Using REXX in a UNIX Environment to Manage Network Operations

Lee Krystek
Boole and Babbage

1. Using REXX in a Unix Environment to Manage Network Operations:

Lee Krystek - Software Manager
Boole and Babbage Network Services

Abstract: When designing our network management and control product, we needed to provide a way for users to construct scripts to control any foreign system they might need to interface with via that foreign system's console. We selected REXX as this tool. Before we could use it, we had to augment the language to give it the capability to be started automatically, connect to those foreign systems, and manipulate our relational database.

1.1. The COMMAND/Post Product:

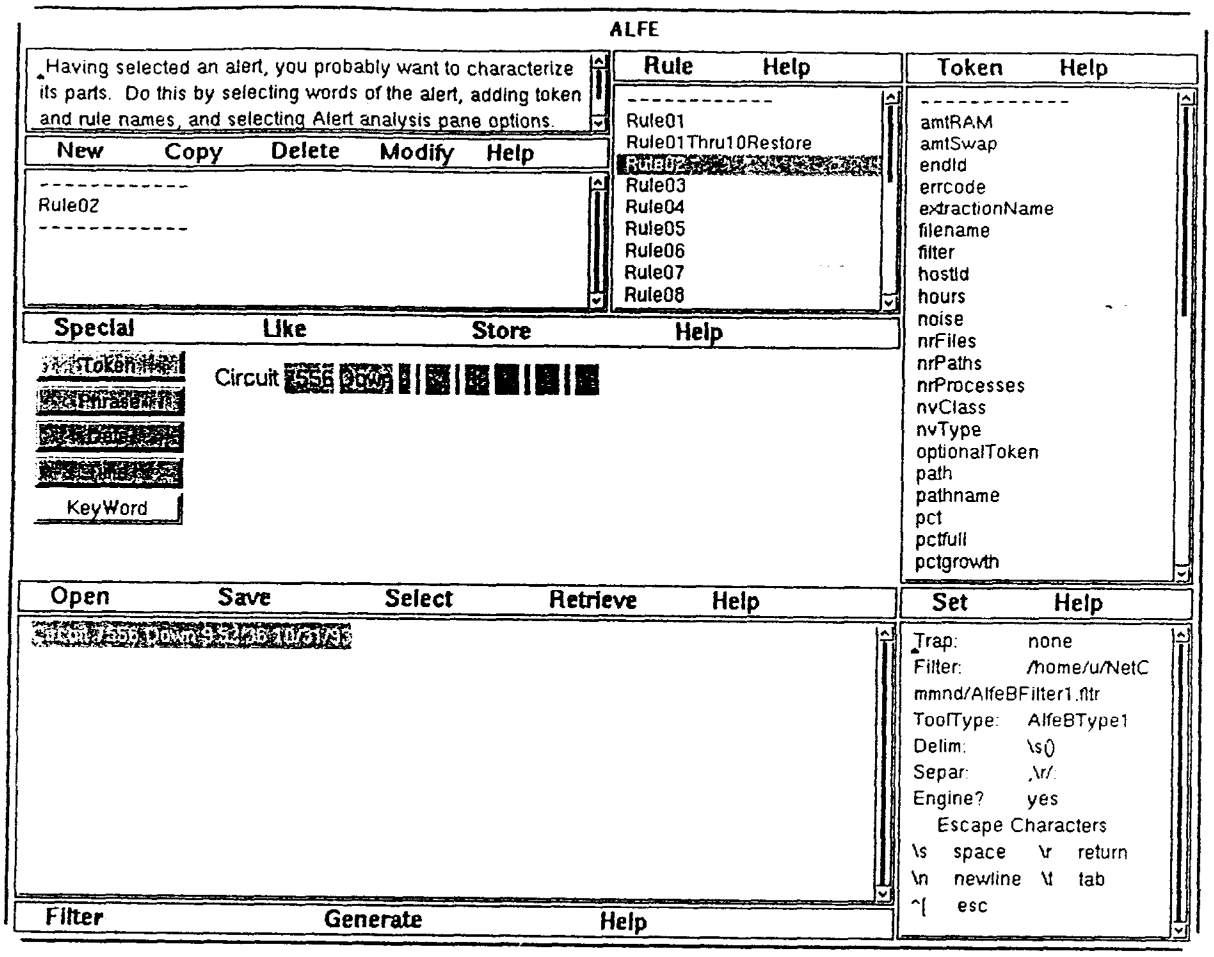
Several years ago Boole and Babbage recognized the need for a product that would be a focal point for network and systems management operations. This product would monitor and control network equipment, computer systems, and even application programs. Of special interest were non-SNA and non-SMNP systems which did not support any network management protocol.

COMMAND/Post runs on a UNIX workstations. Initially the SUN SPARC series of processors was used, however, porting to other UNIX systems is underway. A typical COMMAND/Post system consists of one or more (perhaps even as many as 50 at a large site) workstations with color monitors running a GUI such as Open Windows or Motif. The system is composed of modules written in Smalltalk (an object oriented language) for the user interface and "C" for the more intensive processing tasks. Sybase, a relational database provides the data storage.

COMMAND/Post will typically connect to a network system, such as a modem monitor or T1-monitor or a computer, through the system's printer port and console. (These type of systems are typically referred to as "Element Managers" or EMs because they control one class of element in the whole network.)

An EM's printer port will often produce interesting information such as the failure of a modem or communication line. COMMAND/Post has a tool, called ALFE (ALERT LOGIC FILTER EDITOR), that implementers use through user friendly dialog screens, to construct an alert "filter." (Figure 1) The filter searches the message stream from the EM's printer port and recognizes important messages. The filter parses those messages and then creates an "alert" in the COMMAND/Post system using data obtained from the message. The filter assigns the alert a priority and an classification based on the OSI standard for network management. COMMAND/Post records the actions of supervisors and operators and tracks how the alert is handled and resolved.

COMMAND/Post operators use terminal emulations windows to access the EMs from their workstations. This allows operators to work on problems that might involve a dozen EM's without leaving their seat. (Figure 2)



ALFE

Having selected an alert, you probably want to characterize its parts. Do this by selecting words of the alert, adding token and rule names, and selecting Alert analysis pane options.

New Copy Delete Modify Help

```
------------
Rule02
------------
```

Rule Help

```
------------
Rule01
Rule01Thru10Restore
Rule02
Rule03
Rule04
Rule05
Rule06
Rule07
Rule08
```

Token Help

```
------------
amtRAM
amtSwap
endId
errcode
extractionName
filename
filter
hostId
hours
noise
nrFiles
nrPaths
nrProcesses
nvClass
nvType
optionalToken
path
pathname
pct
pctfull
pctgrowth
```

Special Like Store Help

Token
Phrase
Data
[illegible]
KeyWord

Circuit 7556 Down [illegible]

Open Save Select Retrieve Help

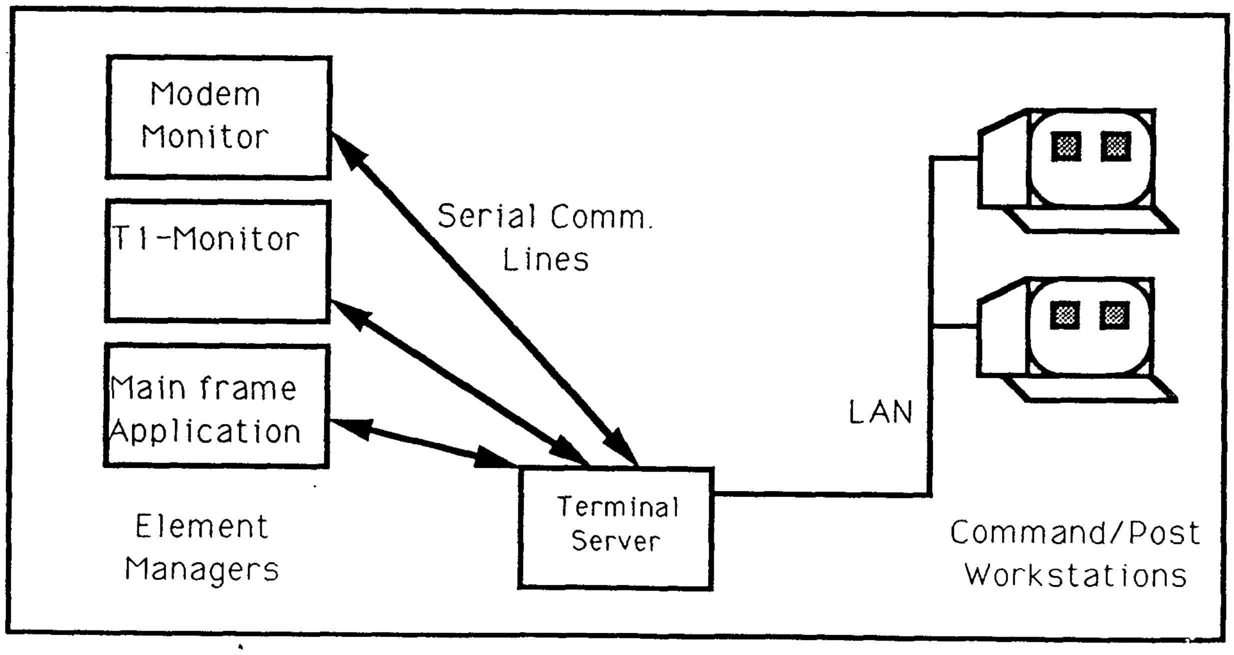
```
Circuit 7556 Down 9:52:36 10/31/93
```

Filter Generate Help

Set Help

```
Trap:       none
Filter:     /home/u/NetC
mmnd/AlfeBFilter1.fltr
ToolType:   AlfeBType1
Delim:      \s()
Separ:      ,\r/.
Engine?     yes
   Escape Characters
\s  space    \r  return
\n  newline  \t  tab
^[  esc
```

An ALFE Screen - Figure 1.



Emulation Connections - Figure 2.

1.2. The Auto Operations Requirement

It became apparent after the initial release of COMMAND/Post that our prospective customers wanted to have the system support automated operations. That is, to have COMMAND/Post not only detect alerts and display them, but to also automatically take actions based on an alerts or alerts received from a single EM, or on a combination of alerts from several EMs.

COMMAND/Post already had the ability to connect with the system consoles for the various EMs. Therefore it seemed logical that if an automated operations facility could be built we could send commands to the appropriate EM, through the emulations, to get an EM to take the desired action.

The automated operations facility needed two parts. First, some kind of detection mechanism that would allow the triggering alert, or combination of alerts, to be recognized. Second, another mechanism that could have a conversion with an EM's console, as if it were a human operator, in order to enter the commands necessary to get the EM to carry out the desired action.

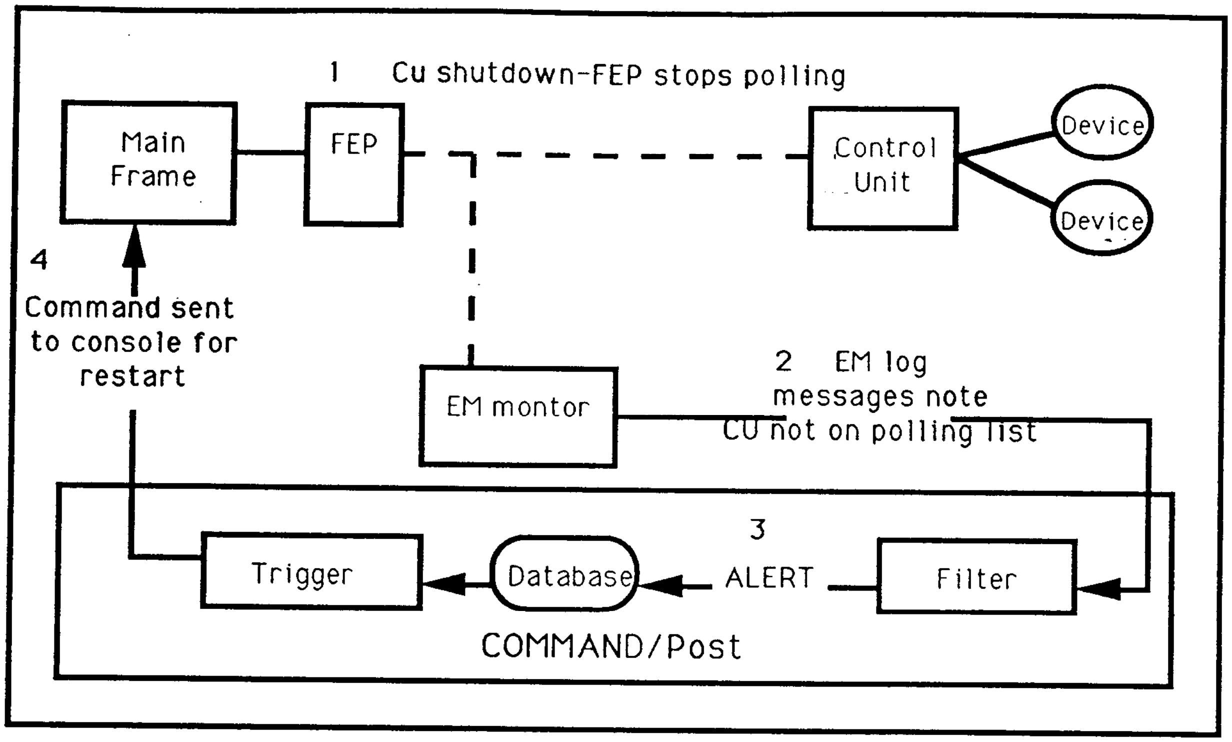
A simple example of an automated operation, though no longer a problem on most networks, is automatic restart of polling on a communication line. A Front End Processor (FEP) is polling several control units at remote sites across a single wide area network line. One of the controllers goes off line for a period of time and the FEP automatically drops that controller from the polling list. When the controller came back on line an operator would command the FEP to add that controller back into the polling list. Under COMMAND/Post automated operations, an EM monitoring that communications line would report the failure of the control unit. COMMAND/Post filters would detect this as an alert, and would then trigger an automatic operation to send a command to the FEP to add the controller back to the list. If the controller failed to respond over a specified period of time, a high-priority alert could be generated to inform the operator that a situation had occurred that could not be remedied through auto operations. (Figure 3)

The design of the alert detection and trigger mechanism took advantage of COMMAND/Post's relational database mechanisms for storing and accessing data. A graphic window display (known as a selector) already existed to select alerts. The implementer uses the selector and the mouse to click on certain rules that describe the alert(s) to be shown on an alert display window. This idea was extended to allow groups of alerts to be detected. When a specified combination of alerts is detected instead of having the alert(s) appear in a window a "trigger" would fire and the auto-operation would start. (Figure 4)

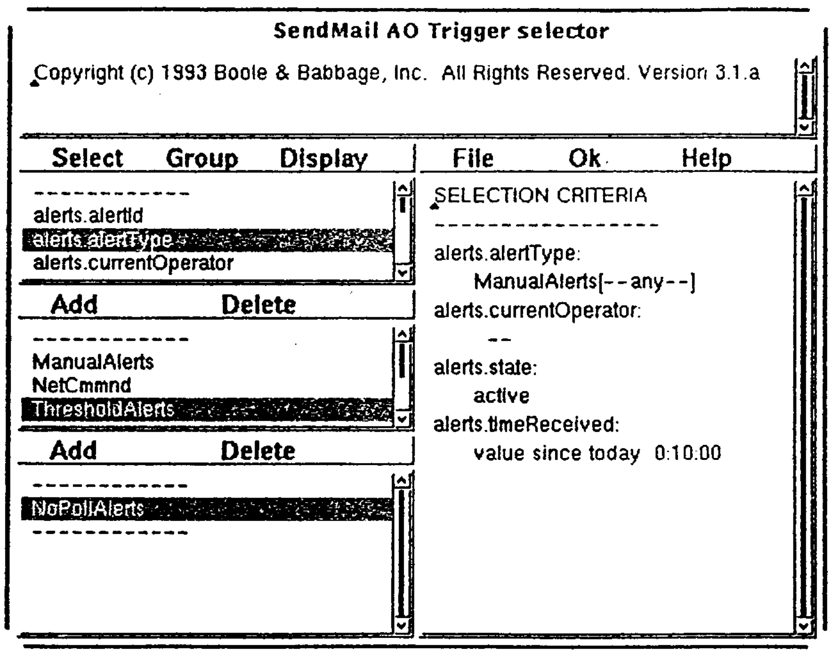
Once the detection facility was decided, the mechanism to allow the system to carry on a conversation with an EM console was next. An augmented version of REXX was chosen for that mechanism.

1.3. Why REXX?

The decision to use REXX was based on several factors. The actual REXX


1 Cu shutdown-FEP stops polling
Main Frame
FEP
Control Unit
Device
Device
4
Command sent to console for restart
EM montor
2 EM log messages note CU not on polling list
3
Trigger
Database
ALERT
Filter
COMMAND/Post

Restarting a Controller by Auto-Operation - Figure 3.


SendMail AO Trigger selector

Copyright (c) 1993 Boole & Babbage, Inc. All Rights Reserved. Version 3.1.a

```
------------
alerts.alertId
alerts.alertType
alerts.currentOperator

------------
ManualAlerts
NetCmmnd
ThresholdAlerts

------------
NoPollAlerts
------------
```

```
SELECTION CRITERIA
-----------------
alerts.alertType:
    ManualAlerts[--any--]
alerts.currentOperator:
    --
alerts.state:
    active
alerts.timeReceived:
    value since today 0:10:00
```

A Selector - Figure 4.

product chosen was uni-REXX from the Workstation Group.

1.3.1. *REXX was already an established language for auto-operations on Boole's mainframe products.*

In addition to COMMAND/Post, Boole already had some main frame products that incorporated auto-operations. They used REXX extensively. It was decided there would be an advantage to keep the auto-operations language consistent between the products.

Using REXX also allowed us to draw upon the experience of our main frame programmers, and some of the extensions to the REXX language to support database operations were based on insights provided by the mainframe REXX group.

1.3.2. *Some of the auto-operations scripts would be written by customers and a language already familiar to IBM type main frame operators was desired.*

Although COMMAND/Post is a Unix-based product, many of the audience for it have their roots in the IBM culture where REXX is widely used. By choosing a familiar language it was hoped there would be less fear and resistance by customers to writing their own REXX scripts.

1.3.3. *REXX's ability to parse data strings would make analysis of messages coming from the EM's easier.*

It was expected that much of the function of the scripts would be to respond to messages coming from the EM systems. The REXX "parse" facility allows most of these messages to be handled without a lot of programming. The "parse" statement is usually easy for even a novice programmer to understand.

1.3.4. *REXX's ability to pass commands to underlying environments makes it easy to address COMMAND/Post's database.*

The extensions to the database were critical if we were to be able to write easy to read scripts. The ADDRESS instruction allowed us pass SQL command directly to the database. Also important was the ability of REXX to create new variables of any type "on the fly" as data was returned from the database. This eliminated the need for a rigid, complicated structure (as used in "C" when getting data back from the DB).

1.3.5. *Use of a "light," interpretive language makes debugging easier for non-professional users.*

Though interpretive languages execute more slowly than compiled languages they are often easier for the novice to debug since there is no compilation wait involved. Also unless the compiled language has a sophisticated debugger, the source line is not displayed in association with a run-time error. In addition, REXX has a built in trace feature which is easily used.

The use of a "light" language that didn't need extensive variable declarations, etc. was also an advantage. While such languages become increasingly difficult to maintain as a single program grows larger and more structure is needed, because of the anticipated size of the scripts (500 lines or less), that was not a concern.

1.4. External Access

The first change we made to REXX was to give it the capability to connect with the EM's. This was more complicated than simply opening a new file descriptor to a new tty port. Connection to EM's for filtering and emulation are managed as resources by COMMAND/Post. A connection to a EM's system console might be used for a period of time by an operator via an emulation, and later reassigned by the system for use by auto-operation via a REXX program.

Connections were made from a program to a physical port using the UNIX socket/stream facility. The actual physical ports might be a tty, or, more likely a port on a terminal server connected remotely, via LAN or WAN, from the workstation where the REXX was actually executing. The resource management system was designed to make the details of the actual connection transparent to the connecting program. This means the REXX program need only know a single name to invoke the connection.

In order to allow the REXX to connect through COMMAND/Post's resource management system several functions were added to the language by inserting additional code into the REXX interpreter so it could use UNIX sockets and streams:

```
<fd> = ao_targetConnect(<name>)

ao_targetClose(<fd>)

ao_targetComm(<fd>,<function>,<data>,<length>,<pos>)
```

The first function, **ao_targetConnect**, requests the opening of a connection to a named port. The name implies more than simply a physical port. It also implies a pathway to get there and, in some cases, a terminal emulation appropriate to the external target system on the other side of the port. These are defined externally to REXX by COMMAND/Post's system management faciiity.

A file descriptor, or more appropriately a "handle" is returned by **ao_targetConnect** to identify the path for future communications calls.

The **ao_targetClose** function simply reverses the connect function closing down the path. The handle from **ao_targetConnect** is the single argument to **ao_targetClose.**

The third function, **ao_targetComm**, actually carries out the transfer of data between the REXX program and the target system.

1.4.1. Application Program Interface

When it came to actually talking to the target system we were faced with an additional problem. Usually the device a REXX EXEC needs to talk to is a system console. That means the program would be responding to the commands we sent it with data (including our own full duplex echo) as well as occasionally sending out, from our point of view, random lines of data as the result of activity on the system. How could we develop an interface for REXX that would allow us to send data at will and handle messages from the target when they came in at any time? Turning to an interrupt model, where we would sit in a wait state until an incoming message would trigger a designated REXX function seemed to be too complicated for easy use by most of our customers, especially when more than one target system might be involved in a single REXX program.

Instead we decided to use an Application Programming Interface (API) to interact with the target. The API we developed was similar to that defined by IBM as the "IBM PC 3270 Emulation Program, Entry Level, High-Level Language Application Program Interface" or EEHLLAPI. Where the IBM was targeted to a 3270 terminal interface, our API widens the definition to cover terminals that do not use field positioning.

The API operates much like a person sitting at the terminal console. A pseudo-screen is created (which does not display on the COMMAND/Post workstation monitor), and the REXX program uses functions defined in the API to interact with this screen. Some functions allow the entire screen to be captured as an array and transferred back into a REXX variable for processing. Other functions allow a portion of a screen to be captured, or in the case of a terminal supporting fields, a field to be captured. Other functions allow data to be sent to the screen as if was coming from the keyboard. There are a number functions dedicated to positioning the cursor and searching the screen, or fields, for text. A few give status information, including the height and width of the screen.

The API also allows for an interrupt driven capability for situations where a simpler set of calls cannot handle the exchange. The REXX program waits until new data arrives on the pseudo-screen and then is released so it can make additional calls to observe how the screen has changed.

All calls to the API interface are made through the **ao_targetComm** function described above. The "fd" argument contains the handle for the particular target system involved, and the "function" argument contains the number of the API function that will be used. The "data", "length", and "pos" arguments definitions vary based on the function call. In general, "data" is data being read or written to the pseudo screen. "Length" is the length of that data string. And "pos" is the position involved when data is written or read. The API views the screen as an array of characters (row one, followed by row two, etc.) and the position is a value pointing to that array.

Using the API model to connect with the target system has a number of advantages. First, as noted above, it removes the need for an interrupt type interface when only simple communications are involved. When only a single thread of communication is involved it is relatively easy to create a loop in REXX to read the screen, write to it, read the screen again, identify what has changed and then act on the new data.

Another advantage is the interface allows some measure of emulation independence. That is, a REXX script can be designed that will operate with either a EM running a VT-320 interface or a IBM3151. Then the only change needed between the two would be in the definition of the pathway during the configuration step external to REXX. The user would define the path as using a VT-100 API interface instead of an IBM3151.

Despite the interface, there are some restrictions on how transparently a REXX program can be written. Some terminals support the use of "fields." A REXX program that made use of the API field related functions to interact with a Tandem 6539 would not work with a VT-100, because it does not support fields.

1.5. Parameters

The REXX interpreter was also augmented to accept command line arguments that could be passed in the the REXX programs as parameters. The command line to the left of a "--" remained the standard uni-REXX command line. The part to the right represented parameters passed to the REXX program. Argument flags (items starting with a "-") became variable names in the program filled with the values that followed them. The following command line:

```
 ncrx -- -customerName "Fred"
```

would cause the REXX program to start execution with a variable called "customerName" initialized to the value of "Fred". This allowed the triggers to pass useful information to a REXX program. Standard information passed included the number of alerts that caused the trigger to fire and the identification numbers of those alerts.

1.6. Database Interface

We also wanted the REXX auto-operations programs to be able to access the COMMAND/Post database so they could create, query, update, and delete the alerts the system maintained.

COMMAND/Post uses a relational database that is divided over two dataserver programs using the Sybase "open server" model. The primary database is accessed through the standard Sybase dataserver. Temporary high activity tables are assigned to the "Event Handler" server: a memory resident server of our own design.

Access to either server is via REXX's ADDRESS instruction. Addressing NCDB connects the REXX program to the primary sybase dataserver, using "ALERTS" connects it to the Event Handler. To interact with either the programmer need only code an SQL command, or use a stored procedure (a Sybase term for SQL routines maintained in the dataserver) in the address command. The success of the command can be evaluated by looking at the special REXX variable "sqlCode".

While, for the most part, addressing the dataservers via this command is straight-forward, a few SQL commands represent a problem. For example, "SELECT *" command may return row after row of data from the table, each row with many individual data items. Each item can be of a variety of data types. Here's where REXX's ability to create variables on the fly and have variables types change make it an excellent choice of our application. As a data item is returned, let say the time field for particular alert, a REXX variable named "TIME" is created, if it does already exist. It is filled with the text representation of the time. The same thing for integers or for character strings (which the database can store in several varieties) The programmer need not immediately be concerned with making sure the variable type matches what's coming back from the database.

Multiple rows are handled by returning one row at a time and having a special "fetch" command. The program can use to indicate that it is finished with the current row and is ready to receive the next. Values for the new row are written over and into the same variables used by the last row. If all rows are exhausted the "sqlCode" variable returns an error value (non-zero). If there is no need for additional pending row a special "cancel" command can be used to drop them.

A typical code fragment to print the item "alertId" from the "ActiveAlert" table might be:

```
address NCDB "select alertId from activeAlerts"
if(sqlCode = 0)then
  do forever
    address NCDB "fetch"
    if(sqlCode <> 0) then
      leave
    else
      say alertId
  end
end
```

One limitation created by this architecture is that all values returned by a "select" statement must have some associated name for creation of the variable. This means that an SQL statement that used some function (like SUM) to create a value that would not have a name associated with it must be written in such a way that it is forced into a variable name. For example:

```
select total = sum(occurrences) from activeAlerts
```

instead of

```
select sum(occurrences) from activeAlerts
```

To make common operations, like creating an alert, (which would normally require multiple table inserts) easier, a number of stored procedure are included in the database. This means that typically only a single "address" clause is needed for even a fairly complex database operation.

1.7. Other Uses of REXX in the Product.

One of the bonuses of implementing REXX as our auto-operations language was that we could use it for general programming. We have a large library of scripts (mostly written in Bourne or C shell) used for installation and maintenance of the product. When these scripts interacted with the database they had to first create a second file that would act as input to the Sybase's Interactive SQL program (ISQL). Then they had to start ISQL directing the second file to the standard input, and finally monitor the standard output for errors. This convoluted approach made the script hard to read. It also made isolating a particular SQL statement that failed difficult since the script was not feeding the commands to ISQL one by one.

Our REXX, with the ability to address the server through the ADDRESS instruction has simplified this problem. Since the REXX can address the database directly it is easier to write and test the script/program. Errors are also easier to detect and handle.

1.8. Results

Over 100 sites now use COMMAND/Post with the automated operations facility. A majority of the customers involved have decided to write their own custom auto-operations scripts which lessens the load on our support staff. We are pleased with our decision to use REXX for auto-operations.

1.9. Bibliography

COMMAND/Post How to Guide, Release 3.0, Boole and Babbage Network Systems, San Jose California, 1993.

Programmers Guide: High Level Language Application Program Interface, IBM Corporation, Austin Texas, 1987.

A REXX CookBook for COMMAND/Post, Boole and Babbage Network Systems, Mt. Laurel New Jersey, 1993.

uni-REXX Reference Manual, The Workstation Group, Rosemont Illinois, 1991.

1.10. Glossary

ALFE - Alert Logic Filter Editor - The facility in COMMAND/Post used to construct a "filter".

API - Application Program Interface - The interface that allows REXX and C based programs to interact with COMMAND/Post emulations.

Dataserver - A process that manages and provides access to a database.

EM - Element Managers - Network Control and Monitoring systems that manage a domain of network elements like modems, communication lines, etc.

Event Handler - COMMAND/Post primary memory resident dataserver.

EXEC - A REXX program for COMMAND/Post that is part of the auto-operations subsystem.

IBM 3151 - IBM async terminal.

IBM 3270 - IBM sync terminal.

Filter - A program in COMMAND/Post which parses a stream of data, usually from some external source, looking for messages. When a message is found the filter created an alert for the COMMAND/Post database.

Open Server - A database design that allows dataservers from multiple vendors to operate together.

RDBS - Relational Database System - A database designed to adhere to relational principles.

Shell - A Unix command interpreter.

Shell Script - A program that is interpreted by a Unix C or Bourne shell.

SQL - A 3rd generation database manipulation language.

Sybase - A RDBS product.

Tandem 6539 - Tandem async terminal

Unix - Operating System on which COMMAND/Post runs.

VT-100 - VT-320 - DEC async terminals

1.11. APPENDIX: Sample COMMAND/Post REXX program.

```
#!/usr/nc/bin/ncrx -s
/*************************************************************
 * REXX program to perform simple paging.  The OSI Severity
 * of the first triggering alert will be sent to the pager.
 *
 *      This program does the following:
 *
 *       >Connects to a tool called "pagerModem" which is
 *              assumed to be an "AT" mode modem
 *       >Initializes the modem for word responses.
 *       >Sends touchtone dial sequence which leaves the modem
 *              in command mode
 *       >Sends "PIN" followed by "#"
 *       >Looks at the first underlying alert passed in and gets
 *              the OSISeverity value from the data base.
 *       >Sends OSISeverity followed by "#" out to the pager.
 *       >Waits 5 seconds for repeat
 *       >Sends final "#" to force posting of message
 *       >Does a "hangup"
 *       >Disconnects from modem.
 *
 *       The following variables should be passed in from the trigger:
 *          pagerModem - Access path of the modem.
 *          PIN - Users pin number.
 *          underlyingAlerts (optional)
 *          alertCount (option)
 *
 *       If no alertCount or underlying alerts are available the value
 *       "9999" will be sent to the pager.
 *
 *       Note: Triggering Alerts must be forwarded to the database so
 *       that the Severity can be obtained.
 *
 *************************************************************/

/*  Get the OSI severity from the database */

if alertCount <> "ALERTCOUNT" then do
  cnt = 1
  alertId = underlyingAlerts.cnt
  address NCDB "select OSISeverity from alerts where ",
  alertId" = alertId"
  if(sqlCode <> 0)then do
    say "Could not get Severity from database"
    exit
  end

  address NCDB "fetch"
  message = OSISeverity
end
else
```

```
    message = "9999"

/*  Connect to the modem */

nl    = "0a"X

path = ao_targetConnect(pagerModem)
if rc \= AON_noError then do
    say "failed to make connection to" pager
    exit
end

/* Get the dimensions of the emulation screen and
   calculate the Presentation Space size */

string = "A"
r = ao_targetComm(path, 22, string, 1, 0)
columns = delstr(delstr(string, 16), 0, 13)
rows = delstr(delstr(string, 14), 0, 11)
PSsize = columns * rows

/* Start the conversation with the modem */

if modemSend(path, "ATV1"nl,"OK") = 1 then
    exit

/* Dial the service */

if modemSend(path, "ATDT9,18007597243@;"nl, "CONNECT") = 1 then
    exit

/* Send the PIN */

if modemSend(path, "ATDT"PIN"#;"nl, "OK") = 1 then
    exit

/* Send the message (OSI Severity) */

if modemSend(path, "ATDT"message"#;"nl, "OK") = 1 then
    exit

address UNIX "sleep 5"

/* Clean up */

if modemSend(path, "ATDT#;nl", "OK") = 1 then
    exit

if modemSend(path, "ATH"nl, "OK") = 1 then
    exit

/* Disconnect from the path */

r = ao_targetClose(path)
```

```
exit

/************************************************************************
 * modemSend
 *
 *   sends the contents of string to modem and checks that the
 *   response from the modem contains the contents of pattern
 *
 ***********************************************************************/

modemSend:
    arg path, string, pattern

    len = length(string)

    /*  Find the cursors current location on the screen */

    r = ao_targetComm(path, 7, 0, cursor, 0)
    if rc \= AON_noError then do
     say "Can't find cursor: rc =" rc
     return 1
    end

    /*  Send the command to the modem */

    r = ao_targetComm(path, 15, string, len, 0)
    if rc \= AON_noError then do
     say "Send failed: rc = "rc
     return 1
    end

    /* Loop ten times waiting each time 2 seconds for a response. */

    do 10
     address UNIX "sleep 2"

     /* Look for response following cursor position */

     r = ao_targetComm(path, 8, string, (PSsize - cursor), cursor)
     if rc \= AON_noError then do
         say "Send failed on check: rc =" rc
         return 1
     end

     /* Search for a the expected pattern */

     if pos(pattern, string) \= 0 then do
         return 0
     end
    end

    say "did not find" pattern
    return 1
```