IMPLEMENATION OF A VISUAL UNIX PROCESS CONNECTOR

by

Mitali Bhattacharyya
David Cohrs
Barton Miller

Computer Sciences Technical Report #677

December 1986

Implementation of a Visual UNIX Process Connector

Mitali Bhattacharyya
David Cohrs
Barton Miller

Computer Sciences Department
University of Wisconsin – Madison
1210 W. Dayton Street
Madison, Wisconsin 53706

ABSTRACT

UPCONN is a tool used by a programmer to visually describe the connections between the processes in a distributed program and to execute the distributed program. With UPCONN, the implementation of processes in a program is separated from the description of the connections between them. The programmer is provided with a screen editor which enables processes to be described and allows the connections between these processes to be specified. When a distributed program is described, UPCONN allows the user to execute the program or save the program for later use.

UPCONN has a modular design which makes it easily extendible and allows its parts to be used independently. A library of processes and procedures is included to perform specialized tasks, such as message monitoring and file access. The library provides a method for adding new features to UPCONN. Several existing UNIX utilities can be linked in with UPCONN to provide a variety of functions. UPCONN is implemented under 4.3BSD UNIX and runs on a network of homogeneous computers.

1. Introduction

There are few tools available that simplify the structural description of programs in a distributed system. Specifying the structure or the interconnection of processes in a distributed computation can be a cumbersome task for the programmer. This has lead to the development of tools to aid in describing the connections in a distributed program. We have implemented one such tool, a visual process connector called UPCONN, which is designed for the UNIX environment.

Our tool allows the programmer to specify process placement and interaction by using a graphical description of a distributed program and to execute the program once the program is described. The graphical description defines the structure of the distributed program and includes the processes in the program and the permanent connections between them. UPCONN is meant to simplify the creation of these permanent connections between processes and to allow a structural specification of the distributed program. By describing the connections with UPCONN, the computation performed by processes is separated from the creation of the connections between them.

UPCONN provides a simple, functional, and extendible interface. The extensions are made using a library rather than filling UPCONN with many special features. The lack of extra features allows UPCONN to be modular in its design and permits the use of parts of the package by other applications. We feel that these characteristics make describing a distributed program easier.

Our paper proceeds as follows. Section 2 presents previous work that has been done in the area of process connection tools. This motivates the need for our tool, UPCONN. If we could build a perfect connector, it would have a number of properties, and Section 3 discusses the properties of such an ideal process connector. We discuss our tool, UPCONN, in Section 4, including a description of the implementation of the tool and its current support software. Current applications of UPCONN are summarized in Section 5. We address work that remains to be done in Section 6, and our conclusions appear in Section 7.

2. Related work

Many of the ideas used in UPCONN are motivated by studying other systems. These ideas include the methods for connecting processes, the structural specification of a program, and a visual and interactive environment for program description and execution.

Each system that we studied includes a method for connecting processes. A process must first locate the process with which it wants to communicate and then set up a connection to that process. It is difficult for unrelated processes to locate each other in a distributed system. In addition, code to handle these connections must be included in every distributed program, resulting in duplication of code. These were the motivations behind the creation of the switchboard task in the DEMOS operating system[Baskett77]. The switchboard provides a mechanism for creating connections. When a process is created, it registers its name and the sending end of a communication channel, or *link*, with the switchboard. When another process wishes to connect to an existing process, it asks the switchboard for the link to the process that the switchboard holds. This enables a process to form connections with another registered process by specifying the name of that process. Application programs no longer need to know the implementation of connection establishment.

The switchboard provides low level support for process connection; a process must still include the calls to the switchboard at run time. A higher level tool can make these calls for the program and execute the program once its connections are in place. The *connector*, a tool to specify these connections, was developed for the Charlotte Distributed Operating System[Finkel83]. The connector separates the implementation of the processes from the implementation of the connections between these processes. It connector uses a special description file that allows the user to specify processes and the connections between them. Using the description file, the connector creates the processes, forms connections between them, and makes known to each process the connections that it holds. The connector uses a switchboard to register the processes that it starts. Using a description file and separating the computation performed by processes from the connections between them are powerful ideas.

There are several ways of specifying the structure of connections in a distributed program. The Charlotte connector uses a description file to define the structure of the program and the way its processes interact. The programmer may specify either single processes or arrays of processes. Arrays allow the specification and connection of a number of similar processes. Arrays are a simple structuring technique, and permit the programmer to quickly specify many distributed programs, such as replicated servers.

The Hierarchical Process Composition model [LeBlanc85] (HPC) uses an object-oriented model for specifying connections. It provides a method for defining typed objects and operations on them. The basic objects in HPC are processes and *channels*, which are communication links between processes. As in the Charlotte connector, HPC has a mechanism for separating the implementation of processes from connections. This is done using an object called the *controller*. A set of processes and channels can be encapsulated into one object similar to the way object modules are encapsulated into an executable program by a linking loader. HPC is a low level mechanism, and a friendly user interface is not provided.

Visual tools aid in constructing distributed programs. Using such tools, a user can interactively edit a pictorial representation of the processes and the connections between processes in a program. The Poker Parallel Programming environment [Snyder84] developed such a graphical tool for the Configurable, Highly Parallel (CHiP) computer. The CHiP computer consists of 64 interconnected processors. The architecture of CHiP can be defined as a two-dimensional structure called a *lattice*. The lattice is made up of nodes, which represent the processors in their system, and arcs, which are the programmable switches or communication channels between them. Poker runs on a VAX 11/780 under UNIX and is used to write and run parallel programs in this environment. To create a parallel program, the user interactively edits a pictorial representation of this lattice structure and specifies which processes should be placed on which processors and the communication between them. Poker provides the architectural characteristics of the system in a pictorial format and allows the user to graphically describe the communication between processes. The visual interface provided by Poker greatly simplifies the description of a parallel program, but the programmer is forced to adapt her algorithm to the CHiP architecture.

A visual interface is especially useful when combined with a system that allows interactive creation and destruction of connections. Silicon Graphics has developed a connection manager called *conman* for this purpose[Haeberli86]. The conman is a process, similar to the switchboard, that allows the user to control the connections between processes. Associated with processes are input and output ports. These ports have names and are registered with conman when the process starts. Connections are made by connecting one process's input port to another's output port. A powerful feature is that connections are created and broken dynamically under user control. To allow the user to maintain these connections, the window manager is used to display, create, and destroy connections. This system is tailored for interprocess communication on a single workstation, rather than executing a distributed program.

Several of these ideas proved important to the development of UPCONN. The graphics interface for defining the distributed program from Poker and conman and the description file from Charlotte were especially important. In our implementation, a variant of a switchboard was used to make the connections.

3. The Ideal Process Connector

By examining related work and examining our own goals, we outline a set of characteristics that an ideal process connector should have and define several terms to use when discussing process connectors. The term *process* refers to a UNIX object file in execution. A *distributed program* is a group of processes that are cooperating toward a common goal. These processes communicate via message passing. A *connection* is a pair of connected 4.3BSD UNIX sockets. An *object* is defined to be either a process or a connection. Finally, a *host* is a computer on which a process in the distributed program can execute.

An ideal connector allows the user to interactively specify the structure of the distributed program. It uses graphical pointing and display capabilities to provide an environment for constructing, modifying and executing a distributed program. Processes, either new processes or existing server processes, are specified using the tool, and connections are made between them. Similar to HPC, the tool can be used to define and replicate complex structures of processes and connections.

Second, the ideal tool interactively executes a distributed program and monitors and controls execution. Given a structural description of the program, the connector will execute the processes of the program on the hosts in the network. By using multiple windows, the tool monitors process activity by providing a monitoring window for individual processes and connections. Statistics about message traffic, such as the contents of messages and the times at which they are sent, are automatically gathered. As well as monitoring message traffic, the user can control execution by modifying connections during execution, as in conman.

The connector should provide process placement. The user specifies a particular host in the network on which a process should execute, restricted to those hosts which the user can access. If the user has no preference, the tool places the process on a host that can most readily handle the extra load.

Last, an ideal connector should be a modular tool which supplies a structure to which other functions can be added. Modularity is attained by including these functions in a library of processes that the user can link in with her program. New functions can be added by extending the library and leaving the primitives unchanged. These functions are provided by ordinary processes, so one can take advantage of the semantics that the tool provides. For example, file access can be added to the tool by providing a special process in the library instead of adding any new types of objects.

4. UPCONN

4.1. Overview

UPCONN consists of three major programs (see Figure 1): UPedit, the editor used to graphically connect a distributed program, UPsh, the program that executes the distributed program, and UPstart, the program that UPsh uses to start up the individual processes on the appropriate hosts.

UPedit allows the user to pictorially describe processes and the connections between them. It also provides a menu of commands with which the user can manipulate the description of processes and connections. UPedit is an interactive editor whose objects are processes and connections rather than text files or documents.

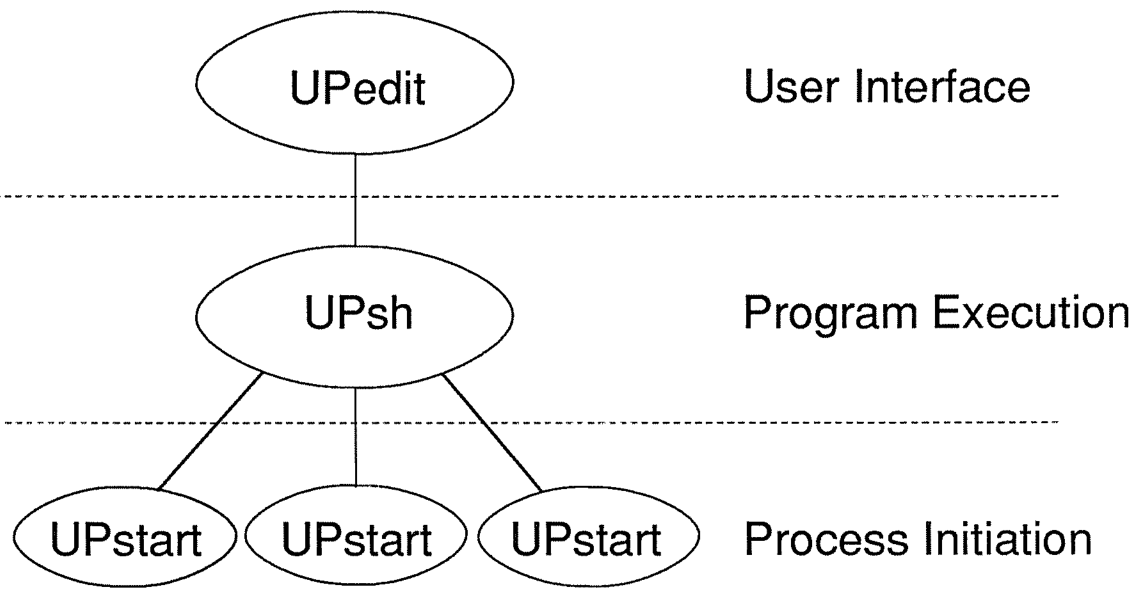


Figure 1: Functional hierarchy of UPCONN

A *description file* is created by UPedit to permanently store the pictorial representation of a program. It is the only means of communication between UPedit and UPsh. The file contains a description of the processes and their connections in a format understood by UPedit and UPsh.

UPsh and UPstart execute the distributed program. UPsh interprets the description file and uses UPstart to execute the individual processes in the program. UPstart forms the connections between processes using the switchboard service [Draheim87] and executes the individual processes. UPsh and UPstart provide a way to separate the execution of a distributed program from the description of a program.

4.2. UPedit

UPedit is the primary user interface to UPCONN. The programmer uses UPedit to describe the structure of a distributed program and to interactively execute the program. UPedit displays the program in a window on the user's screen. An example of this window is shown in Figure 2. The programmer, using a mouse and keyboard, creates processes and defines the connections between them. The basic functions provided by UPedit include creation, modification, replication and deletion of processes and connections. The programming environment is completed by providing a way to execute programs from within UPedit.

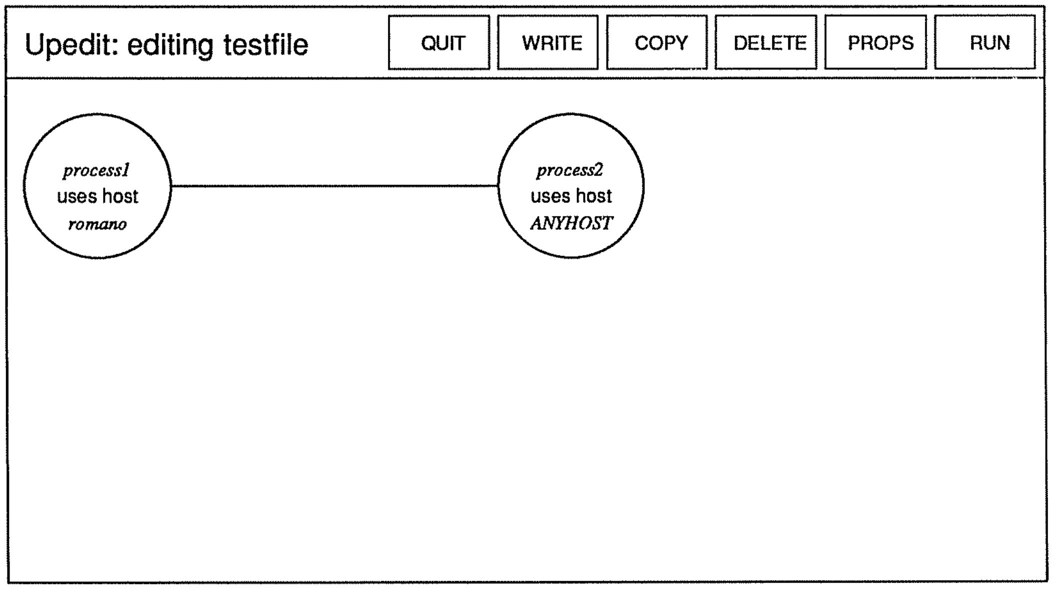


Figure 2: An example of the UPedit display

Invoking UPedit results in a window being displayed on the user's screen. The UPedit window is divided into two regions. The upper section of the window is called the *banner*. The name of the distributed program is displayed in the banner. The rest of the banner region contains the editor functions which are listed in Figure 3 and are described below. The area below the banner is used for editing a program. In Figure 2, "testfile" is the name of the distributed program, and "process1" and "process2" are the names of the processes that constitute the program.

To describe a distributed program, the programmer specifies processes and connections between these processes. A process is represented by a circle and can be displayed on screen by pressing a mouse

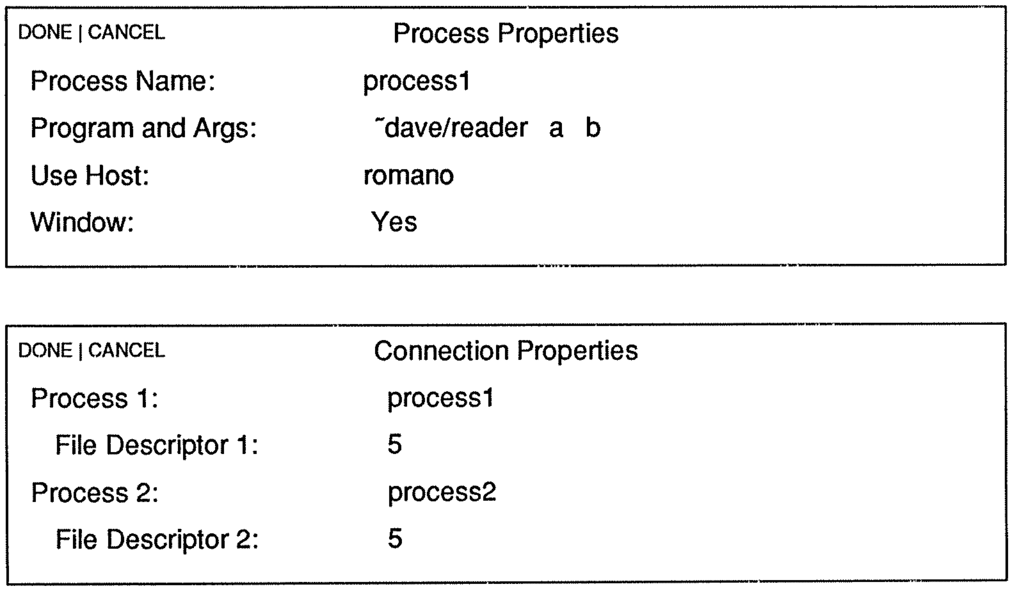
QUIT	Quit the editor
WRITE	Write out the picture into a description file
UNDO	Undo the last change made
COPY	Make a copy of an object - indicate destination with mouse
DELETE	Delete an object
PROPS	Display property sheet for an object
RUN	Run the distributed program

Figure 3: UPedit commands

button. When some processes have been created, the user may create connections between the processes. A connection is represented by a line and is displayed by selecting, with the mouse, the two processes to connect. There is currently a restriction that only one connection may exist between any pair of processes.

After a process or connection has been created, the programmer can specify attributes of this object. Attributes of a process include the process's name, the host on which the process will run, and whether the process needs its own monitor window for displaying debugging information. Connections also have attributes. When creating or modifying a connection, the user can specify the UNIX file descriptors through which the processes should communicate. If the user does not specify some of these attributes, such as the process name or a file descriptor, default values are chosen for them. UPedit displays process or connection attributes in a *property sheet.* Examples are shown in Figure 4. The use of a property sheet was motivated by a similar mechanism provided in the Xerox STAR word processing system[Smith82].

Several commands are available to edit a program. The commands that manipulate the current program are shown in the banner region of Figure 2 and are described in Figure 3. Other commands are entered from the keyboard. For example, UPedit provides a command for saving the program in a specific file. When the user is finished editing, she can save the program for later use. The program may either be



DONE | CANCEL	Process Properties
Process Name:	process1
Program and Args:	~dave/reader a b
Use Host:	romano
Window:	Yes

DONE | CANCEL	Connection Properties
Process 1:	process1
File Descriptor 1:	5
Process 2:	process2
File Descriptor 2:	5

Figure 4: UPedit Process and Connection Property Sheets

retrieved again with UPedit, or it may be executed outside the editor environment as explained in Section 4.3.

Once the pictorial representation has been created, UPedit is used to start execution of a distributed program. UPedit first translates the picture into the description file format. It then uses UPsh to execute the description file. After the description file is made, UPedit has nothing more to do with the actual execution of a distributed program. When execution of the program is completed, control returns back to UPedit.

An example of the description file is shown in Figure 5 (corresponding to the program in Figure 2), with each statement in the example file preceded by an explanation of the statement. The description file is easy to read, but the user of UPCONN need never directly manipulate this file. The complete grammar for description files is given in Appendix A.

The description file separates UPedit from the execution of the distributed program. This separation allows other programs to generate description files. If a specific application needs a different user interface or does not require a user interface at all, it can generate the description file itself.

4.3. UPsh and UPstart

UPsh reads the description file, constructs the commands necessary to execute the program on the remote hosts, and calls UPstart to execute the individual processes. UPsh first interprets the description file and determines on which host each process wants to run. If no host is specified, UPsh picks a host. UPsh starts individual processes of the program on the remote hosts by remotely executing UPstart on each host. The executable image for a process to be started on the remote host might not exist on that host. If this is the case, UPsh opens a special connection (called the File Transfer Path) to UPstart on the remote host and then copies the file (see Figure 6). After the program begins, UPsh waits for all of the processes to complete and then exits. If the user wishes to cancel execution, she may type the interrupt character in the window running the UPsh command.

UPstart forms the connections between the processes and begins their execution. It is executed once for each process in the distributed program, on the host on which the process will run. UPstart forms connections to the process using the switchboard daemon. Execution of a process begins after all connections for this process are made.

```
#!/usr/local/upsh
# UNIX magic necessary to run this program

# first, the name of the process: "process1"
process process1;
# next, the command line necessary to execute the process
      args reader a b;
# the host on which to run the process
      using romano;
# the location to display the process in UPedit
      at (100,100);
# a flag which causes a monitor window to appear when the process runs
      window;

# the description for process "process2"
# no specific host or window for this process
process process2;
      args writer;
      at (300,100);

# a connection between process1, on descriptor 5 and process2, on descriptor 5
connect <5,process1> <5,process2>;
```

Figure 5: An example of a UPCONN Description File

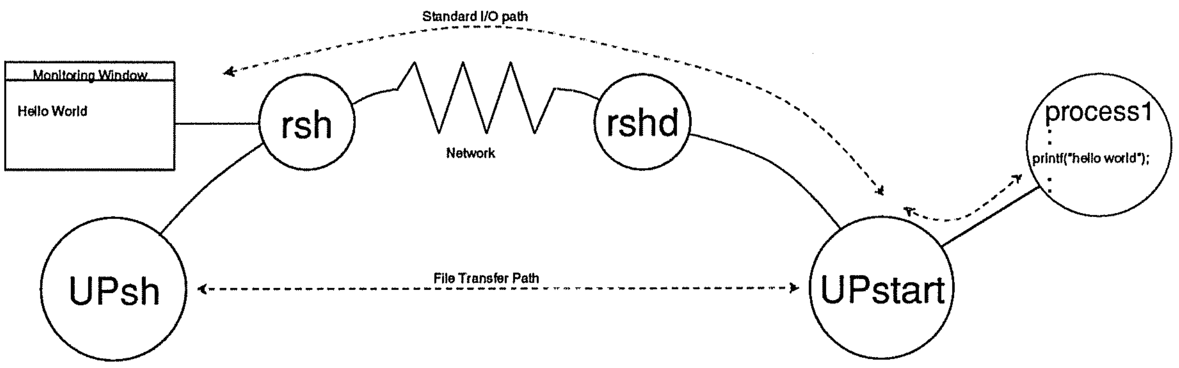


Figure 6: Interaction and communication of UPCONN support processes

UPsh uses the UNIX remote shell (*rsh*) facility[RSH] to execute UPstart on the remote host. Rsh and its remote partner, *rshd*, provide remote execution of a process. This facility authenticates users and initializes the remote run time environment before allowing a process to execute. The interactions between UPsh, UPstart and rsh are shown in Figure 6.

Windows provide a means of monitoring the output from UPsh and UPstart. When execution of the program begins, a window is created that displays output generated from UPsh. In addition, any process can have a monitoring window. If a monitoring window is requested, output from UPstart and the user's process will be displayed in this window. The monitoring window is a process attribute that can be specified when describing the distributed program (see Section 4.2).

Programmers can use the UPCONN modules as a package or use parts of UPCONN by themselves. For example, a programmer may directly invoke UPsh without going through UPedit because stored programs can be executed directly by UPsh. Similarly, a programmer may use UPedit to merely edit a distributed program description and not execute it until a later time. An application may use parts of UPCONN, such as UPsh, without using UPedit if such an interface is not necessary or if a different user interface is desired. This is useful in an application that uses a distributed program to perform a computation; the application can automatically generate a description file for the program and execute it with UPsh. Because of the modular design, use of UPsh is not restricted to the UPedit environment.

4.4. Additional Features

UPCONN is designed to provide primitives upon which features can be added. Several utility processes and interface procedures have been built on top of the basic UPCONN programs. The processes described here are the UPmonitor, a message monitoring facility, and the UPfileserver, a simple utility that makes file access appear to be like any other communication with a process. The processes are designed so that a user can link them into their program using UPedit. Other simple UNIX utilities, like *cat* and *tee* can also be used. Interface procedures are provided through a library, UPlibrary, that can be linked with individual processes that the user creates. The modular design of UPCONN made it easy to provide these utilities without making changes to the other parts of UPCONN.

4.4.1. UPmonitor

UPmonitor is a utility process that monitors messages between any pair of user processes. The user can insert this process between a process pair before executing the program. This is done in UPedit by connecting the two processes to be monitored to UPmonitor rather than to each other. UPmonitor is invisible to the processes that it is monitoring. This allows the monitor to intercept the messages passed between processes and display these messages in the UPmonitor window and log them for later use. UPmonitor also allows the user to stop the message flow between the processes it is monitoring. These features make UPmonitor a useful debugging aid.

4.4.2. UPfileserver

UPfileserver is a utility process that allows processes in a distributed program to treat files as ordinary processes and access these file by passing messages. By providing file access through the UPfileserver, file access is incorporated into the UPCONN semantics without adding special object types. UPfileserver supports read, write, seek, begin transaction and end transaction operations. The begin and end transaction requests provide mutual exclusion and synchronization, but they do not currently perform other functions such as atomic updates. This could be provided in a different utility process. The fileserver operations are performed by calling special access routines, described in the next section.

The programmer connects to the UPfileserver process using a standard UPCONN connection. The message passing is hidden through the access routines we provide, and access to the file server is treated like access to a regular file. A side effect of providing a method for accessing files is that we also provide distributed file access. UPfileserver is an example of the power of UPCONN's simple, modular interface.

4.4.3. UNIX utilities

A number of UNIX utilities are also applicable to UPCONN and may be inserted into a distributed program like any other process. This is because UNIX utilities will generally read from their standard input file and write to their standard output file when no special input or output files are specified. This means that they can be inserted into a distributed program and have connections attached to their input and output descriptors. Some examples are cat, sed, tee, yes, and grep. Cat can be used to generate a stream of data from a file. It can also be used to display all incoming messages in a monitoring window. Sed can edit a stream of data passing between two processes. The tee command can be used to log messages in a file, while passing them through to another process as well. Yes can be used to repeatedly generate messages. This is useful in debugging programs. Finally, grep can be used to filter out unnecessary messages when testing a distributed program. These simple examples show the flexibility provided by UPCONN and demonstrate the wide range of existing software that can be used with UPCONN.

4.4.4. UPlibrary

UPCONN includes a library of procedures that are linked in with individual processes in the distributed program. They include routines to provide mappings between process names and their associated file descriptors. Also included are routines to simplify access to UPfileserver.

The routines, get_conn_fd and get_conn_name provide a mapping between the UNIX file descriptor and the UPCONN name for a connection. Given the name of a process, get_conn_name returns which file descriptor that is associated with the connection to that process. Get_conn_name is used when a process does not know the file descriptor to use to communicate with a specific process. For example, if a file descriptor is not specified when creating a connection in UPedit (see Section 4.2), UPstart will choose the file descriptor. The process's name will be known, but not the file descriptor. Calling get_conn_name will determine this for the process.

Given the file descriptor, get_conn_fd returns the name of the process using that descriptor. This routine is used when the file descriptor is known, but the process's name is not. For example, UPmonitor requires that the file descriptors 0 and 1 be used for making its monitoring connections. When it begins execution, it calls get_conn_fd to determine the names of the processes it is monitoring.

Access routines for the UPfileserver are also provided. They allow the user to treat UPfileserver as a file even though it is implemented as a process. These routines are upfs_read, upfs_write, upfs_seek, upfs_begin_trans and upfs_end_trans. The read, write and seek routines have the same semantics as the standard UNIX read, write and seek routines. This allows access to files through UPfileserver to be the same as access to regular UNIX files.

5. Current Applications

UPCONN has been designed and implemented under 4.3BSD UNIX using the X window package[Gettys85]. This allows it to be used on a large number of computers in our research environment and makes it available to many researchers in our department.

UPCONN has already proven helpful to another project in the Computer Sciences Department. This project, DIB[Finkel85], provides a means of generating automatic parallel implementations of backtracking programs. DIB has previously been designed to run on the Crystal multicomputer[Dewitt84]. Its designers want to be able to use DIB on their UNIX workstations, rather than the Crystal nugget. DIB used UPsh and UPstart to places processes, form the necessary connections and start program execution.

Porting DIB has also shown us that UPCONN needs improvement in some areas. The most important area is in process execution. Processes are executed remotely using the rsh facility. Rsh is very slow in starting process execution because it is a general facility. We are currently working on having UPCONN directly start remote processes.

6. Future Work

UPCONN provides many of the features that our ideal connector describes but does not meet all of its goals. Specific areas of improvement include process placement, improved facilities to allow replicating objects and groups of objects, faster start up time, and greater support for distributed debugging.

If the user does not specify on which host a process should execute, we want to pick a suitable host for it using some form of a process placement policy. This can be done by looking at the current load averages of the hosts and choosing the host with the lowest load. We provide a mechanism for process placement and are not addressing the issues involved in developing load balancing policies, however, any specific load balancing policy could be integrated with this mechanism. See[Chou82] for a description of load balancing policies.

We want to build higher level semantic descriptions on the basic functions provided by UPCONN, similar to the encapsulation concept in HPC. This requires the COPY function in UPedit to be expanded. The user will be able to create higher level descriptions by replicating groups of processes and connections. No changes are needed in the grammer of the description file.

There are also some annoying aspects to UPCONN. The worst aspect is the slow startup time when a program begins execution. The rsh facility is extremely slow, and we are replacing this with a faster remote execution facility.

Greater support for distributed debugging needs to be added to UPCONN by extending UPmonitor. UPmonitor currently records messages and displays them in an easy to read format. Extensions will allow the user to stop and restart the monitor and be able to replay and insert messages.

7. Conclusions

UPCONN is designed to visually aid the user in connecting and executing distributed programs. It simplifies the programmer's task by providing a graphical and structural approach to writing a distributed program. UPCONN provides a mechanism to separate the implementation of processes from the connections between them. Rather than provide an elaborate set of options, our connector tool provides a simple, functional interface. The tool is designed so that new features can be added easily. Some extra features have already been added, such as the UPfileserver, without changing the structure of UPCONN. Instead, they are in a library of processes which the user can include with their distributed program. We feel that this approach is more flexible than an approach which requires adding to the basic system itself.

The modular structure of UPCONN made the design and implementation much simpler. UPCONN can be used to either develop a distributed program, or execute a completed program in a production environment. A user can use the functionality of UPCONN without direct knowledge of the underlying implementation.

We designed UPCONN to make distributed programming possible on the workstations in our research environment and to make describing these programs easier. To this end, UPCONN runs under the 4.3BSD UNIX operating system. The user interface makes use of the bitmapped display provided by the workstations. The tool has been made available for use on our network and is already being used in our computing community.

REFERENCES

[RSH]	*Rsh – remote shell,* 4.2BSD UNIX Programming Manual
[Baskett77]	F. Baskett, J. H. Howard, and J. T. Montague, "Task Communication in Demos," *Proceedings of the Sixth Symposium on Operating Systems Principles,* pp. 23-31 (November 1977).
[Chou82]	T. C. K. Chou and J. A. Abraham, "Load balancing in Distributed Systems," *IEEE Transactions on Software Engineering* **SE-8**(4) pp. 401-412 (July 1982).
[Dewitt84]	D. Dewitt, R. Finkel, and M. Solomon, "The Crystal multicomputer: Design and Implementation Experience," Computer Sciences Technical Report #553, University of Wisconsin – Madison (September 1984).
[Draheim87]	D. Draheim, B. P. Miller, and S. Snyder, "A Reliable and Secure UNIX Connection Service," *Proceedings of the Sixth Symposium on Reliability in Distributed Software and Database Systems,* (March 1987).
[Finkel83]	R. Finkel, M. Solomon, D. Dewitt, and L. Landweber, "The Charlotte Distributed Operating System," Computer Sciences Technical Report #502, University of Wisconsin – Madison (October 1983).
[Finkel85]	R. Finkel and U. Manber, "DIB — A distributed implementation of backtracking," *Proceedings of the Fifth International Conference on Distributed Computing Systems,* pp. 446-452 (May 1985).
[Gettys85]	J. Gettys, R. Newman, and T. Della Fera, *Xlib – C Language X Interface,* MIT Project Athena (November 1985).
[Haeberli86]	P. Haeberli, "A Data-Flow Manager for an Interactive Programming Environment," *Usenix Association Summer Conference Proceedings,* pp. 419-428 Atlanta, Georgia, (June 1986).
[LeBlanc85]	T. LeBlanc and S. Friedberg, "Hierarchical Process Composition in Distributed Operating Systems," *Proceedings of the Fifth International Conference on Distributed Computing Systems,* pp. 26-34 Denver, Colorado, (May 1985).
[Smith82]	D. C. Smith, E. Harslem, C. Irby, and R. Kimball, "The Star User Interface: An Overview," *Proceedings of the National Computer Conference,* pp. 515-528 Houston, Texas, (June 1982).
[Snyder84]	L. Snyder, "Parallel Programming and the Poker Programming Environment," *IEEE Computer* **17**(7) pp. 27-36 (July 1984).

Appendix A: BNF for UPCONN Description File

```
<file>              :     <process_defs> <connect_defs>

<process_defs>      :     process_def { process_def }

<process_def>       :     "process" <word> ";" <clauses>

<clauses>           :     { <clause> }

<clause>            :     "using" <word> ";" |
                          "arg" <arglist> ";" |
                          "at" "(" <number> "," <number> ")" ";" |
                          "window" ";"

<arglist>           :     <word> { <word> }

<connect_defs>      :     { connect_def }

<connect_def>       :     "connect" "<" <number> "," <word> ">"
                          "<" <number> "," <word> ">" ";"

<word>              :     <char> { <char> }
<number>            :     <digit> { <digit> }

<char>              :     <letter> | <digit> | <other>
<letter>            :     "a"..."z" | "A"..."Z"
<digit>             :     "0"..."9"
<other>             :     "!" | "@" | "$" | "%" | "^" | ...
```