The DataExchange Library

Greg Eisenhauer
eisen@cc.gatech.edu

Beth Schroeder
beths@cc.gatech.edu

GIT-CC-96-17
College of Computing
Georgia Institute of Technology
Atlanta, Georgia 30332

1 Introduction

DataExchange is a library of communications routines which manage the complexity of systems with multiple communication links between heterogeneous machines. The library is designed to be used as an implementation basis for networks of agents communicating application-specific data. As such, it contains support for establishing communication between agents, resolving differences between data formats used by agents, forwarding data from agent to agent and processing of data within an agent. This paper details the services and interfaces offered by DataExchange.

2 The Basics

The purpose of DataExchange is to ease the task of creating and initializing networks of entities communicating via sockets. It is built on top of PBIO, a lower-level communications library that supports binary transmission of C-style data structures between heterogeneous machines. PBIO is documented in Portable Self-Describing Binary Data Streams [1]. This document is available in the PBIO source distribution and an older version is Georgia Institute of Technology College of Computing Technical Report GIT-CC-94-45. Many of the concepts in PBIO are reflected in DataExchange, so familiarity with PBIO is useful for understanding DataExchange. While PBIO provides the basic point-to-point data transmission mechanisms, DataExchange augments that with support for establishing connections, managing data distribution and other coordination between multiple PBIO connections.

2.1 Clients and Servers

Colloquially we understand differences in the behavior of clients and servers. Clients are more likely to initiate connections to servers rather than vice versa. Servers are more likely to be event-driven and respond to requests, while clients may have a variety of control styles. In some applications, the distinction between client and server blurs. While the DataExchange library is fundamentally peer-to-peer, programs that we might call clients or servers tend to use different parts of the library and have different principal concerns. Because of this, and to provide a simple context in which to explain DataEx-

Figure 1: Simple hub-and-spoke client-server model.
#include "DE.h"

void

main()

{
    DExchange de = DExchange_create();
    if (DExchange_listen(de, 12345) == -1) exit(-1);
    while (1) {
        DExchange_poll_and_handle(de, TRUE);
    }
}

Figure 2: The basicDataExchange server

change functionality, we’ll call a DataExchange program a ‘client’ if it wants to have only a single network connection at a time, and call it a ‘server’ if it wants to allow the possibility of multiple connections. In particular, in examples we’ll use a simple hub-and-spoke model as shown in Figure 1 where the term ‘server’ refers to the centrally located DataExchange program which communicates with multiple DataExchange ‘client’ programs. In more complex applications those clients may themselves act as servers hosting other clients.

The code in Figure 2 presents a simpleDataExchange server program. This program uses three basicDataExchange functions:

DataExchange_create() – This call creates and initializes aDataExchange data structure. AllDataExchange applications must call this at least once. TheDataExchange data structure provides a means of associating related communication channels.

DataExchange_listenDataExchange de, int port_num) – This call requests DataExchange to create an internet socket on the current host, bind it to IP port port_num and begin listening for connections. Once DExchange_listen() has been called, other programs can initiate a connection with thisDataExchange by calling DExchange_initiate_conn(), discussed further below. A particularDataExchange can only listen on one IP port.

int DExchange_poll_and_handleDataExchange de, int block) – DExchange_poll_and_handle() is the basic event handling call inDataExchange and is analogous to the XtAppNextEvent() / XtAppDispatchEvent pair in X windows toolkit programming. ADataExchange event is a basic communications occurrence, such as a connection request or message arrival. The routine DExchange_poll_and_handle() queries the communication channels associated with DExchange de and processes events from each connection which has input ready. It also checks for connections requests on the port specified by DExchange_listen() and handles those requests. The block parameter specifies whether or notDataExchange should block (suspend program execution) waiting on an event if none is ready. The integer return value (ignored in this program) is the number of events processed by the call.

This simple program performs essentially as a data distribution hub. It accepts connections from otherDataExchange programs and forwards any data it receives from any client to each of the other clients. This automatic data forwarding to other connected clients is the default DataExchange behavior. (See Section 5 for more information on controlling data flow.) The IP port number given
```c
#include <DE.h>
main()
{
    DExchange de = DExchange_create();
    DEPort dep;
    dep = DExchange_initiate_conn(de, "host.domain.org", 12345, FALSE);
    DEPort_close(dep);
    DExchange_close(de);
}
```

Figure 3: A simple client program.

In the `DExchangeListen()` call can be a well-known port number (as the 12345 in this example), or if the parameter is 0, DataExchange will arbitrarily select a free port number. The port number selected is available via the `int DExchange_inet_port(DataExchange de)` function.

A correspondingly simple client program that connects to this server is given in Figure 3. Unlike the sample server program, this client program doesn’t really do anything useful. Assuming that the server program is running on the machine “host.domain.org” and listening at the well-known port number 12345, this client connects to the server, then immediately shuts down the connection (with DPort_close(dep) and shuts down the entire DataExchange with DExchange_close(de). The return value from DExchange_initiate_conn() is a value of type DEPort and is a handle that can be used to write or read to or from the remote DataExchange on the other end of that particular network connection. The last parameter to DExchange_initiate_conn() is the block_by_default value. A DataExchange client has the ability to refuse (or block) data of particular types. Enabling blocking of a particular format prevents the forwarding mechanisms of the DataExchange at the other end of the connection from sending data of that format to this client. A value of FALSE tells the remote server that its default policy should be not to block delivery of any data to this client. Section 5 explains more about forwarding and blocking, the basic mechanisms for controlling data movement in DataExchange.

These examples also introduces the two most important data types in DataExchange, DExchange and DEPort. A DExchange value essentially represents a communications context for a program. DataExchange subroutines which operate on a communications context have a “DExchange_” prefix in their name and take a DExchange value as their first parameter. This paper will refer to a DExchange value as “a DataExchange.” Programs can create multiple DExchanges and operations those will be independent. DEPort values are associated with a DExchange and represent the endpoint of a bidirectional communications link. An arrangement of DExchanges, DEPorts and communications links is shown in Figure 4. A DataExchange acting as a client with a single communications connection will have only one DEPort. A DataExchange operating as a server with many connections will have many DEPort’s, one for each client to which it is connected. DataExchange communications links have a DEPort on each end. DataExchange’s explicit read and write subroutines for data operate on DEPort values and send and receive data across the communications link represented by their DEPort parameter. The next section will examine DataExchange’s data handling facilities in more detail.
3 Receiving and Sending Data

DataExchange offers both implicit and explicit mechanisms for sending and receiving data. A previous section mentioned DataExchange’s implicit default behavior of re-sending (or forwarding) data received from one client to all other connected clients. Section 5 will explain how this implicit re-send mechanism can be more finely controlled. The implicit receive occurs when an application calls DataExchange.poll_and_handle() and will be described in more detail in Section 4.

Explicit send and receive operations are provided by the routines DataExchange_write_data() and DataExchange_read_data() which have profiles:

```c
int DataExchange_write_data(DataExchange dep, int format_id, char *data);
char *DataExchange_read_data(DataExchange dep, int *format_id);
```

These calls read and write directly to the DataExchange specified by the dep parameter. The data parameter to DataExchange_write_data() and the return value from DataExchange_read_data() are pointers to buffers containing data to be read or written. The format_id and format_id_p parameters identify the type of data that has been read or is to be written. How these values are derived requires a bit more explanation.

We have mentioned previously that DataExchange uses PBIOS as a point-to-point transport mechanism for data. Like PBIOS, DataExchange requires that data record formats be registered before data can be sent or received by the application. Record format descriptors are described in the PBIOS documentation, but basically consist of a list of field descriptions, where each field description is a quadruple giving the field name, data type, size and offset within the record. Given this information, PBIOS can pack the record for transmission to other machines and can decode it despite differences in machine architecture or record layout. In DataExchange, record formats are registered once only and the DataExchange library takes care of registering the format to the individual connections. Formats are registered using the routine DataExchange_register_format(DataExchange de, char *format_name, I0FieldList field_list). Each format is assigned a format ID which can be retrieved with the call int DataExchange_get_format_id(DataExchange de, char *format_name). It is this format ID that must be given in the DataExchange_write_data() call to identify the type of data being
```c
#include <DE.h>
void main()
{
  DEExchange de = DEExchange_create();
  DEPort dep;

  DEExchange_register_format(de, "my format", field_list);
  dep = DEExchange_initiate_conn(de, "host.domain.org", 12345, FALSE);
  while (1) {
    rec_ptr recv;
    int rcvd_format;
    recv = (rec_ptr) DEport_read_data(dep, &rcvd_format);
    printf(" Received %d, %s\n", recv->integer_field, recv->string_field);
  }
}
```

Figure 5: A simple read client using an explicit read.

written. Upon return from DEport_read_data(), the int pointed to by format_id_p is filled in with the format ID corresponding to the record which was read. For convenience, though it is slightly slower, there is also a DEport_write_data_by_name() which takes a char *format_name parameter instead of int format_id. Given these calls, we can construct two simple clients to connect to our basic server. One of these clients will do nothing but write records to the server and the other will only read records. Both clients will use the following record structure and field list:

typedef struct _rec {
  int integer_field;
  char *string_field;
} rec, *rec_ptr;

static IOField field_list[] = {
  {"integer_field", "integer", sizeof(int), IOOffset(rec_ptr, integer_field)},
  {"string_field", "string", sizeof(char*), IOOffset(rec_ptr, string_field)},
  {NULL, NULL, 0, 0}};

The client code is given in Figures 5 and 6. These clients should be able to exchange their messages in binary form across Internet-connected machines regardless of the underlying machine architectures. If the read client program connects to the sample server first it will receive all the messages written. If it connects in the middle of the write programs transmission it will receive whatever messages remain. If multiple read clients connect they will each receive the messages the write client sends. If multiple write clients connect the connected read clients will receive messages from each of them.

For compactness, most error handling has been left out of the code given in Figures 5 and 6. More robust versions of these programs are provided in Appendix A. Notice that the programs differ in the last parameter to DEExchange_initiate_conn(). The write program requests that no data be forwarded to it by default (the TRUE value for block_by_default) and the read program requests that all data be forwarded by default (the FALSE value). Also notice that the reading program ignores the record format ID that is set in rcvd_format. This is a point of non-robustness of the program. If it were to receive another type of message, the value returned by DEport_read_data()
#include <DE.h>
void main()
{
    DExchange de = DExchange_create();
    DEPort dep;
    int i, rec_format_id;

    DExchange_register_format(de, "my format", field_list);
    rec_format_id = DGet_format_id(de, "my format");
    dep = DExchange_initiate_conn(de, "host.domain.org", 12345, TRUE);
    for (i = 0; i < 10; i++)
    {
        rec buf;
        buf.integer_field = i; buf.string_field = "Hello World!";
        DEport_write_data(dep, rec_format_id, (char*)&buf);
        sleep(1); /* take some time */
    }
}

Figure 6: A simple write client using an explicit write.

would be a pointer to a different data type and the printf() would generate trash or fail.

4 Action Routines

The routines described above are sufficient for simple applications. But a more realistic application than the simple server described in Figure 2 has more complex needs. A server may need to handle multiple clients wishing to establish connections simultaneously. The server will seldom know upon which connection data will next arrive. Additionally many applications accept data of various formats and take separate actions depending on the data format. Dataexchange provides a solution to handling multiple data formats that is more elegant than a large switch statement. The solution to these and other problems is provided through the mechanism of action routines, routines to be called when data arrives or a connection is established or shutdown.

4.1 Data Handling Action Routines

For data handling, DataExchange provides two types of action routines, filters and functions. Functions are called after any implicit data forwarding. (Data forwarding is described in detail in Section 5.) Filters are called before data forwarding and can return a value which preempts further processing. The calls for registering and unregistering such routines have the following prototypes:

typedef int (*DataHandlingFunc) (DExchange de, DEPort dep, int format_id,
    void *data, int data_length);

void DExchange_register_function(DExchange de, char *format_name,
    DataHandlingFunc func);
int DExchange_unregister_function(DExchange de, char *format_name,
    DataHandlingFunc);
void DExchange_register_filter(DExchange de, char *format_name,
    DataHandlingFunc);
int DExchange_unregister_filter(DExchange de, char *format_name,
    DataHandlingFunc);
Registered functions and filters are associated with specific data formats, though the same subroutine can be registered as a handler for multiple data formats. Action routines may be registered withDataExchange at any time. The register functions check for duplication (an attempt to register a handler routine which has already been registered for that format) and ignore additional registrations.

These functions allow us to rewrite the read client as shown in Figure 7. This version will behave just like the previous version except that the handler routine will be called only when data of format "my format" arrives. Other messages will simply be ignored.

The read client in Figure 7 is interesting because it uses a server-like programming style despite having only the one initiated by the DExchange_initiate_conn() call. However, the basic program can easily be made more complex. For example, adding another DExchange_initiate_conn() to the body yields a program which initiates a connection with two other DataExchanges and prints all the "my format" data received from them. Adding a call to DExchange_listen() produces a program which will also accept connections initiated by other DataExchanges and print any "my format" data received from them as well. Bodies for these programs are shown in Figures 8 and 9.

```c
#include <DE.h>
int handler(de, dep, format_id, data, data_length)
DataExchange de;
DEPort dep;
int format_id;
void *data;
int data_length;
{
  rec_ptr recv = (rec_ptr) data;
  printf("Received %d, %s\n", recv->integer_field, recv->string_field);
  return 1; /* return value is ignored for functions */
}

void main()
{
  DExchange de = DExchange_create();

  DExchange_register_format(de, "my format", field_list);
  DExchange_register_function(de, "my format", handler);
  (void) DExchange_initiate_conn(de, "host.domain.org", 12345, FALSE);
  while (1) {
    DExchange_poll_and_handle(de, TRUE);
  }
}
```

Figure 7: A read client using an action routine to handle a single connection.
void main()
{
    DExchange de = DExchange_create();
    DExchange_register_format(de, "my format", field_list);
    DExchange_register_function(de, "my format", handler);
    (void) DExchange_initiate_conn(de, "host.domain.org", 12345, FALSE);
    (void) DExchange_initiate_conn(de, "host2.domain.org", 45678, FALSE);
    while (1) {
        DExchange_poll_and_handle(de, TRUE);
    }
}

Figure 8: The body of a read client which starts and handles two connections.

void main()
{
    DExchange de = DExchange_create();
    DExchange_register_format(de, "my format", field_list);
    DExchange_register_function(de, "my format", handler);
    (void) DExchange_initiate_conn(de, "host.domain.org", 12345, FALSE);
    (void) DExchange_initiate_conn(de, "host2.domain.org", 45678, FALSE);
    if (DExchange_listen(de, 31415) == -1) exit(-1);
    while (1) {
        DExchange_poll_and_handle(de, TRUE);
    }
}

Figure 9: The body of a read client with multiple connections.
4.2 Other Action Routines

In addition to action routines associated with data arrival, applications may register functions to be called when a new format or a comment\(^1\) arrives on some connection, when a connection is established or when it is closed. The prototypes for these functions are:

```c
typedef void (*FormatHandlingFunc)(DataExchange de, DEPort dep, char* format_name);
typedef void (*CommentHandlingFunc)(DataExchange de, DEPort dep, char* comment);
typedef void (*OpenCloseHandlingFunc)(DataExchange de, DEPort dep);
```

```c
voidDataExchange_register_format_handler(DataExchange de, FormatHandlingFunc funct);
voidDataExchange_register_comment_handler(DataExchange de, CommentHandlingFunc funct);
voidDataExchange_register_close_handler(DataExchange de, OpenCloseHandlingFunc funct);
```

For each of the register routines there is a corresponding unregister routine with identical parameters. The unregister routines return a value of −1 if the routine specified was not already registered and 1 otherwise.

5 Controlling Data Movement

As mentioned previously, the default DataExchange behavior is to resend all data received from one connection to all of its other connections. This mechanism is useful for constructing interesting servers and managing data flow in networks of communicating DataExchanges. Resending potentially occurs in all DataExchanges, but it is obviously less interesting in DataExchanges with only one connection, such as the classic “clients” described in Section 2.1. The DataExchange library allows control of this resending in ways which can be loosely divided into record forwarding and record blocking controls. Record forwarding and blocking controls both affect the implicit resending of received data, but in different ways. Forwarding controls are available to the DataExchange performing the resending operation and allow that resending to be controlled on a per-format basis. Forwarding controls are therefore local controls in the sense that a DataExchange calling those routines affects its own resending. The complementary record blocking controls are instead used by the DataExchanges to whom data is being resent to affect the type of data that is resent to them. In particular, by using the record blocking controls a DataExchange can tell another DataExchange not to send it records of a particular format. Record blocking controls thus affect resending on a per-connection basis, but they differ from the forwarding controls in that blocking routines invoked by one DataExchange affects resending that occurs in other DataExchanges. In this sense, record blocking is a remote operation and is useful to any DataExchange, be it client or server, that wants to influence what data is sent to it by others to whom it is connected. On the other hand, record forwarding controls affect resending in the DataExchange invoking the controls and are thus a local operation.

In general, the record forwarding and blocking functionality is available to control data flow between any two DataExchanges, but it is easiest to understand in the context of the simple client-server situation of Figure 2.1. Here, the data forwarding controls are of most interest to the central server because that DataExchange has multiple connections and is in a position to resend the data it receives from particular clients to its other clients. However, each client could use the record

---

\(^1\)Details on the nature of formats and comments can be found in [1].
blocking controls to keep the server from sending them data of particular formats.\textsuperscript{2} In both record forwarding and blocking, DataExchange’s decision to send or not send a particular record to a particular client is based solely on the type or format of the record.

5.1 Data Forwarding

The two forwarding control routines are:

```c
typedef enum _DEForwardStyle {
    DENever_Forward, DEAIl_But_Sender, DEForward_To_AIl
} DEForwardStyle;

extern void DExchange_set_forward(DExchange de, DEForwardStyle value);
extern void DExchange_set_format_forward(DExchange de, char *format_name,
DEForwardStyle value);
```

`DExchange_set_forward()` sets the default forwarding value assigned to new formats in the DataExchange `de`. Newly created DataExchanges have forwarding set to `DEAll_But_Sender`, which specifies that all incoming data is to be resent on all connections except for the one upon which it was received. The other two possible values are `DENever_Forward` which specifies that no automatic data forwarding is to be done, and `DEForward_To_AIl` which specifies automatic forwarding to all connections, including resending the data to the DataExchange program from which it was received. Applications which wish to change the default will generally call this routine to set the default forwarding state for all formats before initiating any connections or registering any formats. `DExchange_set_format_forward()` sets the forwarding state for the named format to one of the three values described above. This routine can be called and the forwarding flag for a format set at any time. In particular it can be called before the named format has been registered or seen on an input connection.

5.2 Data Blocking

The record blocking API allows the DataExchange on one end of the connection to influence the types of data sent to it by the DataExchange on the other end of the connection. Each DEport connection in each DataExchange has a set of boolean flags, one per format which give the blocking value for that format on that connection. Outgoing records which otherwise would be sent to the connection by the forwarding rules are not sent if the blocking value for that format is TRUE. Each end of a connection has its own blocking table, as is depicted in Figure 10. In this Figure, the flags in DEport M in DataExchange A specify that no records of format x are to be sent to DEport N. The flags in DEport N specify that DEport M should receive records of formats s, v and x, but not t. The flags in DEport N which control what records are sent to DEport M are set by operations performed on DEport M in DataExchange A. Specifically, the routine `DEport_set_format_block(DEport dep, char *format_name, int blocking_value)` performed on DEport M sets the blocking value for a particular format in DEport N. Like the routine `DExchange_set_format_forward()`, `DEport_set_format_block()` can be called at any time during the life of a connection, including before a format has been registered or seen on an incoming connection. A call to `DEport_set_format_block()` affects only the blocking flags in the DEport

\textsuperscript{2}While clients always have the option of simply ignoring data they don’t care about, record blocking allows them to request that those messages never be sent, sparing the system the overhead of sending them.
which is connected to the one specified in the first parameter. It has no direct effects local to theDataExchange which calls it. The prototypes for the blocking controls are given below:

```c
extern void DEport_set_format_block(DEPort dep, char *format_name, int blocking_flag);
extern voidDataExchange_set_block_default(DataExchange de, int blocking_flag);
```

If not explicitly set, the blocking flag for newly registered or received formats is assigned from a default value which is determined at the time the connection is initiated. BecauseDataExchange connection initiation is asymmetric, the mechanism through which the default value for the blocking flags is controlled is also asymmetric. For the initiator of a connection, the default blocking value is specified in the `DataExchange_initiateComm` call which sets up the connection. The value of the `block_by_default` parameter in this call gives the default value of the blocking flag for data of all formats which the target DataExchange might send to the DataExchange initiating the connection. Specifically, if DataExchange A in Figure 10 initiated the connection to DataExchange B, the value of the `block_by_default` flag in the `DataExchange_initiateComm` call it makes provides the default value for the blocking flags in DEport N. The default blocking flag on the acceptor's side of the connection (DEport M default if A initiates) is set by the `DataExchange_set_block_default` call and is FALSE for newly created DataExchanges.

Essentially, an application can use a combination of values in the initiation call and in the `DEport_set_format_block` call to tell the remoteDataExchange “Send me nothing except formats “A”, “B” and “C”” or “Send me everything except formats “X”, “Y” and “Z””. In combination with the `DataExchange_register_format_handler` facility, the decision to accept or reject records of a particular format can be made dynamically.

The record blocking facility allows us to rewrite the body of the read client previously shown in Figure 7 into a more efficient form as shown in Figure 11. This code differs from the previous version only in that value of the `block_by_default` parameter in the `DataExchange_initiateComm` call tells the remoteDataExchange to forward nothing to this connection by default and the additional call to `DEport_set_format_block()` then enables sending of data of the format “my format”. The
void main()
{
    DEExchange de = DEExchange_create();
    DEPort dep;

    DEExchange_register_format(de, "my format", field_list);
    DEExchange_register_function(de, "my format", handler);
    dep = DEExchange_initiate_conn(de, "host.domain.org", 12345, TRUE);
    DEPort_set_format_block(dep, "my format", FALSE);
    while (1) {
        DEExchange_poll_and_handle(de, TRUE);
    }
}

Figure 11: A more efficient read client using format blocking.

Clients of Figures 7 and 11 are functionally identical, but where the earlier version would ignore data that was not of a type it understood, the latter version will never be sent data that it cannot interpret.

In order to allow an application to generate data to be distributed via the forwarding rules, DataExchange supplements the routine DEPort_write_data() with DEExchange_forward_data(), which has the prototype:

    extern void DEExchange_forward_data(DEExchange de, DEPort dep, int format_id, void *data);

This routine causes data to be written to the connections of de as if it had come from port dep. (Which port data comes from is only relevant if the format forwarding style is DEAll_But_Sender. If the dep parameter is NULL, no port will be excluded by the DEAll_But_Sender setting.)

DataExchange's decision to send or not send a particular record to a particular client is based solely on the type or format of the record. Clients can implement content-based forwarding decisions by disabling automatic forwarding and making their own decisions about whom to send data within filters or functions (described in Section 4). We envision that at some point DataExchange will provide additional dataflow control mechanisms, such as mechanisms to accommodate indirect one-to-one communication through intermediate DataExchanges acting as "routers". However only the basic record forwarding and blocking interfaces have been finalized.

6 Accessing Data without Registered Formats

In the previous examples, we have implicitly assumed that applications have compile-time knowledge of the data structures that they want to actually process (as opposed to simply forward). These applications can register the structure of the data formats that they are interested in and DataExchange provides them with the data in that format. However, some applications might not have the luxury of compile-time knowledge.

To create the worst-case scenario, imagine an application that wants to communicate with several clients and forward all data received from them to a third party without loss or truncation of data. Further, it wants to establish a partial order amongst the various data items based on the
values of certain fields in the data. This type of application might arise in a distributed monitoring system where a causal ordering must be enforced in possibly misordered monitoring events. This type of application, which must access data in records it has little knowledge of and which must store that data in binary form for a time before sending it out, requires much more complex support than applications not having this requirement.

DataExchange supports this type of application with some implicit behavior and several sets of routines. First, if the application does not register a specific data format for a record type, DataExchange will automatically select a record format that allows it to forward records without data loss. This process involves selecting the largest size used by any client for each field in each record format. So, for example, if one client is using a 2-byte integer for the field “sequence number” in record “basic event” while another client uses an 4-byte integer for the same field, DataExchange will use a 4-byte integer for its internal record handling and output format. If a new client connects and registers the same record type “basic event” with a larger size for the “sequence number” field, DataExchange will begin using the larger size for internal handling and output.

While this dynamic field sizing behavior preserves DataExchange's ability to forward data without truncation, it significantly complicates the tasks of providing data access and temporarily storing data. In particular, an application cannot simply store binary records for later transmission, because DataExchange's output format might have changed in the meantime. Also, an application desiring data access must be capable of handling changes in the sizes of fields in which it is interested.

To address the former problem, DataExchange directly provides facilities for temporarily storing data and automatically handles converting stored data to the new format when record formats change. The interface to this functionality is provided by DEqueue_data() which requests that DataExchange store a record temporarily, and DErelease_data() which causes a stored record to be released for transmission. Their profiles are below:

```c
extern DESavedRecPtr DEqueue_data(DEPort dep, void *data, int format_id,
        int data_length);    
extern void DErelease_data(DESavedRecPtr saved_data_ptr);
```

For accessing data, DataExchange provides DEget_field_info() which provides information about the size and data type of fields in record formats. DEmake_field_ptr() returns a handle that the application can use to operate on particular fields using DEget_data() and DEput_data(). The prototypes for these routines are shown below:

```c
typedef enum _DEDataResult {    
    DEData_OK, DEData_Too_Small, DEData_Truncate    
} DEDataResult;

extern void DEget_field_info(DExchange de, char *format_name, char *field_name,
        int *size, char **field_type_p);

extern DERecFieldPtr DEmake_field_ptr(DEExchange de, char *format_name, char *field_name,
        int get_size, char *get_type);

extern get_data_result DEget_data(DExchange de, void *data, void *record, DERecFieldPtr field);

extern get_data_result DEput_data(DExchange de, void *data, void *record, DERecFieldPtr field);
```
Because the size of data types can change dynamically, the data access routines return values of 
DEData_OK, DEData_Too_Small, or DEData_Truncate depending upon whether the size the 
application is using is correct for the data, too small but not so small as to truncate the particular 
data being operated on, or so small that current data has been truncated. For access to data which 
is currently stored via DEqueue_data(),DataExchange provides DESavedRec_get_data_ptr() and 
DESavedRec_format_id(). Each of these takes a DESavedRecPtr as an argument and returns the 
pointer to the stored data record and the format_id of the stored record, respectively.

7 Miscellaneous Features

The previous sections have covered most aspects ofDataExchange functionality. This section 
washes up some loose ends and covers smaller topics that do not fit well elsewhere.

7.1 Blocking I/O

WhileDataExchange provides some semblance of an event-driven API to programs which interac 
t over the network, it cannot mask all possible adverse consequences of that interaction. This 
 is particularly true in the area of blocking I/O operations.DataExchange is currently built on 
top of TCP/IP socket connections and it shares the limitations of such systems. In particular, if 
 enough data is written to a socket without being read by the receiver, additional write operations 
will either fail or block, perhaps after transmitting a partial record. Because the lower level PBIO 
library cannot tolerate partial records,DataExchange requires all write operations to be blocking. 
This requirement has several important consequences. For example, it means that aDataExchange 
which does not read its input connections will eventually cause those who send it data to block. 
Thus a single malfunctioning client can disrupt an entire system of communicatingDataExchangees. 
It also means that a simple pair of communicatingDataExchangees may expose themselves to dead-
lock by, for example, each Exchange writing to the other and then each reading what the other has 
written. This type of exchange will probably work as long as the exchanged messages are smaller 
than the buffering capacity of the operating systems on which the programs run. But a change in 
message or buffer size may cause the programs to deadlock.

Approaches to resolving such problems tend to be application-specific andDataExchange does 
not attempt to address them in general.\footnote{Though a UDP version ofDataExchange which is being developed may ease the situation.} However,DataExchange does provide a mechanism 
through which users can be warned that their applications are stalled. In particular, the subroutine 
DEinit_block_check() turns on a self-check mechanism that detects when aDataExchange has 
been blocked on the same operation for more than 1 second. If this condition is detected, a warning 
is printed.

7.2 Multiplexing

It is perhaps most natural to think of a program as operating a singleDataExchange per process 
or thread (and a single thread perDataExchange), but this is by no means a requirement. Multiple 
DataExchangees may be opened and used by a single thread of control. ThoseDataExchangees operate 
independently without coordination of formats, blocking state, etc. Also, multiple threads may 
operate on a singleDataExchange, thoughDataExchange does not protect itself against concurrent 
access and prudent applications must provide this protection outside of theDataExchange library.
7.3 Modifying and Saving Data

As theDataExchange library offers the application program access to arriving data records in several ways, it is sometimes useful to know more specifically how the data is stored and managed and what the effects of changing the data would be. Because DataExchange is built upon PBIO, it shares some of the characteristics of PBIO. When data arrives at a DataExchange the entire input record is placed in a contiguous block of buffer memory. If the record contains internal strings or dynamic elements, those items will be represented as pointers into different segments of that contiguous buffer. (Figure 12 is a diagram, borrowed from the PBIO documentation[1], that depicts the layout of simple record in the buffer.)

Application subroutines operating as filters or functions as described in Section 4 are free to modify the incoming data at any time. Other filters and functions which might be called subsequently will see the modified data. Because filters are called before any implicit data resending, any changes they make to the data will affect the data that is resent to the connections.

After all processing (filters, forwarding and functions) of the input data, the buffer is reused to hold the next input record. Thus programs which wish so store input records beyond the termination that records processing must take steps to ensure that the data isn’t overwritten by the next record. One approach to this involving DQueue_data() was described in Section 6 and should be used in complicated situations where data formats may change over time. However, programs in simpler situations can take advantage of a simpler interface. In particular, the application can call the routine DTake_buffer() to simply acquire the buffer containing the record from DataExchange. The prototype for DTake_buffer() is given below.

```c
extern void DTake_buffer(DataExchange de, void *data);
```

The data parameter is the address of the record being processed and is present only for a consistency check. After DTake_buffer() has been called during processing of a record, the DataExchange library allocates itself a new buffer for holding input data and stops referencing the old buffer. The application is then responsible for eventually passing the buffer to free() when it is no longer of use. Because DataExchange gives up all pointers to the old buffer, DTake_buffer() also implicitly cancels all subsequent processing of the affected record. Thus a filter subroutine that calls DTake_buffer() implicitly cancels resending of that record to connected clients.

**Program Structure:**

```c
struct {   float x;
    int y;
    char *str;
    float z;
}
```

**Incoming File Record:** `{ 3.14159, 5, "hello", 2.717}`

**Incoming buffer layout:**

```
| 3.14159 | 5 | 2.717 | hello | 0 |
```

**Program Structure Length:** 16 bytes  **Required buffer length:** 22 bytes

Figure 12: Buffer layout for an incoming record containing a string.
Acknowledgements

Very early work on DataExchange was based on the Central Monitor in the Falcon system [2] developed by Weiming Gu. Other contributors to DataExchange implementation were Vernard Martin and Jeff Vetter.

References


A Detailed Sample Programs

A.1 Sample DataExchange Server

```c
#include <stdio.h>

#include "DE.h"

void
main(argc, argv)
int argc; char **argv;
{
    DEExchange de = DEExchange_create();
    if (DEExchange_listen(de, 12345) == -1) {
        fprintf(stderr, "Unable to listen\n");
        exit(-1);
    }
    printf("DataExchange server listening at Inet host/port \%s \%d\n", DEExchange_host_name(de), DEExchange_inet_port(de));
    while (1) {
        DEExchange_poll_and_handle(de, TRUE);
    }
}
```

A.2 Simple Write Client

```c
#include <stdio.h>

#include "DE.h"

typedef struct _rec {
    int integer_field;
    char *string_field;
} rec, *rec_ptr;

static IOField field_list[] = {
    {"integer_field", "integer", sizeof(int), IOOffset(rec_ptr, integer_field)},
    {"string_field", "string", sizeof(char*), IOOffset(rec_ptr, string_field)},
    {NULL, NULL, 0, 0}};
```
void main(argc, argv)
int argc;
char **argv;
{
    DExchange de = DExchange_create();
    DEPort dep;
    int i, rec_format_id;

    char *host = "aquarius.cc.gatech.edu";
    int port = 12345;
    if (argc >= 2) {
        host = argv[1];
    }
    if (argc >= 3) {
        sscanf(argv[2], "%d", &port);
    }
    DExchange_register_format(de, "my format", field_list);
    rec_format_id = DEget_format_id(de, "my format");
    dep = DExchange_initiate_conn(de, host, port, TRUE);
    for (i = 0; i < 10; i++) {
        rec buf;
        buf.integer_field = i; buf.string_field = "Hello World!";
        DEPort_write_data(dep, rec_format_id, (char *)&buf);
        sleep(1); /* take some time */
    }
}

A.3 Simple Read Client
#include <stdio.h>
#include "DE.h"

typedef struct _rec {
    int integer_field;
    char *string_field;
} rec, *rec_ptr;

static IOField field_list[] = {
    {"integer_field", "integer", sizeof(int), IOOffset(rec_ptr, integer_field)},
    {"string_field", "string", sizeof(char*), IOOffset(rec_ptr, string_field)},
    {NULL, NULL, 0, 0}};

void main(argc, argv)
int argc;
char **argv;
{
    DExchange de = DExchange_create();
    DEPort dep;

    char *host = "aquarius.cc.gatech.edu";
    int port = 12345;
    if (argc >= 2) {
        host = argv[1];
    }
if (argc >= 3) {
    sscanf(argv[2], "%d", &port);
}
DExchange_register_format(de, "my format", field_list);
dep = DExchange_initiate_conn(de, host, port, FALSE);
while (1) {
    rec_ptr recv;
    int rcvd_format;
    recv = (rec_ptr) DEport_read_data(dep, &rcvd_format);
    printf(" Received %d, %s\n", recv->integer_field, recv->string_field);
}

A.4 Efficient Read Client

#include <stdio.h>
#include "DE.h"

typedef struct __rec {
    int integer_field;
    char *string_field;
} rec, *rec_ptr;

static IDField field_list[] = {
    {"integer_field", "integer", sizeof(int), IOOffset(rec_ptr, integer_field)},
    {"string_field", "string", sizeof(char*), IOOffset(rec_ptr, string_field)},
    {NULL, NULL, 0, 0}};

int handler(de, dep, format_id, data, data_length)
DExchange de;
DEPort dep;
int format_id;
void *data;
int data_length;
{
    rec_ptr recv = (rec_ptr) data;
    printf(" Received %d, %s\n", recv->integer_field, recv->string_field);
    return 1; /* return value is ignored for functions */
}

void main()
{
    DExchange de = DExchange_create();
    DEPort dep;

    DExchange_register_format(de, "my format", field_list);
    DExchange_register_function(de, "my format", handler);
    dep = DExchange_initiate_conn(de, "host.domain.org", 12345, TRUE);
    DEPort_set_format_block(dep, "my format", FALSE);
    while (1) {
        DExchange_poll_and_handle(de, TRUE);
    }
}