Appendix B DTM Documentation

Overview

This appendix outlines the Data Transfer Mechanism (DTM) as written by its creator, Jeff Terstriep, at the National Center for Supercomputing Applications. Refer to Appendix E for a list of standard DTM Routines. The DTM documentation is also available via FTP. Refer to Appendix C, "Obtaining NCSA Software," for those instructions.

Introduction

DTM is a message passing facility. It is designed to facilitate the creation of sophisticated distributed applications. To do this, DTM provides a method to interconnect applications at run-time, a reliable message passing with synchronization and transparent data conversion. DTM has been optimized for large messages containing from 100 Kbytes's to several megabytes, but it's effective for smaller messages as well.

The DTM message is an abstract class and no true instances of this class ever exist. Rather, all messages are instances of a sub-class to the DTM message. These classes inherit the ability to be sent or received between applications from the super-class, but typically add their own specialized functions to access the data. Several predefined classes are in use at NCSA[1], but programmers are free to define their own for special applications.

All messages are exchanged through DTM ports. A *DTM port* is a logical unidirectional communications channel. Applications may define as many DTM ports as are appropriate for their function. For example, a simulation may define an output port for each dataset it produces, filter programs may have one input and one output port and a viewing program may have an input port for each window or object. DTM port connectivity is typically listed on the command line, allowing the interconnections to be defined at run-time.

[1]See the write-up on Multi-Dimensional Data sets (MDD) and Surface Description Language (SDL).

DTM Messages

The DTM message has two parts, a header and a data section. Two features distinguish the sections. First, the *header* is sent or received in its entirety on the call to `DTMbeginWrite` or `DTMbeginRead`. Secondly, no data conversion is provided for the header. The DTM library assumes the header consists of unsigned bytes.

Since data conversion is not provided for the header, the programmer must be prepared to make the header machine independent if the message will travel between architectures. The easiest method is to create the header as an ASCII string[2] or as an XDR buffer[3].

DTM messages should be self describing. The header is designed to contain information about the attributes of data stored in the data section. This information may include the class of the message, the type of the data (`char`, `int`, `float`), a title or any other information relevant to the data section.

The *data section* can be thought of as a delimited stream of elements. Elements are generally primitive data types such as characters, integers or floating-point numbers, although more complex types are possible. The stream is delimited, so the application may receive up to the end of a message, but may not continue without receiving the end of message mark with `DTMendRead`.

Because each message is delimited, an application need not know the number of elements that will be contained in the message apriori. An application that is writing a message may call `DTMwriteDataset` as many times as desired within the message. A receiving application may call `DTMreadDataset` a often as necessary to receive the entire message. The buffer size, or the number of elements sent or received at one time, may differ and each application may choose of size that is appropriate for its task.

Both the header and data sections of a message are optional. Many control message only send the header. And it is possible to have applications that communicate using only the data section. Hence, the smallest legal DTM message is two 4 byte integers, both are zero indicating the header length is zero and the data length is zero.

Since both sections are optional, many applications may decide to ignore either the header or data section when receiving a message. To keep the stream consistent, any data remaining in the header is discarded after the call to `DTMbeginRead`. Similarly, any data in

[2]The NCSA provided message classes MDD and SDL use ASCII headers.
[3]See RFC-1014 for more information about XDR.

the data section is discarded when the message is finished with a call to `DTMendRead`.

DTM Ports

A *DTM port* is a unidirectional synchronized communication channel through which DTM message may be sent or received. DTM ports are based on a reliable communication service such as TCP/IP and have been implemented on UNIX machines on top of the Berkeley sockets. In the current version, each DTM port corresponds to a TCP/IP connection.[4]

DTM ports are created with a call to `DTMmakeInPort` or `DTMmakeOutPort`. Both calls requires a DTM port address, the format of which is **"**`hostname`**:**`port`**"**. For output ports hostname represents the host where the data will be sent. The hostname is optional, if it is missing the local host name is assumed. For input ports, the hostname is always replaced with the name of the local host.

The port number represents the TCP port number to be used. For output ports, this number represents the port where an application will attempt to connect. For input ports, the port is where the system will listen for incoming connections.

DTM messages are sent by calling the `DTMbeginWrite` and `DTMendWrite` pair. Similarly, DTM messages are received by calling `DTMbeginRead` and `DTMendRead`. After each message is received, an acknowledgement is returned to the sender. If the sender attempts to write a new message before the acknowledgement has been returned, it will block.

This acknowledgement system is equivalent to setting the message queue length to one. Limiting the number of pending messages was done for two reasons. First, because DTM was designed to support interactive applications. Allowing only one message to be buffered reduces the latency a user will have between altering a parameter and perceive the results. Secondly, DTM messages frequently hold multidimensional arrays of floating-point numbers. It is not unusual for these messages to be several megabytes in size. Buffering several of these messages would strain system memory and could cause thrashing.

An application may define a DTM port for each class of message it will send or receive, this is known as port level multiplexing. Port level multiplexing is most effective for output ports. For example, if an application is going to produce three types of datasets, providing a port for each type will allow each dataset to be routed to separate applications. The datasets can then be operated on in parallel. This arrangement is more efficient than serializing the

[4]Future releases of the DTM library will remove this restriction.

messages down a single port since each application must examine all messages and copy messages not intended for it to its output for other applications to work on.

In contrast, input ports seem to be most effective when they are not types according to the message they expect to receive. Rather, input ports should be treated identically and each message should be examined and handled correctly based on the message's class and the information it contains. This is known as *message level multiplexing*.

Message level multiplexing works well with DTM messages since only the header is returned on the call to `DTMbeginRead`. The header may be examined to determine the message class and other relevant information. After the header has been decoded the appropriate routine may be called to receive and process the data section of the message.

DTM Applications

DTM applications typically receive connectivity information from the command line. The DTM port address is preceded on the command line by the flag **"**`-DTMIN`**"** for input port or **"**`-DTMOUT`**"** for output ports. Application should follow this convention since it will make invocation easier for users and for automatic configuration managers.

As stated above, the DTM port address should be specified at run-time and not hard-coded into the application. There are two exceptions to this rule. The first case is a server which is designed to listen at a well known address. Typically, a server of this type will register itself in the system services table[5].

The second case, has to do with the special DTM port address ":0". When this address is used, the system will assign an unused TCP port number. The application may retrieve the new DTM port address and specify it on the command line when invoking other applications, register it with a name server or otherwise communicate it to other applications.

Since no special priority is assigned to DTM applications they may be started in any order. If an application attempts to read from a port before a writer has connected, it will block and wait for a connection. If an application attempts to write to a port before a reader is listening, it will loop attempting to make connection once a second. In both cases a time-out will occur and an error will be returned after two minutes.

[5]Under UNIX this would be in /etc/services.

DTM Networks

DTM networks are multiple DTM applications connected and working in harmony. A simple example of this would be an application split into two parts. The "front-end" would be running on the user's workstation and handle user interaction and graphical output. The "back-end" could run on a supercomputer and provide number crunching capabilities.

More complex networks are possible. DTM has a feature that allows running networks to be changed. During each call to `DTMbeginRead`, the port check for new connections. If a new connection is pending the old connection is closed and a message is read from the new connection. This reconnection strategy is known as "bumping".

In addition to specifying connectivity at run-time, at any time a new process may be started that 'bumps' an old process. The ability to reconnect dynamically allows new configurations of modules without the necessity of killing the old configuration and starting a new one. Reconfiguring in this manner is important in a shared environment to prevent all users from perceiving a "glitch" when the configuration is changed.

Bumping is used to re-configure running networks of applications. For example, one view controller may be controlling several viewers running on various workstations. After a period of time, a user may wish to grab control of his viewer. By invoking his own view controller, he is capable of bumping the old view controller and taking command of his own viewer. The other viewer and users are unaffected by this re-configuration.