Dixie Language and Interpreter Issues

R. Stockton Gaines
U. S. C. Information Sciences Institute
4674 Admiralty Way
Marina del Rey CA 90292
gaines@isi.edu

Abstract

Dixie (Distributed Internet Execution Environment) provides a base for sending programs called Dixie applications to Internet sites for execution. It provides the features generally found in operating systems, such as a file system, multiprocessing, interprocess communications, etc., and in addition capabilities to permit Dixie applications to interact with resources at the local site. Security is of first importance; it must not be possible for a Dixie application to have an undesired effect on the local system. This paper explains the Dixie concept and discusses language and execution issues. The languages understood by Dixie, at least initially, will fall in the class of Very High Level Languages, not the least because these languages will support the security requirements of Dixie, as well as the command language requirements. Dixie complements these languages, and provides uniform platform independent of the local hardware and operating systems to support Dixie application programs.

Introduction

Dixie (Distributed Internet Execution Environment) is a virtual operating system and program execution environment that is portable. Once installed on a system connected to the Internet (an Internet host, or simply "host", here), it can execute programs in the languages it supports. Dixie, therefore, provides a means of sending a program to other Internet hosts for execution. Programs which execute in the Dixie environment are called Dixie applications. They are an instantiation of the concept of knowbots or intelligent agents that can travel through the Internet carrying out useful actions.

Dixie complements the recent work on very high level programming languages by providing an interface that is consistent across systems, is secure in that programs executing on Dixie are harmless to the local host system, and provides full operating system functionality. A program written in a language embedded in Dixie will see the same operating system interface anywhere in the Internet, independent of either the operating system or the compilers of the host system itself.

Dixie solves another problem, that of software portability. A Dixie application can be run on any host with Dixie installed, without need for changes to be compatible with the host's underlying operating system, compilers or hardware. Dixie therefore is a vehicle for software distribution. Furthermore, because Dixie installations are themselves accessible through the Internet, a natural means of remote maintenance of Dixie applications (as well as Dixie itself) is available.

Whereas portable servers such as World Wide Web (WWW) are primarily of interest on hosts that act as servers at Internet sites, Dixie will also be useful when installed on workstations. Dixie will include a GUI package so that a Dixie application running on a workstation can be interactive with the workstation user.

Dixie includes a full operating system model, called the Dixie Host Interface (DHI). The main features are: processes, including multiple processes in execution, shared memory between processes, interprocess communications, and process scheduling and swapping; a complete file system interface; and device and service interfaces to the host system on which it is installed.

The first priority in designing and implementing Dixie is that it provide a secure execution environment.

There are two types of security of concern, both important. First, a Dixie application must not be able to harm the host system on which it runs in any way. Second, communications to and from both Dixie applications and the DHI must be authenticated, and be reliable (that is, the message received must be verifiably the message sent). These security considerations will not be discussed further here, but dominate the design of Dixie.

Another system that offers the ability to send a program to a remote site for execution is Safe-Tcl [1, 7, 8]. Safe-Tcl is the most recent version of an active mail system (termed enabled mail in Safe-Tcl papers). Enabled mail is mail that when read by the receiver, executes as a program. Dixie takes a much broader view of the issues and requirements for executing programs at remote Internet sites, but many of the security issues are similar. A good discussion can be found in the Safe-Tcl references.

The Dixie Host Interface will support multiple program execution environments. Initially these will be based on interpreters. Again, the goal is to provide a host-independent environment that supports useful languages.

Dixie will combine three important existing components: the Prospero file system [5], the Tk [6] GUI package, and interpreters for several languages. At the time of writing it is expected that Python [9] will be of great interest. This language is an unusual combination of elegance, simplicity and power, with a number of features that are particularly suitable for Dixie (especially its form of modules). Tcl [6], already in widespread use, is another powerful and important language that will be integrated into Dixie. Other languages of interest are Perl [10], REXX [3] and, when and if it becomes available, Telescript from General Magic. All of these languages are implemented as interpreters, and are suitable both as command languages and programming languages. Since in many cases a Dixie application will execute on a remote site from the invoker of the program, and the invoker will not be connected in a session with the application, the ability of the application to generate commands to its operating system (DHI) as well as lower level operating system calls is an important virtue of all these languages.

One motivation for Dixie is to provide a method that permits programmatic access to local resources through the Internet in a safe way. For example, sites that maintain databases may wish to make the information in the database accessible without exporting the entire database. An SQL interface will be incorporated in DHI through which accesses to local databases can be defined and controlled by the host owner. For example, if various Departments of Motor Vehicles are interested in making information about automobiles and accidents available, but prohibiting access to any personal information about drivers or automobile owners, appropriate views of the database can be defined that will not support the retrieval of such prohibited information. A Dixie application running on such a host can issue SQL commands against these views, but cannot otherwise access the database.

The file system interface will be based on Prospero [5]. It is already being used extensively, and has, for example, been used as the basis for the archie server. Prospero defines a mapped view of the underlying file system. The view that is presented through Prospero consists of a set of directories and files that may be different from the actual structure of the file system of the host computer. The mapping will be definable by the owner of the host system. (Prospero also includes the ability to make visible non-local files that reside on other systems. This, too, may be valuable for Dixie).

An important aspect of Prospero is that attributes can be associated with each Prospero visible file and directory. These attributes can include access methods. For example, a read access method can be defined for each file. When the file is accessed, the routine specified for the file is invoked, rather than simply reading the file in the normal manner provided by the host file system. The attributes can include additional security mechanisms. One example would be the association of an access control list with a file, designating on a per file basis the rights of specific authenticated individuals. Attributes associated with directories would include the right to create a file, and to designate its type. For example, it would be possible, and useful, to restrict the creation of files to files that can be read but not executed. Dixie, through the use of Prospero, will be able to insure that no Dixie application can install a file in the local file system that is executable, which will prevent many well known attacks on systems. The power to control exactly how the Dixie applications can interact with the local file system, including which portion of it is visible, and through the use of file and directory attributes place additional limitations on the access to the file system and the ways in which files are created, named or renamed and modified leads to a high degree of security.

Prospero provides the ability to map a single file into an entire file system, from the viewpoint of a Dixie application. Prospero can also map a disk partition as a file system. This will isolate it completely from the host file system, if that is desirable. As can be seen from these examples, Prospero provides complete flexibility in providing persistent storage through a file system interfaced for Dixie and Dixie applications, with the ability to expose those parts of the host file system that the host owner desires, while restricting all other accesses.

In general, restrictions on the use of the host system's resources will be implemented within DHI. Since DHI provides all support for Dixie applications, which cannot invoke the host operating system directly, restrictions on the language itself will be minimized.

For reasons of efficiency or functionality, it may be desirable that a Dixie application be able to make calls on routines that are compiled to run directly on the host computer. For example, if Dixie had been available and in widespread use, it could have been used as the basis for finding the largest prime number using many computers throughout the world. The heart of this distributed application was a relatively simple C program. All of the communications and coordination parts of the application could have been handled through a Dixie program for each host, since the computation requirements for these were not great. But it would have been necessary to provide an interface to the C subroutine from a Dixie application.

The main issue here is security. The host owner must be able to trust the C program. The host owner could trust the program if written locally, or obtained from a reliable source. Trust could also be based on an inspection of the program's source code, for programs that are simple enough. In the example just given, this could be straightforward. The program should inspect its inputs to insure that they are valid, should not make any system calls, and should communicate with the DHI in a straightforward way, such as accepting a single value as an argument and returning a single value. The routine would need to be registered by the host owner as callable through the DHI in order to be accessible to a Dixie application. To deal with more than very simple cases may be a research question.

Language and Interpreter Issues

The philosophy that motivates Dixie is that there is a clear distinction between an operating system and a programming language. Far too much of the operating system tends to get built into programming languages, limiting flexibility and applicability. This philosophy suggests that the abstractions presented to the programmer should be at a high enough level that there is freedom to do what makes sense during code generation, program execution and in the operating systems to deal with issues of memory management, process structuring and scheduling, etc.

A process has an internal behavior and an external behavior. The programming language provides mechanisms for defining objects that populate the internal environment and specifying actions on those objects. A language is also needed to describe the external actions of a process, but that language is, according to the philosophy being espoused here, not part of the programming language. Rather it is a language invoked through the programming language by calls to routines that cause external actions, and by emitting statements in a language that is understood external to the process. A great virtue of many very high level languages is that they provide good tools for generating these statements for external consumption.

An example from ADA may help to illustrate the point. ADA includes as language constructs "fork" and "join". Fork and join are process management actions. By including these as primitives, ADA was forced to add a lot more baggage within the language to define and manage what amounts to pseudo processes. These features in turn impose restrictions on the operating system, or else result in a complicated run time package to support ADA. If fork and join are calls on routines that are supplied separately from the programming language, they can have a semantics suitable to both the operating system and hardware environment in which the program will execute, and can be optimized for the needs of different types of applications.

The separation of concepts between the programming environment and the supporting operating system environment of Dixie leads to a smaller set of requirements for the languages that provide the execution environment for Dixie applications. The required functionality, to the extent possible, will be provided by a set of run time callable routines that are common to all the programming environments. This has the additional virtue that Dixie can evolve without the need to change all the language interpreters when there is a change in the DHI.

Since the Dixie Host Interface acts as the operating system for a set of Dixie processes that are executing Dixie applications, it must provide for the synchronization of the activities of these processes. A design objective of Dixie is to develop a set of synchronization and coordination tools that will support both processes running on the same machine and processes that are distributed among multiple machines. Semaphores and other synchronization mechanisms will be built into the DHI. Such tools are not ordinarily included in operating systems, but there are a couple of advantages. First, they can be made simple and efficient. In addition, the scheduling and swapping policies for Dixie processes can be aware of process synchronization activities, also improving efficiency.

An issue that has not received much attention from the programming language community is how a programmer can view and act on a program from within the program. At least two aspects of this are pertinent to Dixie applications. Dixie applications will often execute far away from their creator, and must be able to deal with the local environment in ways anticipated by the programmer, but not interactively with the programmer or invoker during execution.

One aspect of making a program aware of itself is to make accessible to the program the attributes of objects within the program. These attributes are known to the compiler or interpreter, and often to the run-time code, but generally are not accessible by the program itself. Objects (simple variables, arrays, structures, procedures and functions, etc.) have attributes such as type, dimension, and whether or not they have been written to (set). There are times when a programmer would like to obtain the values of these attributes. Variables to hold these values can be created and set explicitly in some circumstances, but not always. For example, when an array is passed by name, it would often be convenient to obtain the size of the array from attributes known to the run-time code.

The current type of a variable is an interesting case in several very high level languages. In some of these languages, the type of all variables is "string" at the language level, but has a dynamic type such as integer, floating point or string at run time. Though the interpreter knows or can determine this dynamic type, it is not always available to the programmer. As an example of its use, one might like to construct a sort routine that checks on the types of the elements being sorted, and acted according to this information.

Another aspect of a program that is likely to be of interest for Dixie applications is how long the program has run, according to some measure. Host computers that run Dixie so that Dixie applications can access local resources may wish to provide limitations on the amount of execution time any one Dixie application can consume. An approximation for this is the number of statements executed. The programmer may wish to write a Dixie application that uses most of the available time, and then interrupts itself to prepare a message reporting the results obtained before terminating (or being terminated). Methods of making this information available conveniently will be explored.

A second area of interest is how one constructs programs to react to errors. The REXX language has incorporated the ON CONDITION concept from PL/1. This is very useful in many cases. The basic concept is that if a certain condition arises during a program, this creates a "trap" to a specified subroutine. It may or may not be possible to return to the point at which the trap occurred, depending on the cause of the trap and details the programming language.

This notion of "if some state is reached, invoke this action" as a global statement to be checked for continuously during program execution, in contrast to explicitly programmed checks, is very powerful. It leads to a very useful kind of internal multithreading within programs. I refer to this as "internal" because it is not visible to the operating system.

REXX includes the ability to turn condition checking on and off for specific events. It is very useful for building routines that can react to errors in dealing with the operating system without placing lots of messy error checking code in the middle of what may be already complicated blocks of code.

There are several issue in implementing and using an on condition feature. Obviously it can be expensive to carry out checks continuously, so this must be dealt with in sensible ways. It must be clear to the programmer who cares what the overhead is in using this feature. It must be possible to turn checking on and off, so that sections of code where the condition being checked for will not occur need not bear the overhead.

When a trap occurs, the question arises of how to determine where the trap was generated. It would be nice to be able to insert labels in the code for this purpose (as it would be for some debugging tools). If this were possible, the value of a variable associated with the trap could be checked to identify the trap location. In REXX it is possible to obtain a line number, which is useful for post-mortem debugging. This is hard to make use of at run time because line numbers will change each time the program is modified, and it is a problem to keep track of them accurately for use within a trap routine.

The availability of an on condition that is triggered by the number of statements executed would be a useful solution to the problem mentioned above of trapping near the end of a Dixie applications allotment of execution time.

References

[1] N. Borenstein and M. Rose, “EMail with a Mind of its Own: the Safe-Tcl Language for Enabled Mail”, to be published in *ULPAA `94*.

[2] B. Borden. R. S. Gaines and N. Shapiro, “MH, A Message Handling System for the UNIX Operating System”, The Rand Corporation, R-2376-PAF, October 1979.

[3] M. F. Cowlishaw, *The REXX Programming Language*, Prentice Hall, 1990.

[4] R. S. Gaines, “An Operating System Based on the Concept of a Supervisory Computer”, *Communications of the ACM*, Vol.15, No.3, March 1972.

[5] B. C. Neuman, “The Prospero File System: A global file system based on the Virtual System Model,” *Computing Systems,* 5(4),p. 407-432, FAll 1992

[6] J. Ousterhout, *Tcl and the Tk Toolkit*, Addison-Wesley, Reading Massachusetts, 1994.

[7] M. Rose and N. Borenstein, “A Model for Enabled Mail (EM)”, draft in preparation.

[8] M. Rose and N. Borenstein, “MIME Extensions for Mail- Enabled Applications: Application/Safe-Tcl and Multipart/enabled-mail”, draft in preparation.

[9] G. van Rossum, Python 1.0.1, documentation and code available for anonymous ftp from ftp.uu.net in /

languages/python.

[10] L. Wall and R. Schwartz, *Programming Perl*, O'Reilly & Associates, 1990.

Stockton Gaines has worked in the areas of computer operating systems and computer security for over 25 years. His paper “An Operating System Based on the Concept of a Supervisory Computer” [4] was presented at the 3rd Symposium on Operating Systems Principles in 1971. He was chairman of the ACM's Special Interest Group on Operating Systems (SIGOPS) and Operating Systems editor of the Communications of the ACM from 1975 through 1980. He was a consultant starting in 1981, and consulted on operating systems for IBM, Honeywell and Control Data Corporation, among others, during the years 1981-1989. Dr. Gaines directed ISI's research on parallel computing from 1989 to 1992, which include the porting of the Mach operating system to a distributed memory parallel computer . He developed the concepts of System Manager and Job Manager which form the basis of the Prospero Resource Manager being developed by Cliff Neuman at ISI.

Dr. Gaines chaired the first conference on computer security, held in Princeton, NJ in 1972. He chaired the technical committee to oversee the development of a secure version of Unix for ARPA during 1976-1977. Together with Norman Shapiro of the Rand Corporation, he designed the MH mail handling system [2], and he directed its development. Subsequently, he did research on secure message system . As part of his consulting he worked on security issues for a number of clients.