DTM

Data Transfer Mechanism

Jeff Terstriep
Research Programmer
Network Development Group, NCSA

December 10, 1990

1. 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 100KB's up to several megabytes, but is 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 the programmer is free to define his or her 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 data set 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, this allows the interconnections to be defined at run-time.

2. 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

---

[1] See the appendix on Scientific Data Sets (SDS) and Surface Description Language (SDL).

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

---

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

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

3. 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 "hostnameport". 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 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

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

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, since 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 data set, providing a port for each type will allow each data set to be routed to separate applications. The data sets can then be operated on in parallel. This is more efficient than serializing the 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.

4. 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. DTM Networks

DTM networks are multiple DTM applications connected and working in harmony. A simple example of this would be an

---

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

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 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.

Standard DTM Routines

int **DTMmakeInPort**(*portname*)
char *portname;

DTMmakeInPort creates an input port. *Portname* is pointer to a string with the format 'hostnameport'. 'Hostname' is optional and will always be replaced with the local host's name. *Portname* represent the address where the system will listen for incoming messages. If *portname* is '0' then the system will assign the TCP port number, the value can be retrieved with **DTMgetPortAddr** (see below).

If **DTMmakeInPort** suceeds, it returns a *portid*. The *portid* is a small integer used to refer to the port in all subsequent DTM calls. If there is any problem **DTMmakeInPort** will return **DTMERROR**.

Possible errors are returned by **DTMmakeInPort**

DTMNOPORT	No more open DTM ports.
DTMMEM	Insufficient memory for port.
DTMHUH	Illegal port name.

int **DTMmakeOutPort**(*portname*)
char *portname;

DTMmakeOutPort creates an output port. *Portname* is pointer to a string with the format 'hostnameport'. *Portname* represents the address where outgoing messages will be sent. Therefore, 'hostname' is any legal host name or IP address. 'Port' is a TCP port number where an application is listening, possably through the use of **DTMmakeInPort**.

If **DTMmakeOutPort** suceeds, it returns a *portid*. The *portid* is a small integer used to refer to the port in all subsequent DTM calls. If there is any problem **DTMmakeOutPort** will return **DTMERROR**.

Possible errors are returned by **DTMmakeOutPort**

DTMNOPORT	No more open DTM ports.

DTMMEM	Insufficient memory for port.
DTMHUH	Illegal port name.

int **DTMgetPortAddr**(portid, address, size)
int portid
char *address;
int size;

DTMgetPortAddr returns the IP address of DTM port. This is typically used in conjunction with **DTMmakeInPort("0")** to retrieve the TCP port number and report it to connecting programs.

Portid is value returned on a previous call to **DTMmakeInPort**. *Address* is a buffer where the address in the form 'hostnameport' will be stored. *Size* is the size of the *Address* buffer.

Possible errors are returned by **DTMgetPortAddr**

DTMPORTINIT	invalid value for *portid*.

int **DTMavailRead**(portid)
int portid;

DTMavailRead performs a non-blocking check for a message on the input port *portid*. **DTMavailRead** returns TRUE (1) if a message is available and FALSE (0) if not. **DTMavailRead** will return **DTMERROR** if a problem is encountered. Since **DTMERROR** also represents a TRUE value, an application can check for the possibility of an error by examining **DTMerrno**, for a non-zero state, after the call.

Possible errors are returned by **DTMavailRead**

DTMPORTINIT	invalid value for *portid*.
DTMSOCK	problem creating connection.

int **DTMavailWrite**(portid)
int portid;

DTMavailWrite performs a non-blocking check, on the output port *portid*, to determine if the receiving program has processed the previous message. **DTMavailWrite** returns TRUE (1) if a message is available and FALSE (0) if not. **DTMavailWrite** will return **DTMERROR** if a problem is encountered. Since **DTMERROR** also represents a TRUE value, an application can check for the possibility of an error by examining **DTMerrno**, for a non-zero state, after the call.

Possible errors are returned by **DTMavailWrite**

DTMPORTINIT	invalid value for *portid*.
DTMSOCK	problem creating connection.

int **DTMbeginRead**(portid, header, size)
int portid;
char *header;
int size;

DTMbeginRead receives a message from the input port *portid*. The message header is placed in the buffer *header*. If no message is currently available, this call will block. A non-blocking check for a pending message may be performed with DTMavailRead (see above).

Size indicates the size of the buffer allocated to hold the incoming header. **DTM_MAX_HEADER** is defined to be the largest legal header length and may be used to allocate the header buffer. If the incoming header is larger than the header buffer, **DTMbeginRead** will fill the header buffer, discard the remaning header and return **DTMERROR**. In this case **DTMerrno** will be set to **DTMHEADER**.

Possible errors are returned by **DTMbeginRead**

DTMPORTINIT	invalid value for *portid*.
DTMSOCK	problem creating connection.
DTMREAD	problem reading from connection.
DTMHEADER	incoming header exceeds buffer size.

int **DTMbeginWrite**(portid, header, size)
int portid;
char *header;
int size;

DTMbeginWrite writes the header of a message to the output port *portid*. If the previous message has not been received this call will block. A non-blocking check to determine if the previous message has been received is available with **DTMavailWrite** (see above).

Header is a buffer containing the header of the message to be written. *Size* is the length of the header, it may be calculated with **DTMheaderLength**(*header*).

Possible error condition from **DTMbeginWrite**

DTMPORTINIT	Invalid value for *portid*.
DTMSOCK	Problem creating connection.
DTMTIMEOUT	Time-out waiting for receiver.
DTMWRITE	Error writing header.

int **DTMrecvDataset**(portid, buffer, num_elements, type)
int portid;
char *buffer;
int num_elements;
DTMTYPE type;

DTMrecvDataset reads the data section of a message from the input port *portid*. This call is optional, if it is used it must be preceeded by a call to **DTMbeginRead**. **DTMrecvDataset** will attempt to fill the buffer with number of elements of the specified type, automatic type conversion will be performed where necessary. *Buffer* is assumed to be large enough to hold the amount of data requested.

In the absence of errors, **DTMrecvDataset** returns the number of elements actually read. The process may call **DTMrecvDataset** as often as required to receive the message in its entirety, the value returned from **DTMrecvDataset** will equal 0 at the end of the message.

Possible error conditions from **DTMrecvDataset**

DTMCALL	**DTMbeginRead** must preceed this call.
DTMREAD	Error reading message.

int **DTMsendDataset**(portid, buffer, num_elements, type)
int portid;
char *buffer;
int num_elements;
DTMTYPE type;

DTMsendDataset writes the data section of a message to the output port *portid*. This call is optional, if it is used it must be preceeded by a call to **DTMbeginWrite**. **DTMsendDataset** will write the number of elements of the specified type from the buffer, automatic type conversion will be performed where necessary. **DTMsendDataset** may be called as often as necessary to complete the message.

Possible error conditions from **DTMsendDataset**

DTMCALL	**DTMbeginWrite** must preceed call.
DTMWRITE	Error writing message.

int **DTMendRead**(portid)
int portid;

DTMendRead marks the end of the current message and prepares for the next message on the input port *portid*. Any data remaining in the message is discarded. There must be a matching **DTMendRead** for every call to **DTMbeginRead**.

Possible error conditions from **DTMendRead**

DTMCALL	**DTMbeginRead** must preceed call.

int **DTMendWrite**(portid)
int portid;

DTMendWrite marks the end of the current message. There must be a matching **DTMendWrite** for every call to **DTMbeginWrite**.

Possible error conditions from **DTMendWrite**

DTMCALL **DTMbeginWrite** must preceed call.

int **DTMdestoryPort**(portid)
int portid;

DTMdestroyPort closes all connections associated with the port *portid* and frees the entry in the port table. This call, although optional, is recommended since it may assist connected processes in proceeding correctly.

Possible error conditions from **DTMdestroyPort**

DTMPORTINIT Invalid *portid*.