1-1-1986

View 3: A Programming Environment for Distributed Programming

Ann Kratzer
Dartmouth College

A PROGRAMMING ENVIRONMENT FOR DISTRIBUTED PROGRAMMING

Ann Kratzer

Technical Report PCS-TR86-120

View 3

A Programming Environment for Distributed Programming

Ann Kratzer

Dartmouth College

Abstract

View 3 is an experimental programming environment to support the development and use of distributed programs. It builds upon three major concepts. First, distributed programs and distributed processes are basic objects. Second, the port mechanism allows a process to exchange information with another process, a file or an I/O device without concern for the type of the object on the other end of the port. Third, test and use of distributed programs are facilitated by the user interface program screen format that allows the user to control both the format and contents of the physical terminal.

0) Introduction

Much attention is focused on distributed computer systems. In part this interest stems from the many advantages that these systems offer; research has provided the technology and methodologies needed to permit commercial production of such systems. However, the full potential of distributed systems is realized only with distributed software to run on them. Although it is possible to develop, test and run distributed software, current programming environments do not make this task easy. View 3 is a programming environment designed to facilitate distributed programming.

View 3 builds upon and brings together advances from three areas of past and present research. First, work on programming environments[1-4] has developed the tools needed to develop and use sequential programs. Second, research in the area of programming languages for distributed programming[3,5,6] provides us with the building blocks needed to write programs in which function is logically distributed between communicating sequential processes that run concurrently. Third, the development of network architectures and network operating systems[7-12] demonstrates how the

resources of a physically distributed system can be shared.

In Section 1, an overview of View 3 is given. The View 3 kernel functions and their implementation are discussed in Sections 2 and 3. In Sections 4 and 5, we present the user interface to View 3. Finally, the results of this paper are summarized in Section 6.

1) Overview of View 3

The concepts of distributed program and distributed process are central to View 3. A distributed program is comprised of one or more sequential programs, that are to run at the same time and communicate with one another by exchanging messages. Logically, a distributed program, as a whole, performs a single computation. The function needed to implement the computation is partitioned between the constituent sequential programs. In this sense, the sequential programs are not separate, but rather are a part of the whole that is the distributed program. When executed, a distributed program becomes a distributed process.

Distributed programming has special needs that are not typically supported by conventional programming environments: First, language constructs are needed that allow a program to communicate with external objects, which may be files, I/O devices or other programs, and to synchronize with events. Second, a distributed programming environment should support distributed programs and processes as objects. Third, when running or testing a distributed program, the user or programmer must have control over which process's output appears on the physical terminal.

The View 3 project has as its goal the implementation of a programming environment to support the development and use of programs that are logically distributed but not physically distributed. The system runs on a Digital Equipment Corporation VAX 11/750 running UNIX Berkeley release 4.113. Although UNIX provides the building blocks from which View 3 is built, it does not provide the language constructs needed by distributed programs, does not support distributed programs and distributed processes as objects, and does not implement the terminal control facilities in a form appropriate to our needs. We operate under one major constraint: Because the system that we are using must provide production quality computing, modifications to the UNIX kernel can not be made.

View 3 is structured as a two level hierarchy of function. The lowest, kernel level implements the language constructs needed by distributed programs and objects needed for a distributed programming environment. The highest

level creates the user interface to View 3 that provides the control over terminal output and the command interpreter.

2) The Kernel

The kernel maintains the View 3 objects and implements the operations that can be performed on them. There are five different kinds of objects: distributed processes, processes, files, I/O devices and ports. There are three categories of operations:

1) create and destroy which are legal on all objects except I/O devices,

2) operations defined only for ports: connect, disconnect, send and receive and

3) the synchronization primitives, wait and status.

2.0) Ports

A port provides the means by which a process can communicate with an external object. The external object must be either another process port, a file or an I/O device. Using a port, a process may either send information to or receive information from the object on the other end of the port. Within a process, ports are referred to by number.

When operating on a port, the port is referred to by its number irrespective of the object on the other end of the port. Thus, a process is oblivious as to whether the I/O is to a process port or file; also, a process need not know the number of the process port wired to the other end of one of its own ports. This other-end object invisibility enables the programmer to use the same programs in constructing different distributed programs from a given set of sequential programs. In addition, the programmer can wire process ports to files or to the screen formatting utility discussed in Section 4 while debugging; when installed, the port may be wired to the appropriate I/O device. Thus, no change to a program is needed between the test and debug phase of program development, and the actual installation of the finished system.

The operations on ports are relatively standard. A process may create a port; at creation time, the name of the object to be wired to the port must be specified. An existing port may be connected to or disconnect from the object on the other end of the port; these operations are generalizations of file open and close. At connect time, the format used when sending information through the port must be specified. The options are unformatted and time stamped;

these affect the format of the information stream being sent.

Information may be received or sent through a connected port. When the port is unformatted, information is passed through the port as sent. A time stamped port automatically transfers information in units of messages; a message is a character string passed as the data to a call of send. When delivered, information from a time stampped port has the following format:

message length,

time stamp (as returned from the UNIX system call time13) and

data.

Finally, a port may be destroyed; port destruction implies disconnect.

2.1) Operations on Distributed Processes and Processes

The operations on distributed processes and processes are among the most important in View 3. These operations permit the creation and destruction of distributed processes. When applied to a process within a distributed process, the create and destroy operations allow for the test of a distributed program's ability to gracefully respond to the dynamic death and birth of its constituent processes.

2.1.0) Operations on Distributed Processes

The create operation when applied to a distributed process converts a distributed program into a distributed process. The argument is a description of the distributed process, which we call a template. The template contains:

the name of the distributed process and

templates (descriptors) for each process.

Subsequently, the distributed process can be referred to by the name given when it was created. The processes are described by a process template which includes:

the process name,

the name of the process's executable file,

the number of arguments to the process,

the actual arguments to the process and

templates for the process's ports.

The process is referred to later using the distributed process name and the name given the process at the time of its creation. A port template contains:

the port number,

the type of the object on the other end of the port and

either the file name, I/O device name or the process port name of the object on the other end of the port.

The object on the other end of the port must be either a file, an I/O device or a process port. Process ports are named by giving:

the distributed process name,

the process name and

the port number on the named process.

There are potentially two 'returns' from the create: one to the creator and one to each process created. The creator is returned the outcome of the create: success or failure. The created processes begin execution being passed:

the total number of arguments,

the file name executed,

the distributed process name of this process,

the process's name,

the number of ports on this process and

the arguments.

The ports given at create time in the process template exist, i.e. they do not need to be created. The process must connect a port before sending or receiving through the port. The identity of the object on the other end of the port is hidden from the process.

The destroy operation when applied to a distributed process causes all the currently existing processes in the distributed process to be destroyed. The argument to destroy is the name of the distributed process; the outcome, success or failure, is returned. Only the creator may destroy a distributed process.

2.1.1) Process Create and Destroy

The create and destroy process operations cause a process within a distributed process to be either created or destroyed. They permit the programmer to test a distributed program's ability to respond to the crash and restart of processes.

The create operation adds a process to an existing distributed process. The arguments to process create are:

the distributed process name and

a process template for the new process.

The outcome, success or failure, is returned to the process issuing the create. The new process begins execution as it would after a distributed process create. Only the creator of the distributed process or a process within the distributed process may perform a create process.

The destroy operation causes are process to be destroyed. The arguments are:

the distributed process name and

the process name.

The return to the caller indicates success or failure. The destroy process operation is limited to the creator of the distributed process and to the members of a distributed process. Note that a process may destroy itself; in this case, there is no return from the destroy.

2.2) The Synchronization Primitives

The primitives wait and status enable a process to synchronize with an externally generated event.

Wait suspends execution of the process until the event occurs; if the event has already occurred at the time the wait is invoked, the return is immediate. The process resumes at the point of the wait. A process may wait for the following events:

the destruction or crash of a distributed process,

there to be at least a given number of bytes in an inbound port,

a pending connect attempt by some other process,

the creation of a new process in this process's distributed process,

there to be no more than a given number of bytes in an outbound port and

the destruction or crash of a process within the distributed process.

A process may use the status primitive to obtain the current status of an object. This enables a process to poll an event. A process may obtain the status of:

a distributed process (its template,)

an inbound port (the number of bytes in the port,)

a pending connection (the name of the other end,)

an outbound port (the number of bytes in the port) or

a process (its template.)

The synchronization capabilities are not general; extensions are planned. For instance, it is not possible to implement synchronous message passing using this set of primitives. Also, the mechanism does not support nondeterminism[5,14]; however, a process can itself act nondeterministically based on the status of other objects.

3) Implementation of the Kernel Functions

The restriction that we may not modify the UNIX kernel has forced us to implement the View 3 objects and operations as a non-privileged guest layer[15]. The kernel consists of a set of subroutines that reside in the View 3 library. The appropriate subroutines are linked to a View 3 user program; thus, kernel functions are performed at the UNIX user process level.

The kernel has two major jobs: First, the kernel maps the View 3 system calls into their UNIX equivalents. Second, it is necessary to maintain the information needed to support distributed processes, ports and events and to implement the View 3 operations. Information on a distributed process must be static and sharable by all the processes within a distributed process.

The information on a distributed process is kept in a dynamic template. The dynamic template is very similar to a distributed program template; additional information is kept that records process status, port status, and saves information needed for UNIX system calls such as file descriptors, mpx file names etc. Because the template must be accessed by all the View 3 kernel subroutines linked into each

process within the distributed process, the template is kept in a file. Critical sections on the template file arise; locking and unlocking of the template file using lock files guarantee mutual exclusion.

Implementation of the kernel as a non-privileged guest layer has disadvantages. The functions being performed are those of process management and synchronization; when running on a single machine, these are best implemented at the level of the operating system kernel. In addition, there is overhead and inefficiency incurred because the template is kept in a file. However, this form of implementation does have one advantage: The implementation of the kernel is distributed. Thus, we are gaining experience on how to build a distributed kernel; this should aid us in building a physically distributed View 3.

4) The View 3 User Interface

The user interface to View 3 performs two major functions. The screen format utility enables the user to control both the content and format of the display of the physical terminal. The monitor is the View 3 command interpreter.

4.0) Screen Format

Screen format is a limited virtual terminal facility[16] for CRT terminals with features that support the test of distributed programs. The bottom line of the screen is for input of commands to screen format; the rest of the screen can be used for input to and output from processes. There are a total of six screen format commands.

The toggle command changes the number of windows displayed in the process I/O area of the screen. There may be either one or two process I/O windows. When the screen currently contains one window, toggle reformats the screen into two windows, one above the other; this creates a blank window. If the screen contains two windows, then toggle returns the screen to the single window format; the window whose contents are to be kept is given as the argument.

The place command requires two arguments: the name of a process port and, when in the two window format, the window number. Output from the named port is displayed in the appropriate window; the current contents of the window (if any) are lost. The user may provide input to a process port using the enter command. Enter, which takes as argument a window number, moves the cursor into the window at its lowest line; input may now be entered to this port. A window is left using the leave command which consists of

hitting the escape button on the terminal.

Two commands are useful when testing distributed programs. When in the two window format, the command synch causes the windows to appear side by side. The two ports displayed must be time stamped ports. Output from the two ports is aligned based on their time stamps. The desynch command causes the two window to be desynchronized. The screen reverts to the two window format with one window above the other.

4.1) The Monitor

The monitor is the View 3 command interpreter. It also keeps track of what distributed processes the user has running. There are four monitor commands.

Two commands provide the user with status information. The running command prints out a list of the names of all the distributed processes that the user has running. View displays status information on a given distributed process: process names and how the processes are wired together.

The other two commands control the creation and destruction of processes and distributed processes. The kill command causes the named process or distributed process to be destroyed. The run command starts the execution of a distributed process or adds a process to an existing distributed process. To initiate a distributed process, the user enters the name of a file which must contain a distributed program template. When running a process, the arguments are a distributed process name (to which the new process is added) and the name of a file containing the process's template.

A special form of the run command permits UNIX processes to be run under View 3. The user inputs a shell command line which is then passed to a View 3 version of the UNIX shell. Before creating the UNIX processes, the View 3 shell ensures that I/O is not directed to the physical terminal. If necessary, the shell command line is encased in View 3 filter programs[1] that connect to screen format to provide the UNIX process with I/O.

4.2) Access to View 3

View 3 may be entered at login time using the usual UNIX mechanism[1] for using a program as the shell. Alternately, the command view3 can be used. In either case, the result is to create a View 3 interface which is shown in Figure 1. The interface is the distributed program consisting of screen format and the monitor.

5) The Template Editor

The template editor is the View 3 utility used to create template files. It works in two modes: A line mode is used with regular terminals; the user is prompted for commands and input in a line oriented fashion. A graphics mode is also available which provides a more natural way to create templates.

The graphics mode runs on the Digital Equipment Corporation Gigi terminal17. The Gigi supports medium resolution graphics in a limited set of colors; it also provides a cursor that can be moved up, down, left or right. The screen usage when in graphics mode is shown in Figure 2. The right hand side of the screen is the graphics work area in which the template is displayed. The left hand side of the screen consists of two windows: In the upper window appears the name of the template file. The lower window is used for menus and text input. Currently, the graphics mode can only be used to add to a template; editing of templates must be done in line mode.

6) Summary

View 3 is a programming environment designed to facilitate the development, test and use of distributed programs. There are three concepts that are central to View 3. The most important is that of distributed program and distributed process which are basic objects. The invisibility of the object on the other end of a port permits the construction of many distributed programs from the same set of sequential programs. We have borrowed this notion from previous programming environments and extended it somewhat. Finally, test and use of distributed programs would not be possible without the screen format function. In this area, we build upon previous research extending the virtual terminal concept to include the notion of synchronized windows.

View 3 is an ongoing activity. We are currently working to complete the system as outlined in this paper. Already a number of extensions have been suggested: design and implementation of a more general monitor control language, enhancement of screen format to allow movement of text between windows and to give the user control over the number, size and placement of windows, extension of the port concept to permit transmission of information as defined by the program in both a syntactic and semantic sense, allowing for the transfer of out-of-band information through ports, upgrade of the graphics mode of the template editor, generalization of the synchronization mechanism possibly to include nondeterminism and implementation of a physically

distributed View 3.

7) Acknowledgements

I would like to thank Bob King for his many suggestions, comments and questions; his input was invaluable. Thanks also go to Jon Allen, Bruce Culbertson, Anne Hathaway and Kit Shum for their many contributions. The graphics mode of the template editor is entirely the work of Jonathan Groisser. I would like to thank Steven Tahmosh for his input which included the notion of screen synchronization. Finally, I must thank my many friends at the Kiewit Computational Center, particularly Phil Koch, for their support.

Bibliography

1) "The UNIX Time-Sharing System," D. M. Ritchie, K. Thompson, The Bell System Technical Journal, 57, 6, Part 2 (July-August 1978,), p 1905-1929

2) "The Structure of THE Multiprogramming System," E. W. Dijkstra, CACM, 11, 5, (May 1968,) p 341-346

3) Ada: Programming in the 80's, Computer, 14, 6 (June 1981)

4) "Thoth, a Portable Real-Time Operating System," David R. Cheriton, Michael A. Malcolm, Lawrence S. Melen, Gary R. Sager, CACM, 22, 2, (February 1979,) p 105-115

5) C. A. R. Hoare, "Communicating Sequential Processes," CACM, 21, 8 (August 1978,) p 666-677

6) Gregory R. Andrews, "Synchronizing Resources," TOPLS, 3, 4 (October 1981,) p 405-430

7) John K. Ousterhout, Donald A. Scelza, Pradeep S. Sindhu, "Medusa: An Experiment in Distribute Operating System Structure," CACM, 23, 2 (February 1980,) p 92-105

8) "The Roscoe Distributed Operating System," M. H. Solomon, R. A. Finkel, Proc. ACM SOSP7, December 1979, p 108-114

9) W. D. Sincoskie, D. J. Farber, "SODS/OS: A Distributed Operating System for the IBM Series/1," Operating Systems Review, 14, 3 (July 1980,) p 46-54

10) Andrew S. Tanenbaum, Sape Mullender, "An Overview of the Amoeba Distributed Operating System," Operating Systems Review, 15, 3 (July 1981,) p 51-64

11) "The Cambridge Model Distributed System," M. V. Wilkes, R. M. Needham, ACM SIGOPS, 14, 1, (January 1980,) p 21-29

12) Sandra A. Mamrak, Dennis Leinbaugh, "A Progress Report on the Desperanto Research Project," Operating Systems Review, 17, 1 (January 1982,) p 17-29

13) UNIX Programmer's Manual, Volumes 1 and 2, Seventh Edition, Virtual VAX-11 Version, June 1981, (for the Berkeley 4.1 BSD Version of UNIX)

14) "Guarded Commands, Nondeterminacy and Formal Derivation of Programs," Edsger W. Dijkstra, CACM, 10, 8 (August 1975,) p 453-457

15) "Guest Layering Distributed Processing Support on Local Operating Systems," S. A. Mamrak, P. Maurath, J. Gomez, S. Janardan, C. Nicholas, The Third International Conference on Distributed Computing Systems, p 854-859

16) Andrew Tanenbaum, Computer Networks, Prentice Hall, 1981

17) GIGI, VK 100, Terminal Installations and Owner's Manual, Digital Equipment Corporation, EK-VK100-IN-001

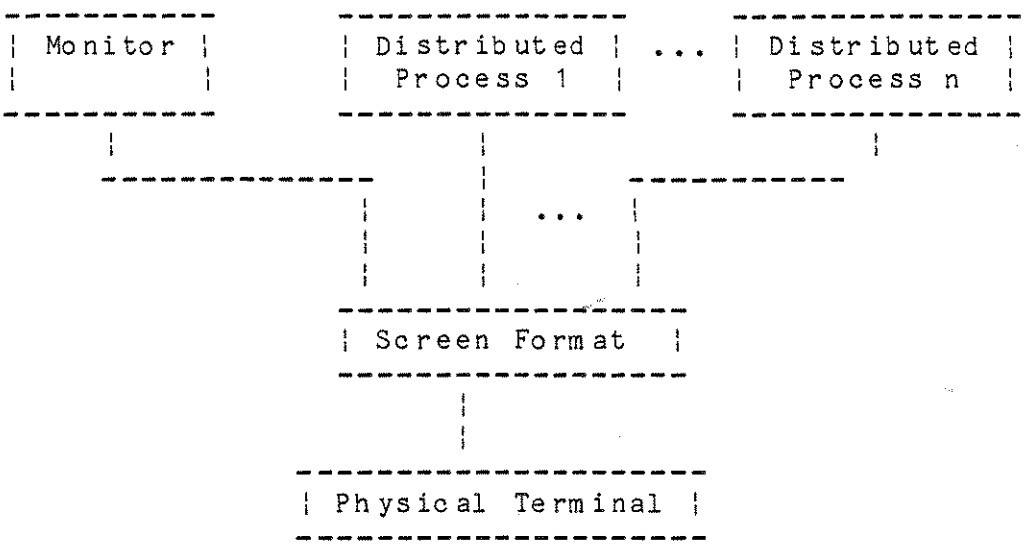

Figure 1: View 3 User Interface Showing User Distributed Processes

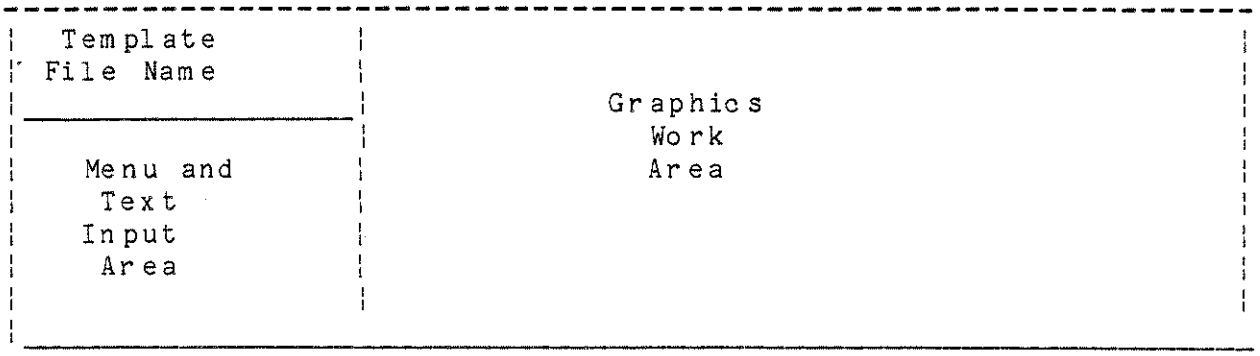

Figure 2: Template Editor Screen Usage