Direct Deposit:
A Basic User-Level Protocol for Carpet Clusters [1]

Mark R. Swanson
Leigh B. Stoller
E-mail: {swanson,stoller}@cs.utah.edu

UUCS-95-003

Department of Computer Science
University of Utah
Salt Lake City, UT 84112, USA

June 25, 1996

Abstract

This note describes the **Direct Deposit Protocol** (DDP), a simple protocol for multicomputing on a carpet cluster. This protocol is an example of a user-level protocol to be layered on top of the low-level, sender-based protocols for the Protocol Processing Engine. The protocol will be described in terms of its system call interface and an operational decription.

[1]This work was supported by a grant from Hewlett-Packard, and by the Space and Naval Warfare Systems Command (SPAWAR) and Advanced Research Projects Agency (ARPA), Communication and Memory Architectures for Scalable Parallel Computing, ARPA order #B990 under SPAWAR contract #N00039-95-C-0018

Contents

1 Introduction

This note describes the Direct Deposit Protocol (DDP), a simple protocol for multicomputing on a carpet cluster. This protocol is an example of a user-level protocol to be layered on top of low-level, sender-based protocols (SBP) [1]. Its salient feature is the ability to transmit message data directly into a receiving process' address space with neither OS involvement nor memory-to-memory copies. The particular SBP which DDP is being targetted at is implemented by a Protocol Processing Engine DDP will be described in terms of its system call interface and an operational decription, including a number of examples.

2 Concepts and Definitions

DDP is *service* and *connection* oriented. A process wishing to provide a service or, more generally, to accept connections, registers a service with the kernel. The process registering the service provides a name, specified by a string, by which the service is known. Other processes may establish a connection to the service named by that string. Communications can only occur via such connections. Connections are either bidirectional or unidirectional. Each bidirectional connection is comprised of two unidirectional *channels*. Flow control is not provided, so bidirectional channels are necessary when the participating processes must engage in high-level flow control.

A server may wish to accept more than one connection to a particular service. At registration, it informs the kernel of the maximum number of connections to be allowed for the service. A server may also register more than one service. Each registered service is associated by the kernel with a unique service identifier, referred to hereafter as a `servid`, which the kernel returns to the server as a result of service registration.

Servers are normally *not* notified *directly* of connections to their registered services. Each message received by a process is accompanied by the identifier of the connection it arrived on. This may be the first indication to a server of the existence of a given connection. Alternatively, the server may arrange for notification of new connections as an option.

To allow servers to listen conveniently for incoming messages in the (possible) presence of multiple services and multiple connections per service, a notification list (identified by a `noteid`) is associated with each incoming connection. The same notification list may be associated with several connections and/or services. Note that a given notification list belongs to a particular process. If the service registration call does not specify a notification list, the kernel allocates one and returns it's identifier as one result of the registration. Notification lists that become unreferenced are quietly deallocated.

At registration of a service, the process must also specify the buffer space for incoming messages. The amount of space must be sufficient to support the specified maximum number of connections. The buffer must be allocated using a special DDP system call (see `ddalloc()`) so that the kernel can arrange for the buffer to be composed of contiguous "wired" pages, thus allowing the message passing hardware to copy messages directly into the processs address space.

Connection to a service requires the name of the service and the logical node number on which it can be found. Once again, a buffer for incoming messages must be provided; this buffer space must also be previously allocated, as it too will be "wired".

Connections are typed; currently there are three types of connections:

`RPC` Synchronous RPC. The client side uses the connection in a synchronous manner; send blocks waiting for the reply message. The ddcall() routine is used combine send and receive into a single operation. Servers are required to generate a reply message for each incoming message so that clients can eventually continue execution.

`SPLIT` Split phase RPC. The client side can decouple the send and receive operations using the ddsend() and ddrecv() routines, thus allowing communication and computation to be overlapped, where appropriate. Servers are not required to generate reply messages in split phase connections.

`CHANNEL` Unidirectional messaging. The resulting connection is a unidirectional channel on which clients can send messages, but not receive replies.

Connection establishment returns a connection identifier, hereafter referred to as a `connid`, which is used to specify the connection to the kernel when a message is sent on it. A `noteid` is associated with the return portion of the connection. For synchronous RPC connections, the kernel always generates a new notification list and does not return its `noteid`. For split-phase connections, the kernel will generate a `noteid` if the process does not provide one (from a previous connect call).

Generic send and receive system calls are provided for message transmission and reception. Servers will generally use ddrecv() to receive incoming messages, and ddsend() to generate replies when appropriate. Clients of split-phase connections, can send a message and return immediately; modulo buffer management considerations, the client must only determine that the message has been completely received before sending another message on the connection. Explicit buffer management and/or flow control is the responsibility of the user for such connections.

Many applications need to send information (metadata) about a message along with the actual data. Since the goal of DDP is to deposit incoming messages directly in the locations where they will be used, it is not always convenient or even possible to bundle the metadata in the message body. Hence, a variant of the send operation is provided that provides for transmitting modest amounts of metadata: a separate source location is used for specifying the metadata to the send operation.

The name spaces for all of the identifiers described in this section (`serviceid`, `noteid`, and `connid`) are process specific; they only have meaning within the context of the owning process.

3 System Call Interface

3.1 `ddserve()`

```
int
ddserve(char        *name,
        int         flags,
        int         maxconn,
        address_t   recvbase,
        size_t      recvsize,
        noteid_t    *noteid,
        servid_t    *servid);
```

`ddserve` registers a service with the given `name`; the kernel generates and returns, in the location pointed to by `*servid`, a service identifier which is used in future kernel-process interactions to specify the service. The buffer area for incoming requests is specified in `recvbase` and it's size by `recvsize`. The kernel divides the area into equal size blocks among the possible maximum number of connections. The buffer area must be allocated with `ddalloc()`. The current implementation requires that the buffer area be aligned to a 128 byte boundry, and that the minumim block size be a multiple of 128 bytes.

At the moment there are no provisions for creating unique service names, so the user is responsible for ensuring that `name` is not already in use on the node. The default value for `flags` is `SERVE_NOOPTS`.

If `flags` contains `SERVE_NOTELIST`, the kernel will associate the notification object specified in `noteid` with the new service. This allows multiple servers to share a single notification object (see `ddrecv()`). Otherwise, the kernel will allocate and return, in `noteid`, the identifier of a new notification object.

If `flags` contains `SERVE_DOACCEPT`, each new connection to the service will result in the allocation of a new notification object; such connections will be reported to the server (on the original notification object) as connection notifications. The program must then use the `ddconninfo()` call to determine the noteid of the new notification object for the connection. This is similar in style to how the Unix system call `accept()` operates in that a single, master noteid (file descriptor for `accept()`) is used to accept new connections, while each new connection results in a new noteid (again, file descriptor).

If `flags` contains `SERVE_BIGBUF`, the incoming message buffer is not partitioned among new connections. Instead, each new connection is given access to the entire buffer. Programs must exercise care when sending messages so as not to overwrite another connection's message data.

`ddserve` returns zero on success and -1 otherwise.

ERRORS:

`EBADID`	The noteid provided was not valid.
`ENOCMP`	The kernel thread servicing DD requests is not running.
`ENORESOURCE`	The kernel is unable to allocate some required data structure.

3.2 ddconn()

```
int
ddconn(char        *name,
       int         node,
       int         type,
       int         flags,
       address_t   *recvbase,
       size_t      recvsize,
       connid_t    *connid,
       noteid_t    *noteid);
```

`ddconn` establishes the client side of a client-server connection by attempting to connect to a service created with `ddserve` on logical node number `node`. `name` is the string associated with the service. The kernel generates and returns, in the location pointed to by `*connid`, a connection identifier which is used in future kernel-process interactions (such as sending a message) to specify the connection. At present, there is no name service available, so it is up to the user to keep track of `name,node` pairs, and to ensure that the pairings are unique. The buffer area that the process will use to receive replies is specified as `recvbase`, and its size is given in `recvsize`. The buffer area must be allocated with `ddalloc()`.

The `type` argument must be specified as either `TYPE_RPC`, `TYPE_SPLIT`, or `TYPE_CHANNEL`. If `type` is `TYPE_RPC`, an RPC style connection is established (see `ddcall()`) in which requests block until a reply is received. A notification object is not returned since there is no need to do a separate receive operation on the connection. If `type` is `TYPE_SPLIT`, a split-phase connection is established (see `ddsend()` and `ddrecv()`) in which requests return immediately, and replies must be explicitly requested from the system. An example of a split-phase client and server follows in Section 5.

If `type` is `TYPE_CHANNEL`, a unidirectional connection is established. No receive channel is provided, and thus a notification object is not returned. As with split-phase connections, it is up to the user to ensure that target buffers are not overwritten before they are consumed by the receiver. Servers must also take care to check the type of new connections (using `ddconninfo()`) so that reply messages are not generated, which would cause a program error.

If `flags` contains `CONN_NOTELIST`, and the `type` is `TYPE_SPLIT`, the kernel will associate the notification object specified in `*noteid` with the new connection. This allows multiple split-phase connections to share a single notification object (see `ddrecv()`). Otherwise, the kernel will allocate and return, in `*noteid`, the identifier of a new notification object.

`ddconn` returns zero on success and -1 otherwise.

ERRORS:

`EBADID`	The noteid provided was not valid.
`EBADSERVER`	The service name was not registered on the node specified.
`ENOCMP`	The kernel thread servicing DD requests is not running.
`ENORESOURCE`	The kernel is unable to allocate some required data structure.

3.3 ddnewnotelist()

```
int
ddnewnotelist( noteid_t  *noteid,
               int       count );
```

`ddnewnotelist` creates and returns a notelist object in `*noteid`. The number of notification objects in the new notelist is given by `count`. This new notelist can used in the establishment of new connections, and may be shared among connections if appropriate. This call is most useful when the number of notification objects must be more closely controlled by the program; default notelists created by the kernel contain a small number of notification objects, which might not be appropriate for programs that expect to receive large numbers of messages. Programs that receive messages more quickly than they can be processed run the risk of losing notifications.

`ddnewnotelist` returns zero on success and -1 otherwise.

ERRORS:

`ENORESOURCE`	The kernel is unable to allocate some required data structure.

3.4 ddclose()

```
int
ddclose( connid_t  connid );
```

ddclose terminates the connection specified by connid, freeing any kernel resources associated with it. ddclose can be applied to either the client or the server side of a connection. The "other" end of of the connection is closed as well.

ddclose returns zero on success and -1 otherwise.

ERRORS:

EBADID The connid provided was not valid.

3.5 ddunserve()

```
int
ddunserve( servid_t  servid );
```

ddunserve terminates a server established with ddserve, freeing any kernel resources–except for the buffers– associated with it. All existing connections to the server are terminated. The client side of established connections are marked as closed, and subsequent use of those connections results in an error being returned to the client.

ddunserve returns zero on success and -1 otherwise.

ERRORS:

EBADID The servid provided was not valid.

3.6 ddsend()

```
int
ddsend( connid_t   connid,
        address_t  addr,
        size_t     len,
        offset_t   roffset,
        option_t   options );
```

ddsend sends a message on the connection referred to by connid. The message begins at addr and is len bytes in length. The address must be aligned on a word boundry. The length must be a multiple of a word (4 bytes), up to MAXDDMSGSIZE. Zero length messages are permitted. The message is deposited at roffset bytes into the destination buffer. The default value for options is SEND_NOOPTS. ddsend returns immediately, possibly before the message has been completely sent. The process must be careful not to alter the message buffer until the message transmission is known to have completed, as data corruption could otherwise occur. The dddone system call should be used to determine when the message has been completely sent, and the buffer can safely be reused.

The connection cannot be of type `TYPE_RPC` (see `ddconn`) since that implies synchronous RPC operation. If the connection type is of `TYPE_SPLIT`, then multiple messages can be sent on a connection without waiting for program level replies (see `ddrecv`). This allows greater overlap of communication with computation, and avoids unnecessary synchronization. The program must be careful to avoid overwritting the target buffer before it has been consumed by the receiver. If the connection type is of `TYPE_CHANNEL`, then replies should not be expected, or generated by the server side of the connection.

If `options` includes `SEND_BLOCK`, `ddsend` does not return until the message has been completely sent. At this point, the buffer can safely be reused to compose a new message.

If `options` includes `SEND_UNWIRED`, `ddsend` forces the data to be sent by the CPU. Instead of sending the data using a DMA transfer, the message data is written directly by the CPU. This option is helpful when the program cannot easily arrange for message data to be within a wired buffer.

If `options` includes `SEND_DMA`, `ddsend` forces the data to be sent using a DMA transfer. The data must reside in a wired down buffer previously allocated with `ddalloc()`.

If `options` includes `SEND_NONOTIFY`, the send operation will *not* cause a notification to be generated at the receiver. This is useful when the programmer deems it unnecessary for the receiver to know when each message has arrived. Care should be taken however to ensure that unconsumed messages are not overwritten.

`ddsend` returns zero on success and -1 otherwise.

ERRORS:

`EBADID`	The connid provided was not valid.
`ECONNCLOSE`	The connection is no longer valid since the other side has been closed.
`ESLOTBUSY`	A request is still outstanding; the reply has not been consumed.
`EBADTYPE`	The connid refers to a connection that has a type of `TYPE_RPC`.
`EBADRANGE`	The message is not contained within the buffer that was specified with `ddserve` or `ddconn`.

3.7 `ddsend_with_metadata()`

```
int
ddsend_with_metadata( connid_t   connid,
                      address_t  addr,
                      size_t     size,
                      offset_t   roffset,
                      option_t   options,
                      address_t  metap,
                      size_t     metacount );
```

`ddsend_with_metadata` is identical to `ddsend`, with the exception that data pointed to by `metap` is transmitted along with the normal message payload. This data appears in the receiver's notification. `metacount` specifies the number of (4 byte) words of metadata to be sent; it is currently limited to 4.

3.8 dddone()

```
int
dddone( connid_t  connid,
        int       flags );
```

dddone is used to determine when the last message sent on connid has been completely sent, and the corresponding data buffer can safely be reused. dddone is normally used in conjunction with ddsend_async to ensure that the data buffer is ready for reuse. By default, dddone returns a status value immediately. If options includes SDONE_SLEEP, and the message has not been completely sent, dddone will block in the kernel until the message has been completely sent.

dddone returns -1 on error. 1 is returned when the message has been completely sent, and 0 otherwise.

ERRORS:

EBADID The connid provided was not valid.
EINVAL Improper options were specified.

3.9 ddflush()

```
int
ddflush( address_t  addr,
         size_t     len );
```

ddflush is used to maintain consistency between cache and main memory. Systems that possess coherent IO subsystems may need to take no action at all to maintain this coherence, while others will need to explicitly flush the address range from the cache.

ddflush should be used to flush a receive buffer after the program has consumed the data within it and before the program informs the sender that the buffer can be reused, so that subsequent messages sent to the buffer will not be shadowed by any residual cache contents.

ddflush always returns 0.

3.10 ddrecv()

```
int
ddrecv( noteid_t      noteid,
        ddrecv_desc_t recvblk;
        option_t      options,
        timeout_t     timeout );

typedef struct {
       address_t    msgaddr;
       int          msgsize;
       connid_t     connid;
       servid_t     servid;
```

```
        unsigned        metad[4];
        int             metacount;
} ddrecv_desc_t;
```

`ddrecv` consumes the next available incoming message for any of the connections associated with `noteid`. The second argument, a a pointer to a `ddrecv_desc_t` structure, is used to collect various return values. `*servid` is set to the unique service identifier associated with the connection (this value is generally only meaningful for the server side of connections). `*msg` is set to the byte address of the start of the message, and `*length` is set to the number of bytes in the message. The default value for `options` is `RECV_SPIN`, which causes `ddrecv` to spin in the kernel, waiting for a notification to be posted. The process may be context switched by the kernel as appropriate, but is otherwise capable of receiving a message as soon as it arrives, without any kernel intervention. This is in contrast to specifying `RECV_SLEEP`, which causes the process to sleep in the kernel until a notification is posted. The `RECV_SPIN` option, while resulting in faster notifications, is only appropriate for single-application or lightly loaded systems, due to its potential for consuming CPU resources.

If `options` includes `RECV_NOBLOCK`, `ddrecv` always returns immediately. If a message is not pending, `ddrecv` returns -1, and the global variable `errno` is set to `EWOULDBLOCK`.

If `options` includes `RECV_TIMEOUT`, `ddrecv`, will block for the amount of time specified by `timeout`. The kernel expects `timeout` to be in the format `timeout * SBP_HZ`. Note: The value of `SBP_HZ` is equal to the system clock rate; 100 ticks per second in HPUX and BSD4.3.

If the incoming message contains metadata, it is returned in the unsigned `metad` array, and the count of (4 byte) words of metadata is placed in `metacount`. The current implementation limits the amount of metadata to 4 words. If the incoming message does not contain metadata, `metacount` is set to zero, and the `metad` array is left untouched.

`ddrecv` returns -1 on error, and 0 otherwise.

ERRORS:

`EBADID`	The noteid provided was invalid.
`EINVAL`	Improper options were specified.
`EWOULDBLOCK`	`RECV_NOBLOCK` was specified, and no message is pending.

3.11 `ddcall()`

```
int
ddcall( connid_t   connid,
        address_t  saddr,
        size_t     ssize,
        address_t  *repaddr,
        size_t     *repsize,
        option_t   options,
        timeout_t  timeout );
```

`ddcall` combines the send and receive operations into a single, synchronous operation on the connection referred to by `connid`. The request message begins at `saddr`, and is `ssize` bytes in length. The

```
typedef struct conninfo {
        short           type;
        short           node;
        noteid_t        noteid;
        address_t       recvbase;
        size_t          recvsize;
        int             dstsize;
        int             notecount;
} conninfo_t;
```

Figure 1: conninfo_t data structure.

address must be aligned on a word boundry. The length must be a multiple of a word (4 bytes), up to `MAXDDMSGSIZE`. Zero length messages are permitted. Unlike `ddsend`, the request message is always placed at the beginning of the receive buffer on the target processor. `ddcall` then blocks until a reply message is received, at which time, `*repaddr` is set to the byte address of the start of the reply message, and `*repsize` is set to the number of bytes in the reply message. `ddcall` thus implements a form of remote procedure call.

If `options` includes `RECV_TIMEOUT`, `ddcall`, will block waiting for a reply for the amount of time specified by `timeout`. The kernel expects `timeout` to be in the format `timeout * SBP_HZ`. At the end of the timeout, `ddcall` will return -1, and the global value `errno` will be set to `ETIMEDOUT`. It is up to the application program to recover from this error.

`ddcall` returns -1 on error, and 0 otherwise.

ERRORS:

`EBADID`	The connid provided was invalid.
`EINVAL`	Improper options were specified.
`ECONNCLOSE`	The connection is no longer valid since the other side has been closed.
`EBADTYPE`	The connid refers to a connection that does not have a type of `TYPE_RPC`.
`EBADRANGE`	The message is not contained within the buffer that was specified with `ddconn`.
`ETIMEDOUT`	Options included `RECV_TIMEOUT`, and the timeout expired.

3.12 ddconninfo()

```
int
ddconninfo( connid_t    connid,
            conninfo_t  *conninfo );
```

`ddconninfo` returns information about `connid`. `*conninfo` should point to the address of a `conninfo_t` structure (see Figure 1), and is defined below. The `ddconninfo` call is most useful when used in conjunction with `ddserv` to determine the addresses at which buffers have been placed for each new connection (see example code in Section 5).

`type` The type of the established connection, and is one of TYPE_RPC, TYPE_SPLIT, or

TYPE_CHANNEL.

node The logical node number of the processor the connection is established with.

noteid The notification object for receiving messages.

recvbase The starting address of the incoming message buffer.

recvsize The size of the incoming message buffer.

dstsize The size of the remote receive buffer.

dstnotecount The number of note entries in the remote notelist object.

`ddconninfo` returns -1 on error, and 0 otherwise.

ERRORS:

`EBADID` The connid provided is invalid.

`EINVAL` `*conninfo` is an invalid pointer.

3.13 ddpoll()

```
int
ddpoll( int        count,
        noteid_t   *notelist,
        timeval_t  *timeout );
```

`ddpoll` examines each of the noteids contained in the vector pointed to by `*notelist` to see if any of them have unconsumed messages pending. `count` is the number noteids contained in `notelist`. On return, `ddpoll` replaces the contents of `*notelist` with the subset of noteids that have unconsumed messages pending. Note that each noteid is modified in place; noteids with no messages pending are set to zero, while noteids with messages pending are left as is. This is similar in operation to the UNIX system call select. `ddpoll` returns the number of ready noteids contained in the new set.

If `timeout` is a non-NULL pointer, it specifies a maximum interval to wait for the `ddpoll` operation to complete. If `timeout` is a NULL pointer, `ddpoll` blocks indefinitely. To affect a poll, the `timeout` argument should be non-NULL, pointing to a zero-valued timeval structure.

`ddpoll` returns -1 on error. Otherwise, the number of noteids with unconsumed messages is returned.

ERRORS:

`EBADID` One of the noteids provided is invalid.

`EINVAL` `*timeout` or `*notelist` is an invalid pointer.

3.14 ddalloc()

```
address_t
ddalloc( size_t  size );
```

`ddalloc` allocates a new memory region for use with either `ddconn` or `ddserve`. The region is `size` bytes in length. Memory allocated with `ddalloc` is special in that the kernel will permanently wire the memory down so that DMA operation to and from the region will work properly. The memory region returned by `ddalloc` is always aligned on a 128 byte boundry.

ddalloc returns the starting address of the new memory region. If the memory cannot be allocated, NULL is returned.

3.15 ddpeek()

```
int
ddpeek( noteid_t  noteid,
        size_t    *length,
        connid_t  *connid );
```

ddpeek is used to determine if there are any unconsumed incoming messages on noteid. For the first unconsumed message, *connid is set to the connection identifier of the message, and *length is set to the number of bytes contained in the message. If there are no unconsumed messages, *connid is set to -1, and length is set to 0. *Should this be encoded in the return value instead?*

ddpeek returns -1 on error, and 0 otherwise.

ERRORS:

EBADID The noteid provided was invalid.

3.16 ddready()

```
int
ddready( noteid_t  noteid );
```

ddready is used to determine if there are any unconsumed incoming messages on noteid.

ddready returns -1 on error, 1 if the noteid has an unconsumed message pending, and 0 if there are no unconsumed messages pending.

ERRORS:

EBADID The noteid provided was invalid.

3.17 ddmynode() and ddmaxnodes()

```
int
ddmynode( void );
```

ddmynode returns the logical node number of the current procesor. The node number is a integer value between 0 (inclusive) and the value returned by ddmaxnodes (non-inclusive).

```
int
ddmaxnodes( void );
```

ddmaxnodes returns the total number of processors in the system, as determined by the kernel at boot time.

3.18 `ddpacketsize()`

```
int
ddpacketsize( void );
```

`ddpacketsize` returns the maximum packet size supported by the interface and interconnect fabric.

3.19 `dderror()`

```
void
dderror( char  *string, ... );
```

`dderror` finds the error message corresponding to the current value of the global variable `errno`, and writes a desciptive message to `stderr`. The argument `string` should be a format string, followed by optional arguments. The resulting message is prepended to the system message. `dderror` is similar in operation to the C library function `perror`, but works with SBP error values as well as system error values.

4 Sample Program 1

The first sample program is a simple "echo" client and server in which the server responds to messages from its clients by returning the data it receives.

4.1 Echo Server

The main lines of a very simple server program is presented in Figure 2. Initially, a service with the name "echo" is registered. It is capable of concurrently supporting two connections and has incoming buffer space of 4096 bytes. Each connection will, therefore, be allocated receive buffer space of 2048 bytes. Next, the server simply loops, receiving a message, copying it to a reply buffer, and sending it back to the caller.

4.2 Echo Client

Correspondingly, a very simple client of the echo server is presented in Figure 3. First, it connects to the echo server (which it *knows* is on node 1). Note that it passes `NULL` in place of the `noteid` parameter, since the notification list for a `TYPE_RPC` connection is always anonymous. The client contructs a 128 byte "request" in the send buffer, which is simply returned by the echo server. `ddcall` waits for the reply, enforcing a synchronous form of RPC. The reply is placed in the receive buffer, and the actual address is returned by ddcall. The client then exits.

```
#include "fedex.h"

main()
{
    noteid_t      noteid;
    address_t     foo, bar;
    servid_t      thisserv;
    ddrecv_desc_t frd;
    conninfo_t    finfo;

    bar = ddalloc(0x1000);
    foo = ddalloc(0x1000);

    /*
     * Establish a server.
     */
    if (ddserve("echo", SERVE_NOOPTS, 2, foo, 0x1000,
                 &noteid, &thisserv) < 0) {
        dderror("ddserve");
        exit(1);
    }

    for (;;) {
        if (ddrecv(noteid, &frd, RECV_SLEEP, NULL) < 0) {
            dderror("ddrecv");
            exit(1);
        }
        if (ddconninfo(frd.connid, &finfo) < 0) {
            dderror("ddconninfo");
            exit(1);
        }
        bcopy(frd.msgaddr, bar, frd.msgsize);
        if (ddsend(frd.connid, bar, frd.msgsize,
                   0, SEND_NOOPTS) < 0) {
            dderror("ddsend");
            exit(1)
        }
    }
}
```

Figure 2: Echo server program.

```
#include "fedex.h"

main()
{
    int         i;
    address_t   foo, bar, msg;
    connid_t    connid;
    size_t      size;

    /*
     * Establish a connection to a server.
     */
    foo = (addr_t)ddalloc(0x1000);
    bar = (addr_t)ddalloc(0x1000);

    if (ddconn("echo", 1, TYPE_RPC, CONN_NOOPTS,
                foo, 0x1000, &connid, NULL) < 0) {
        dderror("ddconnect");
        exit(1);
    }

    /*
     * Fill in the message contents;
     */
    for (i = 0; i < 32; i++)
        bar[i] = i;

    /*
     * Do an RPC.
     */
    if (ddcall(connid, bar, 128, &msg, &size) < 0) {
        dderror("ddcall");
        exit(1);
    }
}
```

Figure 3: A simple client program.

5 Sample Program 2

Sample program two is an exmample of a multiple-service server and client, and demontrates the use of split-phase connections.

5.1 Multiple-Service Server

A server program that offers two services is shown in Figure 4. Note that in registering the second service, the `noteid` value provided is the one returned from the registration of the first service. This allows the server to use a single `ddrecv` call to listen for requests to either service. In order to differentiate between requests, which can be for either service, it remembers the `serviceid`'s returned at registration. `ddrecv` passes back the `serviceid` with which the incoming message is associated and the server uses this to pick the right action.

5.2 Multiple-Service Client

Here, a split-phase client of the echo server is presented in Figure 5. In connecting to the echo service, it provides a pointer to a null notification, which the kernel will replace. In connecting to the reverse service, it passes this `noteid`, so that it can listen for respones from either service with a single `ddrecv` call. It sends a message to each of the services, performs some other task and eventually uses `ddrecv` to read a response. The `serviceid` parameter is not meaningful in this context, so the client supplies a null pointer. It uses the value returned in `conn` to determine which service the response is from.

References

[1] Wilkes, J. Hamlyn - an interface for sender-based communication. Tech. Rep. HPL-OSR-92-13, Hewlett-Packard Research Laboratory, Nov. 1992.

```
main()
{
    noteid_t        noteid = 0;
    address_t       foo, bar;
    servid_t        echoid, reverseid;
    ddrecv_desc_t   frd;
    conninfo_t      finfo;

    foo = ddalloc(0x1000);
    bar = ddalloc(0x1000);
    if (ddserve("echo", SERVE_NOOPTS, 2,
                foo, 0x1000, &noteid, &echoid) < 0) {
        dderror("ddserve");
        exit(1);
    }

    foo = ddalloc(0x1000);
    if (ddserve("reverse", SERVE_NOOPTS, 2,
                foo, 0x1000, &noteid, &reverseid) < 0) {
        dderror("ddserve");
        exit(1);
    }

    for (;;) {
        if (ddrecv(noteid, &frd, RECV_SLEEP, NULL) < 0) {
            dderror("ddrecv");
            exit(1);
        }
        if (ddconninfo(frd.connid, &finfo) < 0) {
            dderror("ddconninfo"); exit(1);
        }
        if (frd.servid == echoid)
            bcopy(frd.msgaddr, bar, frd.msgsize);
        else
            reverse(frd.msgaddr, bar, frd.msgsize);

        if (ddsend(frd.connid, bar, frd.msgsize,
                  0, SEND_NOOPTS) < 0) {
            dderror("ddsend");
            exit(1);
        }
    }
}
```

Figure 4: Multiple-Service server program.

```
main()
{
    int           i;
    address_t     req1, req2, resp1, resp2;
    connid_t      connid1, connid2;
    ddrecv_desc_t frd;
    noteid_t      noteid = 0;

    resp1 = ddalloc(0x1000);
    req1  = ddalloc(0x1000);
    if (ddconn("echo", 1, TYPE_SPLIT, CONN_NOOPTS,
                resp1, 0x1000, &connid1, &noteid) < 0) {
        dderror("ddconn");
    }

    resp2 = ddalloc(0x1000);
    req2  = ddalloc(0x1000);
    if (ddconn("reverse", 1, TYPE_SPLIT, CONN_NOOPTS,
               resp2, 0x1000, &connid2, &noteid) < 0) {
        dderror("ddconn");
    }

    for (i = 0; i < 32; i++)
        req1[i] = req2[i] = i;

    if (ddsend(connid1, req1, 128, 0, SEND_NOOPTS) < 0)
        dderror("ddsend");
    }
    if (ddsend(connid2, req2, 128, 0, SEND_NOOPTS) < 0)
        dderror("ddsend");}

    do_something_else();
    if (ddrecv(noteid, &frd, RECV_SLEEP, NULL) < 0) {
        dderror("ddsend");
    }
    if (frd.connid == connid1)
        process_echo_response();
    else
        process_reverse_response();
}
```

Figure 5: Multiple-Service client program.