MTX – A Shell that Permits Dynamic Rearrangement of Process Connections and Windows

Stephen A. Uhler

Computer Systems Research Division
Bellcore
445 South St.
Morristown, NJ 07960
sau@bellcore.com

ABSTRACT

MTX is a UNIX™ *shell,* implemented as a single user process, that has an expanded notion of pipes and I/O redirection. MTX supports creating and manipulating processes, connecting their inputs and outputs in an arbitrary directed graph. Each process can have as many windows or *virtual terminals* as required for user interaction. Through a control window, MTX can dynamically manipulate the connections among ongoing processes. In addition, MTX can rendezvous and connect to MTX servers on remote hosts, conveniently providing pipe connections across a network of computers. This paper describes the use and implementation of MTX on a network of UNIX workstations.

Introduction

An important contribution of UNIX to computer science is the notion of a pipe [1]. A pipe enables the output of one process to be directly connected as the input of another process, as in `ls | wc.` The `ls` lists the files in the current directory; the `wc` counts them. Neither `ls` nor `wc` needs any prior knowledge of the other to work together. They only require a common format for data exchange: ASCII characters.

Although pipes are pervasive in UNIX, especially for text manipulation, they have several inherent limitations that, if mitigated, could make pipelines even more useful. The common UNIX shells, *sh*[2] *ksh*[3] and *csh*[4] support linear pipelines. Each process in a pipeline has exactly one input, and one output stream.

Although interprocess communication is limited to linear pipelines by the shell, UNIX supports a more general connection structure, that of a directed graph of processes and interconnections Figure 1 shows an example of four processes connected in a nonlinear pipeline. There are at least two shells that remove the restriction of strict linearity of shell pipelines, *2dsh*[5] and *gsh*[6]. Although these two shells better exploit the UNIX interprocess communication primitives than do the common shells, because the command language used to specify the graphs of processes and interconnects is still linear, the command language is awkward. Setting up sophisticated graphs of processes is complex and cumbersome. To use the new connection facility effectively, processes must determine how many inputs and outputs they have, and what the semantics of the various streams are. Nonetheless, there are circumstances in which this type of extension to pipes is useful.

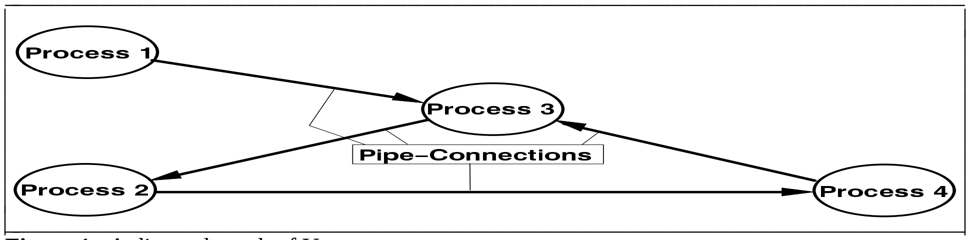


Figure 1. A directed graph of UNIX processes

UNIX also has the notion of a *terminal,* which is a channel for interaction with the user. Traditionally, a terminal is a physical device with a keyboard and a display. Normally there is only one terminal per user, so a pipeline of processes has at most one terminal. Because of this limitation, most processes that function effectively in a pipeline are data filters. They read their input, transform it based on some preset rules, then write their output. Once the process is started, no interaction with the user is allowed.

If each process in a pipeline could have its own interactive channel, or *virtual terminal*, it could prompt the user for information as it was running. Currently, a process must obtain its input from the command line. This would alleviate the burden of completely specifying the function of the command in advance. One could imagine a special version of *grep* that would permit the user to change the regular expression as *grep* was running. While not possible in the traditional UNIX timesharing environment, with a physical terminal per user, the emergence of window systems as the primary user interface to UNIX, makes it practical to provide as many *virtual terminals*, (windows with terminal semantics) to the user as needed. This presents the possibility of constructing a pipeline of interactive processes, as in Figure 2. By taking advantage of this capability, processes that participate in pipelines need not be restricted to passive data filters. They can be fully interactive and permit the user to change their operation dynamically without interfering with other interactive processes elsewhere in the pipeline.

By allowing processes to be connected in a directed graph, with a *virtual terminal* for each process, a much greater range of capabilities is available to the user. Along with this greater flexibility comes a greater complexity. Setting up the connections, processes, and windows properly can quickly get out of hand for all but the simplest of pipelines.

Fortunately, with the ability to use multiple windows, it is possible for the shell, which traditionally relinguishes control of the terminal as soon as the pipeline is started, to have its own window in which to maintain a dialogue with the user, even after starting a command. Through this shell window, connections among processes can be altered dynamically, once a extended pipeline has begun. New processes or connections can be added or removed from the pipeline as needed.

The advantages of dynamic reconfiguration of interactive processes has been demonstrated in the Synergy Dataflow System [7]. Interprocess communication in this system, unlike UNIX pipelines, was designed for programs written specifically to take advantage of the connection manipulation capabilities. It is implemented using new system calls as enhancements to the UNIX kernel, which are understood by the programs that use them. It does not address the ability to integrate existing programs into this environment, and consequently can not provide an environment compatible with existing application programs.

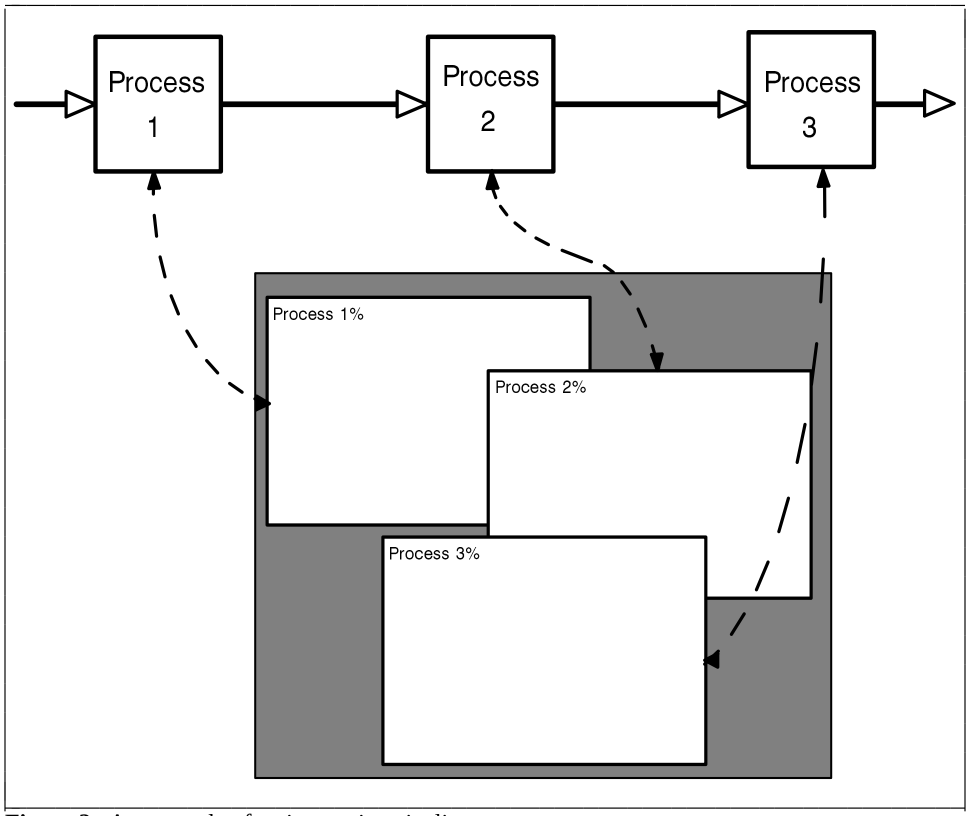


Figure 2. An example of an interactive pipeline

What is MTX

MTX is a UNIX *shell* that has an expanded notion of pipes and I/O redirection. MTX supports: 1) the creation of processes whose inputs and outputs can be connected in an arbitrary directed graph, 2) the creation of as many *virtual terminals* as required by the processes, 3) the dynamic manipulation of the processes and connections after the processes have been started and, 4) the rendezvous and connection to MTX servers on remote hosts, which provides pipe connections across a network of computers.

Ordinarily, MTX gives a process two pairs input-output (I/O) channels. The first pair is for *stdin* and *stdout*, which are normally used by UNIX shells for pipe connections. The second pair of I/O channel is obtained by reading from */dev/tty*, and by writing to *stderr* or */dev/tty*. This is the interactive connection to the user. Many existing UNIX programs already follow this convention, and thus can be used unchanged with MTX. For non-interactive programs such as data filters, or for programs that use only one I/O channel, the two channels can be combined, with the process seeing a familiar terminal interface. The common command shells and editors are in this category. A separate window, belonging to MTX, allows the user to control and modify the state of the process and window connections, and to add or remove processes, windows or connections as needed.

When multiple input streams are directed to the same process, their inputs are merged onto the same stream. Processes do not have to know how many inputs they have, or which file descriptors they need to read. Consequently they need not deal with the complexities of retrieving input from multiple places. This avoids the problem of waiting for input on one input stream while data accumulates on another. Existing programs that deal only with one stream still function properly. An application that is designed to deal with multiple input streams, instructs MTX to put a marker in the data stream whenever input arrives from a different source. Upon reading this marker, the application program can determine origin of the input. By multiplexing this information over a single channel, writing new programs is simplified, while existing programs exhibit reasonable default behavior.

The output from a program can be sent to as many destinations as desired. MTX duplicates the data stream and sends it to all destinations. Before reading the data generated by a process, MTX checks all of the destinations. If any of the destinations is unable to accept data, and would cause a *write* to block, MTX refrains from reading the output generated by the process. This in turn causes the generator of the data to stop. In this manner, MTX passes flow control information upstream. As an example, if the output to *cat* is directed to both a process and a window, and the user keys *CNTL-S* in the window, causing output to the window to stop, MTX causes the *cat* to stop writing data by sending a *stop* command to the *pseudo-terminal* connected to *cat*. Thus any destination of a process can be used to throttle the output destined for other processes or windows.

MTX extends the dynamic redirection capability to a network of computers. MTX can find and communicate with MTX shells running on other workstations, and have inputs and outputs routed to remote displays. These connections to remote hosts can then be used locally as sources or destinations of data for local processes. This feature provides a connection service that can be used as a foundation for shared windows or applications programs that assist cooperative work spread out over a large network of computers.

Sample Uses for MTX

The expanded notion of I/O redirections provided by MTX has several uses. Some of these uses are obvious, others more obscure. Below are some sample uses of MTX for tasks that are difficult, if not impossible, to achieve with an ordinary UNIX shell.

As a simple example, MTX can be used to keep a transcript of a terminal session. Since MTX permits any stream to be duplicated, all input from the user can be saved in a file as it is sent to a program. Similarly, all output from the program to the user can be duplicated and saved as well, either in the same file or a different one. If desired, the program transcript can be filtered before it is saved in a file, perhaps to remove superfluous output, or to time stamp the data. Transcripting can be started or stopped at any time by adding or removing a connection to a running process.

MTX can be used to capture output from a command. It is common to issue a command, then to realize valuable output is scrolling off the display, such as listing a file with *cat* that was longer than you expected. The output would be nice to keep, if only to browse in a more leisurely manner. A typical solution is to terminate the command, and start it over, only with its output directed to a file or into a file browser. Unfortunately, sometimes it is expensive to restart a command, either because it will take a long time to reach the point of output generation, or because the process changes its environment as it runs, so recreating the initial state of the program is not feasible. By using MTX, the output of the program can be temporarily suspended, its output duplicated and sent either to a file or to another program. In an MTX environment, it is possible to have a browser available, so output from an arbitrary program can be connected to it.

A somewhat more obscure, but still useful application of the current implementation of MTX is that of a session multiplexor. Often it is only possible to have a single connection to another machine. This is the normal case with dial-up lines. Ordinarily a program such as *tip* [8] or *kermit* [9] would be used to set up the connection, but then only a single window on the local host can be used on the remote computer. However, if MTX is run on the remote computer after *tip* is used to start a comnection, MTX can create multiple windows on the local host, with one or more for each process running on the remote host, as shown in Figure 3. The data and commands for the processes are multiplexed over the same connection, thus providing a multi-window environment using a single byte stream connection. MTX derives its name from this capability, that of **M**ul**T**iple**X**ing multiple process streams.

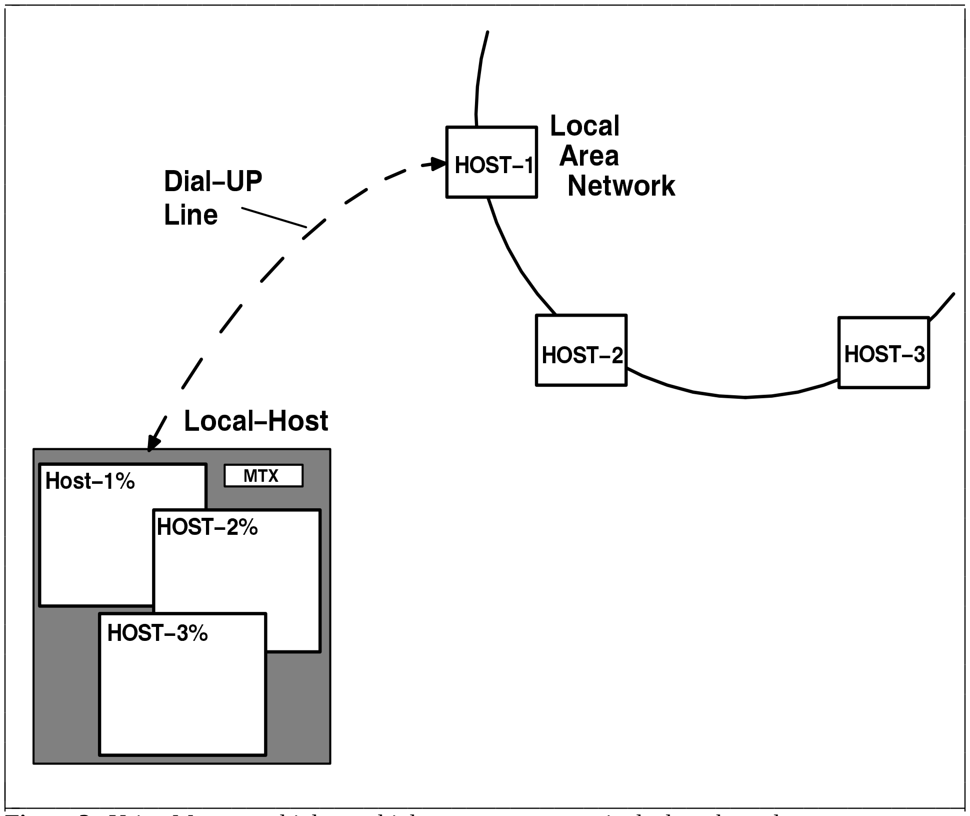


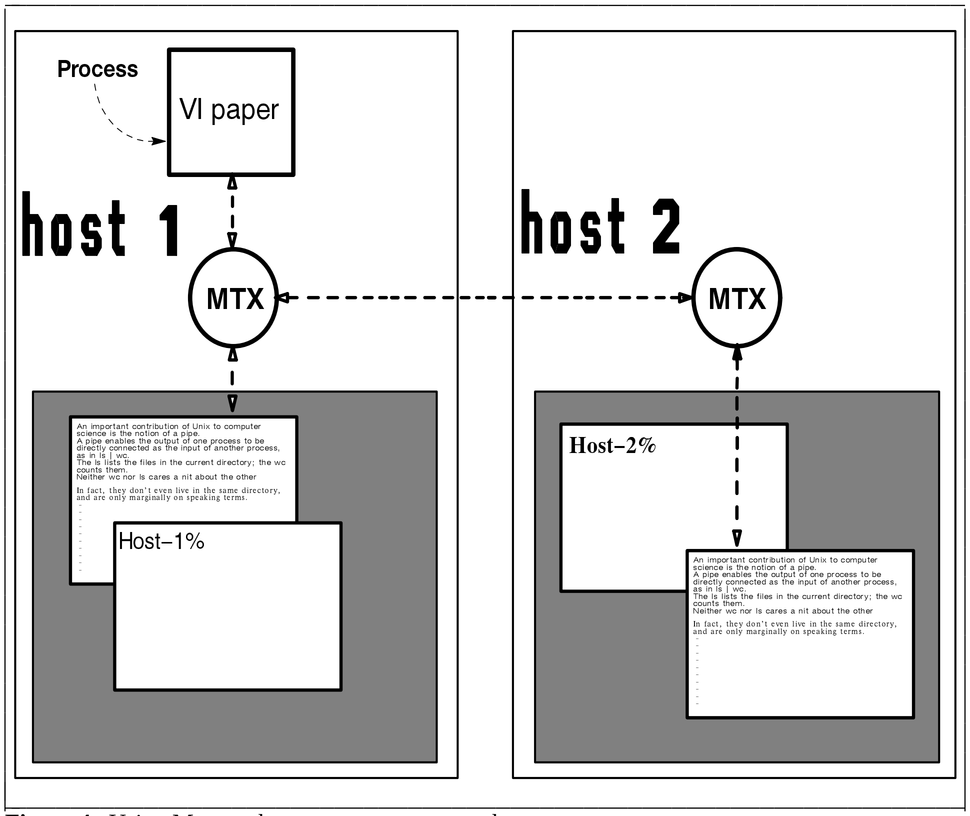
Figure 3. Using MTX to multiplex multiple processes over a single data channel

Perhaps the greatest potential use of MTX is shared windows. A user can connect to an instance of MTX running on different workstations, and send output to it, or receive input from it. An illustrative example is multiple users sharing an editor in different places. A user can start a single instance of the editor, with its output going to several windows, one on each computer. Similarly, input from one or all of the computers can be sent to the editor, permitting simultaneous editing of the same file by several users. This is shown in Figure 4, with the editor *vi* on *host1* being shared by two users, one on *host1*, the other on *host2*.

The above example demonstrates a simple use of sharing; there are no facilities to control or moderate the shared session. The shared application, in this case an existing editor, was not written to understand the notion of sharing. The MTX facilities allow for programs to interrogate and control their connections so new applications can be built that make intelligent use of this capability.

User Interface Description

To understand the user interface, it is necessary to understand the terminology used by the current MTX implementation. MTX understands two basic entities, objects and connections. An object has two communicating ports, one for input (to the object) and the other for output (from the object). MTX understands three types of objects: 1) windows, 2) processes, and 3) ports to remote hosts (or just hosts). All objects have a name, either chosen by the user, or picked by



```
An important contribution of Unix to computer
science is the notion of a pipe.
A pipe enables the output of one process to be
directly connected as the input of another process,
as in ls | wc.
The ls lists the files in the current directory; the wc
counts them.
Neither wc nor ls cares a nit about the other

In fact, they don't even live in the same directory,
and are only marginally on speaking terms.
```

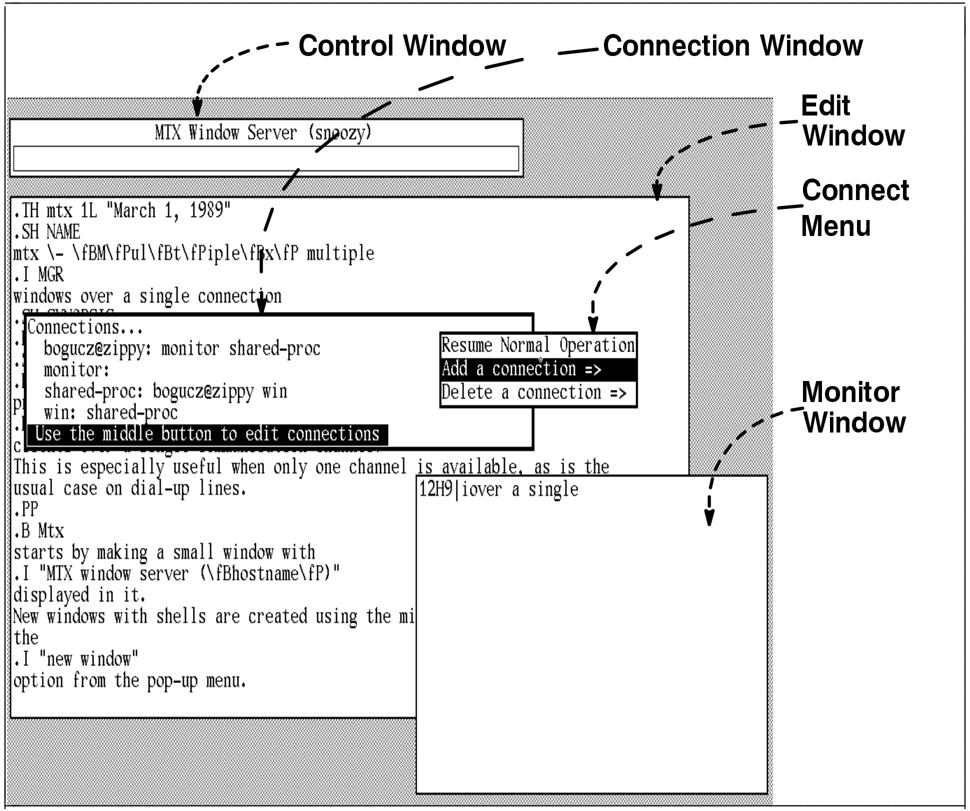
```
An important contribution of Unix to computer
science is the notion of a pipe.
A pipe enables the output of one process to be
directly connected as the input of another process,
as in ls | wc.
The ls lists the files in the current directory; the wc
counts them.
Neither wc nor ls cares a nit about the other

In fact, they don't even live in the same directory,
and are only marginally on speaking terms.
```

Figure 4. Using MTX to share a processes on two hosts

MTX, in addition to other state information that varies according to the object type. A window object is needed to write data to a set of windows or obtain data from the keyboard or other user controlled input device. A set of windows that are all controlled by a single application is accessed through a single window object. Similarly, a process object is the communication port to a process. Each process has one or two process objects, one for file descriptors 0 and 1, the other for file descriptor 2. The process sees a *virtual terminal* interface for each of its process objects. Finally, host objects are used for connecting to MTX servers on remote hosts. There are one or more host objects for each remote host, one object for each data stream. A host object is connected, via a stream socket, to a host object on the remote MTX.

Data flows between objects through a unidirectional data channel called a connection. Each object can have many connections; an object can even be connected to itself. Every connection must start at the output port of one object and terminate at the input port of another. The user interface to MTX consists of commands that manipulate the objects and their connections.

When MTX is started, it configures a control window on the display. This control window has an input line for typing commands to MTX, a message line for status information and messages, and a title line to identify MTX. Most of the user interaction with MTX is through a hierarchy of pop-up menus associated with the control window. The primary entries on the root menu are for creating windows and processes; displaying, adding, or deleting connections; and connecting to or displaying the names of remote hosts. The other entries exist to manipulate various debugging and configuration information, as well as exit or suspend MTX. Selecting an item on the root menu invokes the default action for the submenu anchored at that item. Figure 5 shows a sample MTX session taken from a portion of the window system display on the host *snoozy*. In this example two users, one on host *snoozy* and the other on host *zippy* are sharing a single instance of the editor *vi*. A command script created the *vi* process (shared-proc), a pair of windows, a channel to a remote host (zippy), and all the connections. The smaller window (named *monitor*), displays the keystrokes typed by the user on *zippy*, whereas the *edit* window appears with identical contents on both hosts. The MTX connection window displays the connections among the objects.



```
MTX Window Server (snoozy)
```

```
.TH mtx 1L "March 1, 1989"
.SH NAME
mtx \- \fBM\fPul\fBt\fPiple\fBx\fP multiple
.I MGR
windows over a single connection
```

```
Connections...
  boguczezippy: monitor shared-proc
  monitor:
  shared-proc: boguczezippy win
  win: shared-proc
Use the middle button to edit connections
```

```
Resume Normal Operation
Add a connection =>
Delete a connection =>
```

```
This is especially useful when only one channel is available, as is the
usual case on dial-up lines.
.PP
.B Mtx
starts by making a small window with
.I "MTX window server (\fBhostname\fP)"
displayed in it.
New windows with shells are created using the mi
the
.I "new window"
option from the pop-up menu.
```

```
12H9|iover a single
```

Figure 5. Sample display of MTX running

The default action for the root menu is to create a process (usually an ordinary shell), a window, and a pair of connections. This configuration behaves exactly like shell connected to a terminal. The options on the sub-menu include starting a process or window with no connections. The connections are added later, either through the connection modification options, or with a command script. Provisions are available for users to customize the submenus, adding their own entries.

The user can view and change the current connection information by selecting the *change connection* entry of the root menu. When selected, a configuration window pops-up. The name of each object is displayed along with the other objects connected to it. Pop-up menus associated with the configuration window can then be used to add or delete connections.

The *host* entry on the root menu has a submenu that lists the remote hosts on the network that are available for connections. If a remote host is selected, both the local and remote MTXs create an object for the other host. That *host* object will then appear in the connection window, and can be either the source or destination of a connection. If the remote MTX initiates the connection to the local MTX, a local host object is created. A message is displayed on the status line notifying the local user of the connection. When a remote MTX server is created or destroyed, a message is displayed, thus informing the user of which remote hosts are available for connections.

In addition to the menu driven command interface, MTX supports a primitive command interpreter that allows known configurations of windows, processes and their connections to be set up automatically. The functionality of the command language will be extended as the need arises. This will probably include mechanisms to invoke command scripts automatically when certain connections occur. For example, to share an application with a user on a remote host, simply establishing a connection to the other user could cause the proper configuration of the processes, connections, and windows needed to share the application.

The MTX Operating Environment

The current version of MTX is implemented as a single user process. No kernel modifications required to accomplish the dynamic re-routing of connections (see Figure 6). The MGR[10] window system is used to provide the underlying windowing and user interface capabilities. MTX is started as a window system client program; one MTX is started for each display. MTX need not be on the same host as the workstation, but can run on any computer to which a serial byte stream connection can be obtained. Additional windows and processes are created and manipulated by MTX. Both the input and output to all processes spawned by MTX are routed through one or more pairs of *pseudo-terminals* so MTX can maintain control of all the data streams and route the data around internally. Pseudo-terminals, a pair of virtual devices that simulate the semantics of a UNIX terminal, are used instead of pipes or sockets because a *pseudo terminal* can be configured with either terminal or pipe semantics. Thus, when two processes are connected by MTX in a ‘‘pipeline’’, MTX reads the data from the first process and forwards it to the second one.

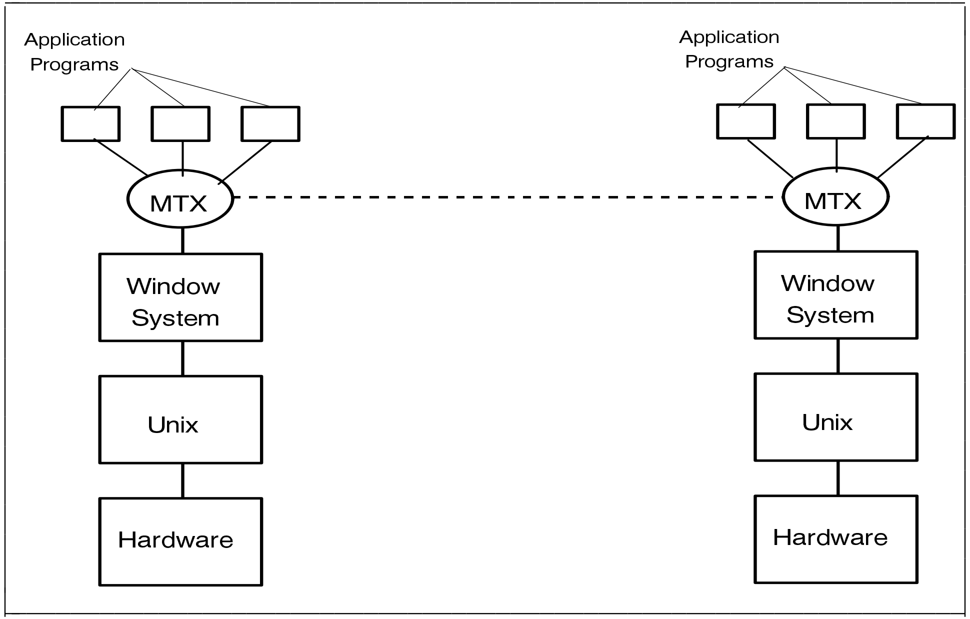


Figure 6. MTX relationship to other elements of the computing environment.

This extra step causes a decrease in the data throughput, compared to an ordinary pipe. Figure 7 shows an example of the performance decrease caused by running MTX. The tests are based on *cat*ting */etc/termcap* to one or two windows simultaneously, and counting the number of characters per second written to a window (on a SUN Microsystems model 3/60).

MTX performance (characters/second)		
Test	MTX	No MTX
1 window	3700	4700
2 windows	1900	2400

MTX provides a terminal emulator in each window it creates for use by programs. Each window emulates the full window system protocol, so any window system application can run on top of MTX transparently (including MTX). Application programs need not be aware they are running on MTX.

In addition to implementing the window system protocol in each window, MTX provides extensions to the protocol as a means for programs to learn about their connection state. An application program can request to be informed by MTX when a connection to (or from) it is added or removed. MTX will provide the name of the object at the other end of the connection, as well as its type, and the type of its connection (input or output). By using this information, application programs can coordinate sharing among themselves, the users and other workstations. MTX also supports application queries for connection information. Applications that use these protocol extensions can still run directly with the window system, as the invalid protocol requests that would be handled directly by MTX are interpreted as unimplemented requests by MGR, and harmlessly ignored.

When MTX is started for a new display, it sends out a broadcast message to inform other MTX's running on remote hosts of its existence. Included in this message are the local user and hostname, as well as a port number to use to request a connection. When a user requests a connection to another host, a stream connection is made to the remote MTX, using the information contained in the broadcast message, and all data and control information are sent over this connection. By implementing MTX as a distributed service, the bottleneck that would result from a single MTX server on the entire network is relieved.

Summary

MTX is a shell for Unix that permits the dynamic rerouting of pipe connections, the creation of multiple virtual terminals, and a more general mechanism for communication among processes than is typically found in the shell. In a network environment, MTX provides sharing of windows and processes among multiple computers and their displays. These capabilities permit the shell and other application programs to take advantage of the networking and windowing capabilities of modern UNIX systems.

References

[1] Ritchie, D. (1978), ‘‘A Retrospective’’ *Bell System Technical Journal,* 57 (6), pages 1947-1970.

[2] Bourne S. (1978), ‘‘The UNIX Shell’’ *Bell System Technical Journal,* 56 (6), pages 1971-1990.

[3] Korn, D. (1983), ‘‘KSH – a Shell Programming Language’’, *Proceeding of the Summer USENIX Conference,* pages 191-202.

[4] Joy, W. (1980), ‘‘An Introduction to the C Shell’’, *UNIX User's Manual, Supplementary Documents,* Berkeley Ca.

[5] Rochkind, M. (1980), ‘‘2dsh - An Experimental Shell for Connecting Processes With Multiple Data Streams’’, Bell Labs Technical Memorandum 80-9323-3

[6] McDonald, C. and Dix, T. (1988), ‘‘Support for Graphs of Processes in a Command Interpreter’’, *Software Practice and Experience* 18 (10) pages 1011-1016.

[7] Haeberli, P. (1986), ‘‘A Data-flow Environment for Interactive Graphics’’, *Proceeding of the Summer USENIX Conference* pages 419-428.

[8] Karels, M. and Leffler, S. (Ed.) (1984), *Unix User’s Manual Reference Guide,* Berkeley, Ca.

[9] Da Cruz, F. (1987)*, Kermit, a File Transfer Protocol,* Bedford Mass: Digital Press

[10] Uhler, S. A. (1987), ‘‘MGR - C Language Application Interface’’, Bellcore Technical Memorandum TM-ARH-010796