             The Architecture of Direct Data Placement (DDP)
          and Remote Direct Memory Access (RDMA) on Internet Protocols

Status of This Memo

Copyright Notice

Abstract

   This document defines an abstract architecture for Direct Data
   Placement (DDP) and Remote Direct Memory Access (RDMA) protocols to
   run on Internet Protocol-suite transports. This architecture does
   not necessarily reflect the proper way to implement such protocols,
   but is, rather, a descriptive tool for defining and understanding the
   protocols. DDP allows the efficient placement of data into buffers
   designated by Upper Layer Protocols (e.g., RDMA). RDMA provides the
   semantics to enable Remote Direct Memory Access between peers in a
   way consistent with application requirements.

Table of Contents

1. Introduction

   This document defines an abstract architecture for Direct Data
   Placement (DDP) and Remote Direct Memory Access (RDMA) protocols to
   run on Internet Protocol-suite transports. This architecture does
   not necessarily reflect the proper way to implement such protocols,
   but is, rather, a descriptive tool for defining and understanding the
   protocols. This document uses C language notation as a shorthand to
   describe the architectural elements of DDP and RDMA protocols. The
   choice of C notation is not intended to describe concrete protocols
   or programming interfaces.

   The first part of the document describes the architecture of DDP
   protocols, including what assumptions are made about the transports
   on which DDP is built. The second part describes the architecture of
   RDMA protocols layered on top of DDP.

1.1. Terminology

   Before introducing the protocols, certain definitions will be useful
   to guide discussion:

   o   Placement - writing to a data buffer.

   o   Operation - a protocol message, or sequence of messages, which
        provide an architectural semantic, such as reading or writing of
        a data buffer.

o   Delivery - informing any Upper Layer or application that a
     particular message is available for use. Therefore, delivery
     may be viewed as the "control" signal associated with a unit of
     data. Note that the order of delivery is defined more strictly
     than it is for placement.

o   Completion - informing any Upper Layer or application that a
     particular operation has finished. A completion, for instance,
     may require the delivery of several messages, or it may also
     reflect that some local processing has finished.

o   Data Sink - the peer on which any placement occurs.

o   Data Source - the peer from which the placed data originates.

o   Steering Tag - a "handle" used to identify the buffer that is
     the target of placement. A "tagged" message is one that
     references such a handle.

o   RDMA Write - an Operation that places data from a local data
     buffer to a remote data buffer specified by a Steering Tag.

o   RDMA Read - an Operation that places data to a local data buffer
     specified by a Steering Tag from a remote data buffer specified
     by another Steering Tag.

o   Send - an Operation that places data from a local data buffer to
     a remote data buffer of the data sink's choice. Therefore,
     sends are "untagged".

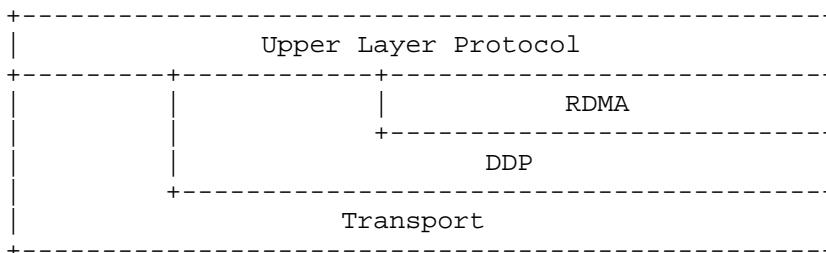1.2. DDP and RDMA Protocols

   The goal of the DDP protocol is to allow the efficient placement of
   data into buffers designated by protocols layered above DDP (e.g.,
   RDMA). This is described in detail in [ROM]. Efficiency may be
   characterized by the minimization of the number of transfers of the
   data over the receiver's system buses.

   The goal of the RDMA protocol is to provide the semantics to enable
   Remote Direct Memory Access between peers in a way consistent with
   application requirements. The RDMA protocol provides facilities
   immediately useful to existing and future networking, storage, and
   other application protocols. [FCVI, IB, MYR, SDP, SRVNET, VI]

   The DDP and RDMA protocols work together to achieve their respective
   goals. DDP provides facilities to safely steer payloads to specific
   buffers at the Data Sink. RDMA provides facilities to Upper Layers
   for identifying these buffers, controlling the transfer of data

between peers' buffers, supporting authorized bidirectional transfer
between buffers, and signalling completion. Upper Layer Protocols
that do not require the features of RDMA may be layered directly on
top of DDP.

The DDP and RDMA protocols are transport independent. The following
figure shows the relationship between RDMA, DDP, Upper Layer
Protocols, and Transport.

```
          +-----------------------------------------------+
          |             Upper Layer Protocol              |
          +---------+-----------+-------------------------+
          |         |           |              RDMA       |
          |         |           +-------------------------+
          |         |           |              DDP        |
          |         +-----------------------------------------+
          |             Transport                         |
          +-----------------------------------------------+
```

2. Architecture

   The Architecture section is presented in two parts: Direct Data
   Placement Protocol architecture and Remote Direct Memory Access
   Protocol architecture.

2.1. Direct Data Placement (DDP) Protocol Architecture

   The central idea of general-purpose DDP is that a data sender will
   supplement the data it sends with placement information that allows
   the receiver's network interface to place the data directly at its
   final destination without any copying. DDP can be used to steer
   received data to its final destination, without requiring layer-
   specific behavior for each different layer. Data sent with such DDP
   information is said to be 'tagged'.

   The central components of the DDP architecture are the 'buffer',
   which is an object with beginning and ending addresses, and a method
   (set()), which sets the value of an octet at an address. In many
   cases, a buffer corresponds directly to a portion of host user
   memory. However, DDP does not depend on this; a buffer could be a
   disk file, or anything else that can be viewed as an addressable
   collection of octets. Abstractly, a buffer provides the interface:

```
        typedef struct {
          const address_t start;
          const address_t end;
          void            set(address_t a, data_t v);
        } ddp_buffer_t;
```
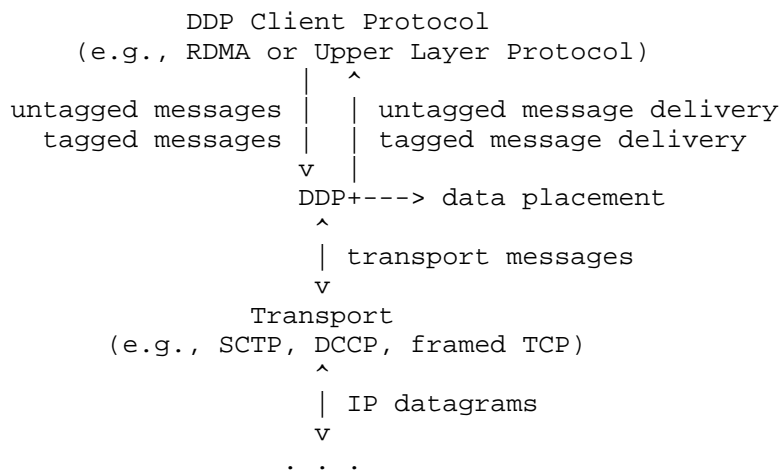
   address_t

      a reference to local memory

   data_t

      an octet data value.

   The protocol layering and in-line data flow of DDP is:

```
                       DDP Client Protocol
                 (e.g., RDMA or Upper Layer Protocol)
                              |  ^
            untagged messages |  | untagged message delivery
              tagged messages |  | tagged message delivery
                              v  |
                            DDP+---> data placement
                             ^
                             | transport messages
                             v
                          Transport
                 (e.g., SCTP, DCCP, framed TCP)
                             ^
                             | IP datagrams
                             v
                            . . .
```

   In addition to in-line data flow, the client protocol registers
   buffers with DDP, and DDP performs buffer update (set()) operations
   as a result of receiving tagged messages.

   DDP messages may be split into multiple, smaller DDP messages, each
   in a separate transport message. However, if the transport is
   unreliable or unordered, messages split across transport messages may
   or may not provide useful behavior, in the same way as splitting
   arbitrary Upper Layer messages across unreliable or unordered
   transport messages may or may not provide useful behavior. In other
   words, the same considerations apply to building client protocols on
   different types of transports with or without the use of DDP.

A DDP message split across transport messages looks like:

```
DDP message:                Transport messages:

  stag=s, offset=o,             message 1:
  notify=y, id=i                  |type=ddp  |
  message=                        |stag=s    |
    |aabbccddee|-------.          |offset=o  |
    ~   ...    ~----.  \          |notify=n  |
    |vvwwxxyyzz|-.  \   \         |id=?      |
                | \  \  '--->|aabbccddee|
                |  \  \     ~    ...   ~
                |     +----->|iijjkkllmm|
                |  |
                |  |
              + |     message 2:
               \ |  |   |type=ddp  |
                \ |  |   |stag=s    |
                 \   +   |offset=o+n|
                  \   \  |notify=y  |
                   \   \ |id=i      |
                    \   '-->|nnooppqqrr|
                     \      ~    ...   ~
                      '---->|vvwwxxyyzz|
```

Although this picture suggests that DDP information is carried in-
line with the message payload, components of the DDP information may
also be in transport-specific fields, or derived from transport-
specific control information if the transport permits.

2.1.1. Transport Operations

For the purposes of this architecture, the transport provides:

```
        void      xpt_send(socket_t s, message_t m);
        message_t xpt_recv(socket_t s);
        msize_t   xpt_max_msize(socket_t s);
```

socket_t

        a transport address, including IP addresses, ports and other
        transport-specific identifiers.

message_t

        a string of octets.

    msize_t (scalar)

        a message size.

    xpt_send(socket_t s, message_t m)

        send a transport message.

    xpt_recv(socket_t s)

        receive a transport message.

    xpt_max_msize(socket_t s)

        get the current maximum transport message size. Corresponds,
        roughly, to the current path Maximum Transfer Unit (PMTU),
        adjusted by underlying protocol overheads.

    Real implementations of xpt_send() and xpt_recv() typically return
    error indications, but that is not relevant to this architecture.

2.1.2. DDP Operations

    The DDP layer provides:

        void   ddp_send(socket_t s, message_t m);
        void   ddp_send_ddp(socket_t s, message_t m, ddp_addr_t d,
                              ddp_notify_t n);
        void   ddp_post_recv(socket_t s, bdesc_t b);
        ddp_ind_t ddp_recv(socket_t s);
        bdesc_t   ddp_register(socket_t s, ddp_buffer_t b);
        void   ddp_deregister(bhand_t bh);
        msizes_t   ddp_max_msizes(socket_t s);

    ddp_addr_t

        the buffer address portion of a tagged message:

                typedef struct {
                  stag_t stag;
                  address_t offset;
                } ddp_addr_t;

    stag_t (scalar)

        a Steering Tag. A stag_t identifies the destination buffer for
        tagged messages. stag_ts are generated when the buffer is
        registered, communicated to the sender by some client protocol

convention and inserted in DDP messages. stag_t values in this
DDP architecture are assumed to be completely opaque to the
client protocol, and implementation-dependent. However,
particular implementations, such as DDP on a multicast transport
(see below), may provide the buffer holder some control in
selecting stag_ts.

ddp_notify_t

the notification portion of a DDP message, used to signal
that the message represents the final fragment of a
multi-segmented DDP message:

```
        typedef struct {
          boolean_t notify;
          ddp_msg_id_t i;
        } ddp_notify_t;
```

ddp_msg_id_t (scalar)

a DDP message identifier. msg_id_ts are chosen by the DDP
message receiver (buffer holder), communicated to the sender by
some client protocol convention and inserted in DDP messages.
Whether a message reception indication is requested for a DDP
message is a matter of client protocol convention. Unlike
stag_ts, the structure of msg_id_ts is opaque to DDP, and
therefore, it is completely in the hands of the client protocol.

bdesc_t

a description of a registered buffer:

```
        typedef struct {
          bhand_t bh;
          ddp_addr_t a;
        } bdesc_t;
```

'a.offset' is the starting offset of the registered buffer,
which may have no relationship to the 'start' or 'end' addresses
of that buffer. However, particular implementations, such as
DDP on a multicast transport (see below), may allow some client
protocol control over the starting offset.

bhand_t

an opaque buffer handle used to deregister a buffer.

recv_message_t

> a description of a completed untagged receive buffer:

```
        typedef struct {
          bdesc_t b;
          length_t l;
        } recv_message_t;
```

ddp_ind_t

> an untagged message, a tagged message reception indication, or a
> tagged message reception error:

```
        typedef union {
          recv_message_t m;
          ddp_msg_id_t i;
          ddp_err_t e;
        } ddp_ind_t;
```

ddp_err_t

> indicates an error while receiving a tagged message, typically
> 'offset' out of bounds, or 'stag' is not registered to the
> socket.

msizes_t

> The maximum untagged and tagged messages that fit in a single
> transport message:

```
        typedef struct {
          msize_t max_untagged;
          msize_t max_tagged;
        } msizes_t;
```

ddp_send(socket_t s, message_t m)

> send an untagged message.

ddp_send_ddp(socket_t s, message_t m, ddp_addr_t d, ddp_notify_t n)

> send a tagged message to remote buffer address d.

    ddp_post_recv(socket_t s, bdesc_t b)

        post a registered buffer to accept a single received untagged
        message. Each buffer is returned to the caller in a ddp_recv()
        untagged message reception indication, in the order in which it
        was posted. The same buffer may be enabled on multiple sockets;
        receipt of an untagged message into the buffer from any of these
        sockets unposts the buffer from all sockets.

    ddp_recv(socket_t s)

        get the next received untagged message, tagged message reception
        indication, or tagged message error.

    ddp_register(socket_t s, ddp_buffer_t b)

        register a buffer for DDP on a socket. The same buffer may be
        registered multiple times on the same or different sockets. The
        same buffer registered on different sockets may result in a
        common registration. Different buffers may also refer to
        portions of the same underlying addressable object (buffer
        aliasing).

    ddp_deregister(bhand_t bh)

        remove a registration from a buffer.

    ddp_max_msizes(socket_t s)

        get the current maximum untagged and tagged message sizes that
        will fit in a single transport message.

2.1.3. Transport Characteristics in DDP

    Certain characteristics of the transport on which DDP is mapped
    determine the nature of the service provided to client protocols.
    Fundamentally, the characteristics of the transport will not be
    changed by the presence of DDP. The choice of transport is therefore
    driven not by DDP, but by the requirements of the Upper Layer, and
    employing the DDP service.

    Specifically, transports are:

      o   reliable or unreliable,

      o   ordered or unordered,

      o   single source or multisource,

          o   single destination or multidestination (multicast or anycast).

   Some transports support several combinations of these
   characteristics. For example, SCTP [SCTP] is reliable, single
   source, single destination (point-to-point) and supports both ordered
   and unordered modes.

   DDP messages carried by transport are framed for processing by the
   receiver, and may be further protected for integrity or privacy in
   accordance with the transport capabilities. DDP does not provide
   such functions.

   In general, transport characteristics equally affect transport and
   DDP message delivery. However, there are several issues specific to
   DDP messages.

   A key component of DDP is how the following operations on the
   receiving side are ordered among themselves, and how they relate to
   corresponding operations on the sending side:

          o   set()s,

          o   untagged message reception indications, and

          o   tagged message reception indications.

   These relationships depend upon the characteristics of the underlying
   transport in a way that is defined by the DDP protocol. For example,
   if the transport is unreliable and unordered, the DDP protocol might
   specify that the client protocol is subject to the consequences of
   transport messages being lost or duplicated, rather than requiring
   that different characteristics be presented to the client protocol.

   Buffer access must be implemented consistently across endpoint IP
   addresses on transports allowing multiple IP addresses per endpoint,
   for example, SCTP. In particular, the Steering Tag must be
   consistently scoped and must address the same buffer across all IP
   address associations belonging to the endpoint. Additionally,
   operation ordering relationships across IP addresses within an
   association (set(), get(), etc.) depend on the underlying transport.
   If the above consistency relationships cannot be maintained by a
   transport endpoint, then the endpoint is unsuitable for a DDP
   connection.

   Multidestination data delivery is a transport characteristic that may
   require specific consideration in a DDP protocol. As mentioned
   above, the basic DDP model assumes that buffer address values
   returned by ddp_register() are opaque to the client protocol, and can

be implementation dependent. The most natural way to map DDP to a
multidestination transport is to require that all receivers produce
the same buffer address when registering a multidestination
destination buffer. Restriction of the DDP model to accommodate
multiple destinations involves engineering tradeoffs comparable to
those of providing non-DDP multidestination transport capability.

A registered buffer is identified within DDP by its stag_t, which in
turn is associated with a socket. Therefore, this registration
grants a capability to the DDP peer, and the socket (using the
underlying properties of its chosen transport and possible security)
identifies the peer and authenticates the stag_t.

The same buffer may be enabled by ddp_post_recv() on multiple
sockets. In this case any ddp_recv() untagged message reception
indication may be provided on a different socket from that on which
the buffer was posted. Such indications are not ordered among
multiple DDP sockets.

When multiple sockets reference an untagged message reception buffer,
local interfaces are responsible for managing the mechanisms of
allocating posted buffers to received untagged messages, the handling
of received untagged messages when no buffer is available, and of
resource management among multiple sockets. Where underprovisioning
of buffers on multiple sockets is allowed, mechanisms should be
provided to manage buffer consumption on a per-socket or group of
related sockets basis.

Architecturally, therefore, DDP is a flexible and general paradigm
that may be applied to any variety of transports. Implementations of
DDP may, however, adapt themselves to these differences in ways
appropriate to each transport. In all cases, the layering of DDP
must continue to express the transport's underlying characteristics.

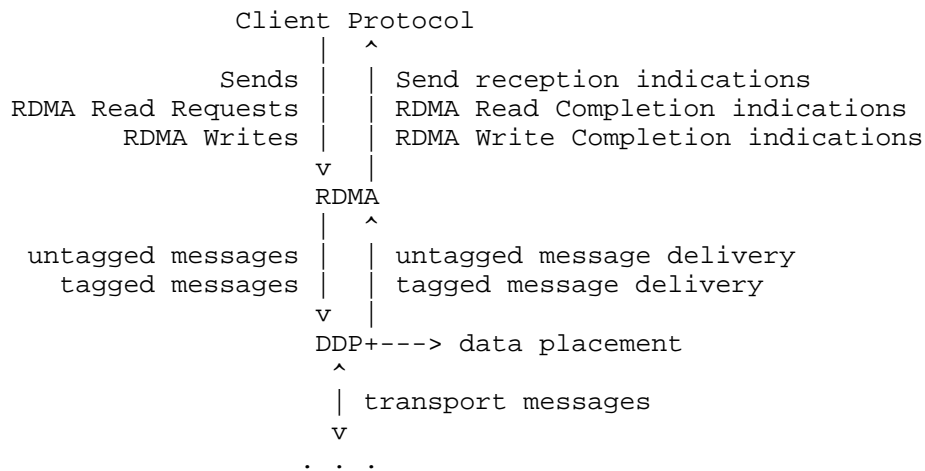2.2. Remote Direct Memory Access (RDMA) Protocol Architecture

   Remote Direct Memory Access (RDMA) extends the capabilities of DDP
   with two primary functions.

   First, it adds the ability to read from buffers registered to a
   socket (RDMA Read). This allows a client protocol to perform
   arbitrary, bidirectional data movement without involving the remote
   client. When RDMA is implemented in hardware, arbitrary data
   movement can be performed without involving the remote host CPU at
   all.

In addition, RDMA specifies a transport-independent untagged message
service (Send) with characteristics that are both very efficient to
implement in hardware, and convenient for client protocols.

The RDMA architecture is patterned after the traditional model for
device programming, where the client requests an operation using
Send-like actions (programmed I/O), the server performs the necessary
data transfers for the operation (DMA reads and writes), and notifies
the client of completion. The programmed I/O+DMA model efficiently
supports a high degree of concurrency and flexibility for both the
client and server, even when operations have a wide range of
intrinsic latencies.

RDMA is layered as a client protocol on top of DDP:

```
                  Client Protocol
                        | ^
               Sends |  | Send reception indications
    RDMA Read Requests |  | RDMA Read Completion indications
          RDMA Writes |  | RDMA Write Completion indications
                      v  |
                      RDMA
                        | ^
      untagged messages |  | untagged message delivery
        tagged messages |  | tagged message delivery
                      v  |
                      DDP+---> data placement
                       ^
                       | transport messages
                       v
                     . . .
```

In addition to in-line data flow, read (get()) and update (set())
operations are performed on buffers registered with RDMA as a result
of RDMA Read Requests and RDMA Writes, respectively.

An RDMA 'buffer' extends a DDP buffer with a get() operation that
retrieves the value of the octet at address 'a':

```
        typedef struct {
          const address_t start;
          const address_t end;
          void            set(address_t a, data_t v);
          data_t          get(address_t a);
        } rdma_buffer_t;
```

2.2.1. RDMA Operations

   The RDMA layer provides:

```
        void        rdma_send(socket_t s, message_t m);
        void        rdma_write(socket_t s, message_t m, ddp_addr_t d,
                        rdma_notify_t n);
        void        rdma_read(socket_t s, ddp_addr_t s, ddp_addr_t d);
        void        rdma_post_recv(socket_t s, bdesc_t b);
        rdma_ind_t  rdma_recv(socket_t s);
        bdesc_t     rdma_register(socket_t s, rdma_buffer_t b,
                        bmode_t mode);
        void        rdma_deregister(bhand_t bh);
        msizes_t    rdma_max_msizes(socket_t s);
```

   Although, for clarity, these data transfer interfaces are
   synchronous, rdma_read() and possibly rdma_send() (in the presence of
   Send flow control) can require an arbitrary amount of time to
   complete. To express the full concurrency and interleaving of RDMA
   data transfer, these interfaces should also be reentrant. For
   example, a client protocol may perform an rdma_send(), while an
   rdma_read() operation is in progress.

   rdma_notify_t

        RDMA Write notification information, used to signal that the
        message represents the final fragment of a multi-segmented RDMA
        message:

```
                typedef struct {
                  boolean_t notify;
                  rdma_write_id_t i;
                } rdma_notify_t;
```

        identical in function to ddp_notify_t, except that the type
        rdma_write_id_t may not be equivalent to ddp_msg_id_t.

   rdma_write_id_t (scalar)

        an RDMA Write identifier.

rdma_ind_t

    a Send message, or an RDMA error:

```
typedef union {
  recv_message_t m;
  rdma_err_t e;
} rdma_ind_t;
```

rdma_err_t

    an RDMA protocol error indication. RDMA errors include buffer
    addressing errors corresponding to ddp_err_ts, and buffer
    protection violations (e.g., RDMA Writing a buffer only
    registered for reading).

bmode_t

    buffer registration mode (permissions). Any combination of
    permitting RDMA Read (BMODE_READ) and RDMA Write (BMODE_WRITE)
    operations.

rdma_send(socket_t s, message_t m)

    send a message, delivering it to the next untagged RDMA buffer
    at the remote peer.

rdma_write(socket_t s, message_t m, ddp_addr_t d, rdma_notify_t n)

    RDMA Write to remote buffer address d.

rdma_read(socket_t s, ddp_addr_t s, length_t l, ddp_addr_t d)

    RDMA Read l octets from remote buffer address s to local buffer
    address d.

rdma_post_recv(socket_t s, bdesc_t b)

    post a registered buffer to accept a single Send message, to be
    filled and returned in-order to a subsequent caller of
    rdma_recv(). As with DDP, buffers may be enabled on multiple
    sockets, in which case ordering guarantees are relaxed. Also as
    with DDP, local interfaces must manage the mechanisms of
    allocation and management of buffers posted to multiple sockets.

```
rdma_recv(socket_t s);
```

>   get the next received Send message, RDMA Write completion
>   identifier, or RDMA error.

```
rdma_register(socket_t s, rdma_buffer_t b, bmode_t mode)
```

>   register a buffer for RDMA on a socket (for read access, write
>   access or both). As with DDP, the same buffer may be registered
>   multiple times on the same or different sockets, and different
>   buffers may refer to portions of the same underlying addressable
>   object.

```
rdma_deregister(bhand_t bh)
```

>   remove a registration from a buffer.

```
rdma_max_msizes(socket_t s)
```

>   get the current maximum Send (max_untagged) and RDMA Read or
>   Write (max_tagged) operations that will fit in a single
>   transport message. The values returned by rdma_max_msizes() are
>   closely related to the values returned by ddp_max_msizes(), but
>   may not be equal.

2.2.2. Transport Characteristics in RDMA

   As with DDP, RDMA can be used on transports with a variety of
   different characteristics that manifest themselves directly in the
   service provided by RDMA. Also, as with DDP, the fundamental
   characteristics of the transport will not be changed by the presence
   of RDMA.

   Like DDP, an RDMA protocol must specify how:

           o   set()s,

           o   get()s,

           o   Send messages, and

           o   RDMA Read completions

   are ordered among themselves and how they relate to corresponding
   operations on the remote peer(s). These relationships are likely to
   be a function of the underlying transport characteristics.

There are some additional characteristics of RDMA that may translate
poorly to unreliable or multipoint transports due to attendant
complexities in managing endpoint state:

   o   Send flow control

   o   RDMA Read

These difficulties can be overcome by placing restrictions on the
service provided by RDMA. However, many RDMA clients, especially
those that separate data transfer and application logic concerns, are
likely to depend upon capabilities only provided by RDMA on a point-
to-point, reliable transport. In other words, many potential Upper
Layers, which might avail themselves of RDMA services, are naturally
already biased toward these transport classes.

3. Security Considerations

Fundamentally, the DDP and RDMA protocols themselves should not
introduce additional vulnerabilities. They are intermediate
protocols and so should not perform or require functions such as
authorization, which are the domain of Upper Layers. However, the
DDP and RDMA protocols should allow mapping by strict Upper Layers
that are not permissive of new vulnerabilities; DDP and RDMAP
implementations should be prohibited from 'cutting corners' that
create new vulnerabilities. Implementations must ensure that only
'supplied' resources (i.e., buffers) can be manipulated by DDP or
RDMAP messages.

System integrity must be maintained in any RDMA solution. Mechanisms
must be specified to prevent RDMA or DDP operations from impairing
system integrity. For example, threats can include potential buffer
reuse or buffer overflow, and are not merely a security issue. Even
trusted peers must not be allowed to damage local integrity. Any DDP
and RDMA protocol must address the issue of giving end-systems and
applications the capabilities to offer protection from such
compromises.

Because a Steering Tag exports access to a buffer, one critical
aspect of security is the scope of this access. It must be possible
to individually control specific attributes of the access provided by
a Steering Tag on the endpoint (socket) on which it was registered,
including remote read access, remote write access, and others that
might be identified. DDP and RDMA specifications must provide both
implementation requirements relevant to this issue, and guidelines to
assist implementors in making the appropriate design decisions.

For example, it must not be possible for DDP to enable evasion of
buffer consistency checks at the recipient. The DDP and RDMA
specifications must allow the recipient to rely on its consistent
buffer contents by explicitly controlling peer access to buffer
regions at appropriate times.

The use of DDP and RDMA on a transport connection may interact with
any security mechanism, and vice-versa. For example, if the security
mechanism is implemented above the transport layer, the DDP and RDMA
headers may not be protected. Therefore, such a layering may be
inappropriate, depending on requirements.

3.1. Security Services

The following end-to-end security services protect DDP and RDMAP
operation streams:

   o   Authentication of the data source, to protect against peer
        impersonation, stream hijacking, and man-in-the-middle attacks
        exploiting capabilities offered by the RDMA implementation.

        Peer connections that do not pass authentication and
        authorization checks must not be permitted to begin processing
        in RDMA mode with an inappropriate endpoint. Once associated,
        peer accesses to buffer regions must be authenticated and made
        subject to authorization checks in the context of the
        association and endpoint (socket) on which they are to be
        performed, prior to any transfer operation or data being
        accessed. The RDMA protocols must ensure that these region
        protections be under strict application control.

   o   Integrity, to protect against modification of the control
        content and buffer content.

        While integrity is of concern to any transport, it is
        important for the DDP and RDMAP protocols that the RDMA
        control information carried in each operation be protected, in
        order to direct the payloads appropriately.

   o   Sequencing, to protect against replay attacks (a special case
        of the above modifications).

   o   Confidentiality, to protect the stream from eavesdropping.

IPsec, operating to secure the connection on a packet-by-packet
basis, is a natural fit to securing RDMA placement, which operates in
conjunction with transport. Because RDMA enables an implementation
to avoid buffering, it is preferable to perform all applicable

security protection prior to processing of each segment by the
transport and RDMA layers. Such a layering enables the most
efficient secure RDMA implementation.

The TLS record protocol, on the other hand, is layered on top of
reliable transports and cannot provide such security assurance until
an entire record is available, which may require the buffering and/or
assembly of several distinct messages prior to TLS processing. This
defers RDMA processing and introduces overheads that RDMA is designed
to avoid. In addition, TLS length restrictions on records themselves
impose additional buffering and processing for long operations that
must span multiple records. TLS therefore is viewed as potentially a
less natural fit for protecting the RDMA protocols.

Any DDP and RDMAP specification must provide the means to satisfy the
above security service requirements.

IPsec is sufficient to provide the required security services to the
DDP and RDMAP protocols, while enabling efficient implementations.

3.2. Error Considerations

Resource issues leading to denial-of-service attacks, overwrites and
other concurrent operations, the ordering of completions as required
by the RDMA protocol, and the granularity of transfer are all within
the required scope of any security analysis of RDMA and DDP.

The RDMA operations require checking of what is essentially user
information, explicitly including addressing information and
operation type (read or write), and implicitly including protection
and attributes. The semantics associated with each class of error
resulting from possible failure of such checks must be clearly
defined, and the expected action to be taken by the protocols in each
case must be specified.

In some cases, this will result in a catastrophic error on the RDMA
association; however, in others, a local or remote error may be
signalled. Certain of these errors may require consideration of
abstract local semantics. The result of the error on the RDMA
association must be carefully specified so as to provide useful
behavior, while not constraining the implementation.

4. Acknowledgements

The authors wish to acknowledge the valuable contributions of Caitlin
Bestler, David Black, Jeff Mogul, and Allyn Romanow.

5. Informative References

   [FCVI]   ANSI Technical Committee T11, "Fibre Channel Standard
             Virtual Interface Architecture Mapping", ANSI/NCITS 357-
             2001, March 2001, available from
             http://www.t11.org/t11/stat.nsf/fcproj.

   [IB]   InfiniBand Trade Association, "InfiniBand Architecture
             Specification Volumes 1 and 2", Release 1.1, November 2002,
             available from http://www.infinibandta.org/specs.

   [MYR]   VMEbus International Trade Association, "Myrinet on VME
             Protocol Specification", ANSI/VITA 26-1998, August 1998,
             available from http://www.myri.com/open-specs.

   [ROM]   Romanow, A., Mogul, J., Talpey, T., and S. Bailey, "Remote
             Direct Memory Access (RDMA) over IP Problem Statement", RFC
             4297, December 2005.

   [SCTP]   Stewart, R., Xie, Q., Morneault, K., Sharp, C.,
             Schwarzbauer, H., Taylor, T., Rytina, I., Kalla, M., Zhang,
             L., and V. Paxson, "Stream Control Transmission Protocol",
             RFC 2960, October 2000.

   [SDP]   InfiniBand Trade Association, "Sockets Direct Protocol
             v1.0", Annex A of InfiniBand Architecture Specification
             Volume 1, Release 1.1, November 2002, available from
             http://www.infinibandta.org/specs.

   [SRVNET] R. Horst, "TNet: A reliable system area network", IEEE
             Micro, pp. 37-45, February 1995.

   [VI]   D. Cameron and G. Regnier, "The Virtual Interface
             Architecture", ISBN 0971288704, Intel Press, April 2002,
             more info at http://www.intel.com/intelpress/via/.

Authors' Addresses

   Stephen Bailey
   Sandburst Corporation
   600 Federal Street
   Andover, MA   01810 USA
   USA

   Phone: +1 978 689 1614
   EMail: steph@sandburst.com

   Tom Talpey
   Network Appliance
   1601 Trapelo Road
   Waltham, MA   02451 USA

   Phone: +1 781 768 5329
   EMail: thomas.talpey@netapp.com