch for their constructive criticisms and continual support throughout the development of my program. Thank you to Joann Dennett and Heather Draper for their help in editing this paper.
REFERENCES

