NFS over RDMA

Brent Callaghan, Theresa Lingutla-Raj, Alex Chiu,
Peter Staubach, Omer Asad
Sun Microsystems, Inc.

Abstract

The NFS filesystem was designed as a work-group filesystem, making a central file store available to and shared between a number of client workstations. However, more recently NFS has grown in popularity in the server room, connecting large application servers with back-end file servers. In this environment, where high-speed access to data is critical, high capacity interconnects like gigabit Ethernet, Fibre Channel and Infiniband are to be expected. With RDMA technology we can fully utilize the data capacity of these interconnects, while providing relief for host CPU and memory buses from the demands of managing a "fire hose" of data.

Here we describe the use of RDMA as an RPC transport layer. We show that the NFS protocol running over this new transport achieves higher throughput than a conventional TCP transport along with a reduction in CPU utilization. The benefits of an RDMA transport are enjoyed not only by the applications that use NFS mounts, but extend to any RPC service that requires high speed and efficient transfer of large volumes of data.

1 Introduction

The NFS protocol was developed on 10 megabit Ethernet and, for most implementations, the migration to "fast" 100 megabit Ethernet was no surprise: an expected tenfold improvement in performance. However the next jump to gigabit Ethernet did not provide another tenfold increase in performance. Instead, we saw only a typical 50% of bandwidth utilization and much heavier use of host CPU. Clearly, we need to fix the performance shortfall before the next order of magnitude leap in network performance to 10 Gbps.

Guiding us in this effort was a belief that the NFS protocol is not inherently slow, since it is not significantly different from other high-performance data movement protocols, like SCSI over Fibre Channel or iSCSI, when comparing data payload with header size. But its implementation as a LAN based protocol using a generic, host-resident TCP/IP stack needed a re-think. Although techniques such as Large Send Off-load (LSO [3]) can reduce processing overhead significantly on the transmit side, there is little that can be done on the receive side without the introduction of an NFS-aware network card. A good summary of protocol processing improvements for NFS aimed at data copy reduction is presented in [1,2].

RDMA (Remote Direct Memory Access) is becoming popular as a technology for memory-to-memory transfer of data over high speed networks. By offloading the data movement into network hardware and providing direct data placement, RDMA relieves not only the host CPU but also reduces contention for the host memory and I/O buses. RDMA is a feature of new Infiniband network fabrics and is being developed through the IETF as a new protocol to run over TCP/IP connections.

We saw RDMA as an opportunity to move the responsibility for NFS data movement from a host stack to specialized network hardware. However we had a strong desire to bring RDMA to NFS in a more subtle way – without changing the NFS protocol as described in RFC 1094 (NFS version 2), RFC 1813 (NFS version 3) or RFC 3530 (NFS version 4). Although the addition of RDMA as a new RPC transport does not affect the logical representation of the protocol, it does affect the representation of the protocol on the wire, so interoperability depends on the client and server using the same transport. Similarly, NFS is already used over UDP and TCP connections but a client that uses only UDP cannot inter-operate with a server that uses only TCP. Additionally we wanted an RDMA enhanced NFS to be fully compatible with existing applications and NFS administration practice.

In the following sections we introduce the ONC RPC protocol upon which NFS is based. We start by explaining how RPC data is moved by conventional transports like

UDP and TCP. Then we introduce RDMA as a new RPC transport explaining how it achieves direct data placement. In section 7 we provide some performance measurements that demonstrate the effectiveness of this new transport.

2 NFS as an ONC RPC Protocol

ONC RPC (RFC 1831 [12]) is often overlooked as the foundation of the NFS protocol. We talk about "NFS over TCP" omitting the underlying RPC protocol layer. The NFS protocol is just one of a family of protocols that comprise NFS in the large. Every NFS mount is preceded by use of the MOUNT protocol to obtain an initial filehandle. File locks are managed by the Network Lock Manager Protocol (NLM) and Access Control Lists by the NFS_ACL protocol. The NFS protocol itself exists in three versions. The most recent, NFS version 4 (RFC 3530 [11]), integrates most of these NFS "companion" protocols into a single protocol. All of these protocols are implemented as instances of RPC protocols. RPC is not complete without the XDR layer (RFC 1832 [13]), which determines the representation of RPC data on the network.

The RPC layer provides a degree of transport independence. It is possible to plug new transports into RPC without unduly affecting the protocols implemented on top of RPC. Protocols implemented on RPC can be switched from one transport to another with little or no change to the protocol implementation.

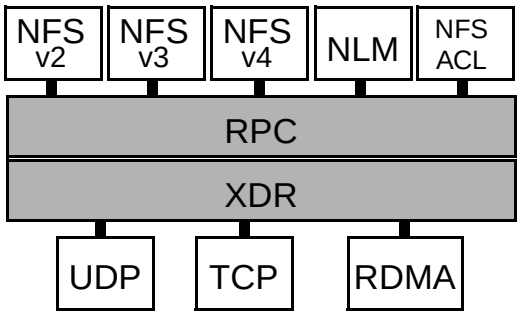

Figure 1. NFS is a family of protocols layered over RPC. The XDR layer encodes RPC arguments and results onto one of several RPC transports.

2.1 RPC Programming Model

An RPC protocol like NFS is defined by an XDR language specification. The XDR language describes both the procedures making up the protocol, as well as the data structures comprising the arguments and results for each of those procedures. For example, the following XDR language fragment describes the NFS version 3 READ procedure

```
READ3res NFSPROC3_READ(READ3args) = 6;
```

which is procedure 6 of 21 procedures in the protocol. The READ procedure arguments and results are defined by two structures:

```
struct READ3args {
  nfs_fh3  file;
  offset3  offset;
  count3   count;
};

struct READ3resok {
  post_op_attr   file_attributes;
  count3         count;
  bool           eof;
  opaque         data<>;
};
```

The final "opaque" item describes the file data that is returned from the NFS server.

The XDR language compiler converts this description into a C header file that describes the arguments and results as C structures. For example, the C structure for `READ3resok` becomes:

```
struct READ3resok {
  post_op_attr file_attributes;
  count3 count;
  bool_t eof;

  struct {
    u_int data_len;
    char *data_val;
  } data;
}
```

Notice that the opaque data item that represents the file data returned by the READ procedure has been split into two fields: a length field, `data_len`, and a character pointer to a data buffer, `data_val`. Additionally, the compiler produces XDR code to marshall data into and out of these data structures.

```
ret = xdr_bytes(xdrs,
   (char **)&objp->data.data_val,
   &objp->data.data_len, MAXCNT);
```

Where `xdr_bytes` is an XDR library function that encodes or decodes an arbitrary length array of bytes.

2.2 XDR Encoding and Decoding

XDR encodes application data into a canonical form suitable for transmission from one computer to another. Integers are represented in big-endian, network byte order. All XDR data is 32 bit aligned.

XDR functions like `xdr_bytes` are used to encode data into an XDR byte stream. Additionally the same function can be used to move data from a stream into a supplied buffer. It is common for an application to provide a null

address for a receiving buffer. In this instance the XDR code will itself allocate memory to receive data from the XDR stream and return the address of the buffer to the caller.

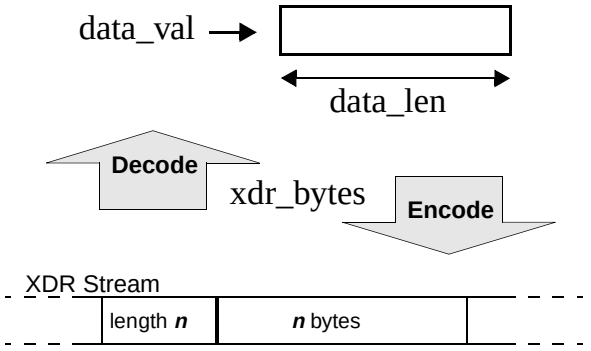

Figure 2. XDR Encoding and decoding.

XDR has an encode/decode function for each supported data type: integers, booleans, floating point numbers and strings. Using these primitive data types, XDR supports encoding of more complex structures such as pointers, arrays and linked lists. All XDR data movement funnels through two pairs of stream encoding functions: `putint32`/`getint32` for encoding/decoding 32 bit integers, and `putbytes`/`getbytes` for arbitrary strings of bytes. For instance, when `xdr_bytes` encodes an arbitrary sequence of bytes it invokes `putint32` to encode the byte count onto the stream as an integer, followed by a call to `putbytes` to deposit the bytes themselves onto the stream.

2.3 The XDR Stream

The XDR encoding and decoding functions have an abstract view of an XDR stream. It is considered to be just a stream of bytes with a beginning, a current offset, and an end. The stream encoding functions provide XDR some flexibility in mapping this abstract representation of a stream onto more convenient data structures as shown in figure 3.

An xdr_mem stream is the simplest kind of XDR stream. It is commonly used for small XDR streams. An XDR mblk stream is useful when the stream is not contiguous or if the size of the stream is not known at encoding time. To support the RDMA transport, we added an XDR RDMA stream.

2.4 RPC Headers and the RPC Protocol

The RPC protocol uses XDR to encode and decode the arguments and results to remote procedure calls. The RPC protocol itself also uses XDR to encode the RPC headers. An RPC call message header contains the RPC program number, program version and procedure number and security field. The first item on all RPC messages is a transaction ID (XID). This is generated by the client at the front of an RPC call message and is returned by the server at the start of the reply message. The client uses the XID to match the reply message with its corresponding call message, since multiple RPC messages can be in flight over a single network connection.

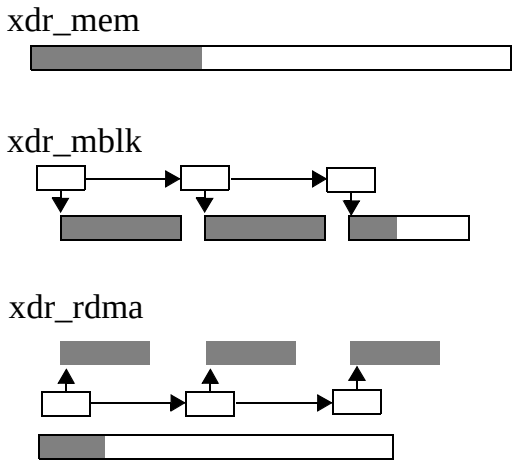

Figure 3. Three kinds of XDR streams. The simplest is the XDR mem stream – a contiguous memory buffer. The XDR mblk stream provides an extensible stream built from a chain of memory buffers. The xdr_rdma stream isolates large chunks of data for direct placement.

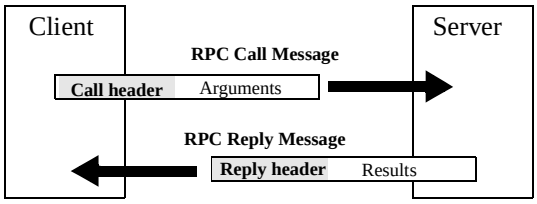

Figure 4. An RPC transaction comprises a call message with arguments and a reply message with results. Each has an XDR encoded header.

2.5 RPC Transport Encapsulation

The most appropriate way to move RPC messages is specific to each transport. Over UDP the message is contained within a datagram, which imposes an upper limit on RPC message size of 64 KB. Since TCP is a stream oriented protocol, RPC messages are defined by a simple record marking protocol consisting of a 32 bit length field followed by the message bytes.

Although the UDP and TCP transports move RPC messages differently, they have no logical effect on the RPC protocol. The message is the same, even if the vehicle is different. So in utilizing RDMA as a new RPC transport, we can leverage all the strengths of RDMA without concern for encapsulations used by other transports.

3 The RDMA Transport

Our goal in building an RDMA transport was to achieve the following:

- Lower CPU utilization by offloading network protocol processing onto RDMA hardware.
- Increase in throughput from reduction in memory copies, through the use of direct data placement.

3.1 RDMA Operations

The classic RDMA model features a SEND operation for conveying messages to pre-posted receive buffers in addition to READ and WRITE operations for moving large chunks of data with direct data placement.

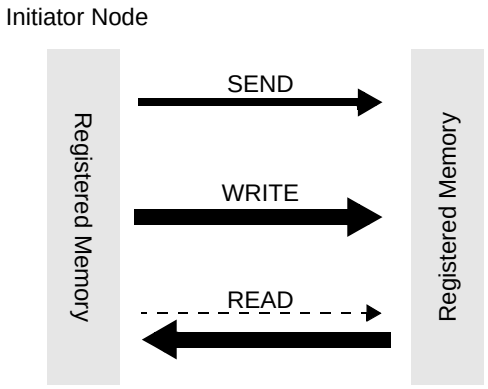

Figure 5. RDMA operations

The SEND operation is the one most familiar to conventional networking practice. A message is sent to a pre-posted buffer on the receiving node, causing a completion interrupt, i.e. the receiver is notified that the message has arrived. It is incumbent on the receiving node to post a receiving buffer large enough to hold the message. If the buffer is too small, an error will result and the message will be lost. The READ and WRITE operations are unique to RDMA networking. The initiator provides not only the address and size of the data source buffer, but also provides the address of the receiving buffer, thereby enabling direct data placement and avoiding unnecessary copies through staging buffers.

The WRITE operation does not signal a completion interrupt at the receiver. This allows the initiator to avoid unnecessarily interrupting the receiver – particularly if the WRITE is one of many comprising a data stream. The initiator can signal completion with a SEND message. Similarly, a READ operation does not interrupt the node at the data source, though the initiator of a READ is notified when the data has arrived.

3.2 Registered Memory

RDMA operations are constrained to operate only on buffers in a registered memory region. Registered or “pinned” memory has a fixed virtual to physical address mapping for the duration of the registration. This allows RDMA hardware to access memory independently of the host operating system. Registration also allows the RDMA hardware to cache virtual to physical mappings. The process of registering and de-registering memory regions can involve large system overhead, so it is advantageous to keep registration activity to a minimum. For instance, a large number of buffers can be allocated and freed within a large, pre-registered memory pool.

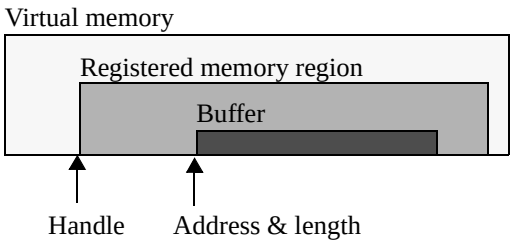

Figure 6. A registered memory region is identified with a handle. Buffers within this region are identified by a virtual address or offset within the region and a length.

A registered memory region is identified with a *handle* that is obtained when the registration is done. Buffers within this memory region are identified with a virtual address and a length. An RDMA READ or WRITE operation refers to the source and destination buffers with two triplets: the memory region handle, address and length of the source buffer, and the handle, address and length of the destination buffer.

3.3 Small RPC Messages

The NFS protocol has a characteristic that appears common to many RPC protocols: most of the procedures in the protocol feature small call and reply messages. Of the 21 procedures in the NFS version 3 protocol, only four can carry any significant amount of data: **read**, **write**, **readdir** and **readdirplus**. The most commonly used procedures, like **lookup** and **getattr** use no more than a couple of hundred bytes in their call or reply message. These small messages derive no advantage from the direct data placement offered by RDMA READ and WRITE, so the obvious candidate is to move small RPC messages with a SEND operation.

Although the movement of RPC messages with SEND operations is relatively straightforward, the goal of direct data placement for the very large data buffers in NFS **read** and **write** requests cannot be achieved without the use of RDMA direct placement operations like READ and WRITE. However, at the level of XDR data encoding, it is easy to identify these large buffers or *chunks* of data and move them via direct placement.

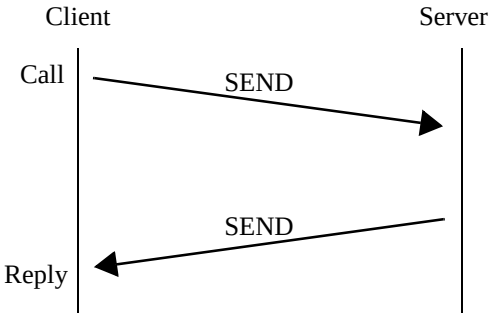

Figure 7. Small RPC messages over RDMA

3.4 Identifying Data Chunks

The direct placement offered by RDMA READ and WRITE provide a significant benefit in data copy reduction. However, these operations are not without cost. Since they are distinct network operations, they have a transaction cost not only in scheduling the operation, but within the RDMA hardware itself. These overheads must be compared against the cost of leaving the data in-line with other RPC data within a SEND buffer and a copy operation in the client, server or both.

Small chunks of data like 20 character filenames or 32 character filehandles are best left in-line and copied. However, much larger chunks of data like 32KB of file data from an NFS **read** request are much more expensive to copy and direct data placement is favored. The size of a data chunk that justifies direct data placement must be greater than the *minimum chunk size*. This minimum chunk size will vary depending on factors like the performance of the host CPU, memory bus, memory registration overhead and the performance of the RDMA hardware.We use a default chunk size of 1 KB, but that value is tunable and easily changed. We plan additional performance work to identify an optimal chunk size for each RDMA transport.

In our prototype, we implemented an XDR RDMA data stream that identifies large chunks of data so that RDMA direct placement can move them. The XDR routines move RPC call or reply arguments into or out of an XDR stream. Integers are encoded with `putint32` and byte sequences comprising strings or opaque buffers are encoded with `putbytes`. The putbytes routine is quite simple: given the address of a buffer and its length, it moves the bytes into the XDR stream. The `putbytes` function can identify a chunk of data suitable for direct data placement simply by comparing its length with the minimum chunk size.

4 The Chunk List

The chunk list is a linked list that identifies one or more chunks of data in RPC call or reply arguments that are large enough to be considered candidates for direct placement. Rather than move these chunks in-line with other data in an RPC message, these large buffers are moved separately via an RDMA direct placement operation. The chunk list is encoded as an XDR linked list before it is sent across the network with the de-chunked RPC message.

4.1 Message Encoding

When the `putbytes` function identifies a data chunk smaller than the minimum chunk size, it just copies the bytes in-line into the XDR byte stream. However, if the chunk is greater or equal to the minimum chunk size, it leaves the data in-place – the chunk is not copied into the XDR stream. Instead, the chunk memory is registered in preparation for RDMA transfer. The triplet comprising memory handle, address and length of the registered chunk is recorded in a chunk list entry along with the current offset into the XDR byte stream.

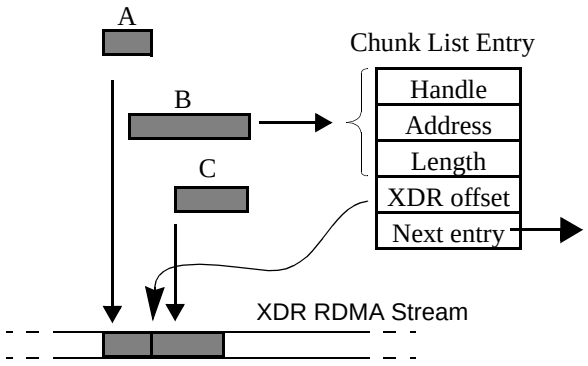

Figure 8. Encoding chunks: inline and in the chunk list

In figure 8, three data chunks are encoded onto an XDR stream. The first chunk, A, is smaller than the minimum chunk size, so it is copied into the stream. Chunk B is larger than the minimum chunk size, so it is registered and recorded in a new chunk list entry. The last chunk, C, which is smaller than the minimum chunk size, is copied into the stream.

When XDR encoding is complete, the result is an XDR stream for the RPC message containing only small chunks of data and a chunk list that records the addresses of the large chunks along with the positions they would have occupied in the XDR stream. The RPC message, with large chunks removed, is transmitted in an RDMA SEND along with the XDR encoded chunk list. The buffers containing large chunks remain behind.

4.2 Message Decoding

The receiver first decodes the chunk list, then with the chunk list in hand, proceeds to decode the XDR stream comprising the RPC message itself. The XDR `getbytes` function is called whenever a data chunk needs to be decoded from the data stream. The bytes are moved to a receiving buffer supplied by the application.

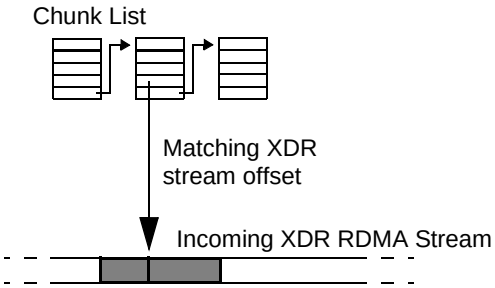

Figure 9. Message decoding. When the XDR offset in a chunk list entry matches the offset in the incoming XDR stream, an RDMA READ operation is initiated.

The `getbytes` function first compares the XDR offset in the *current* chunk list entry with the current XDR offset. If the offsets match, then instead of moving bytes from the byte stream, an RDMA READ operation is initiated using the handle, address and length triplet from the current chunk list entry. The destination buffer is identified by the `getbytes` caller. The chunk list is assumed to be ordered with ascending XDR offsets.

4.3 The Protocol in Action

Where the RPC messages contain no data chunks exceeding the minimum chunk size, the RPC call and reply is just a pair of SEND operations as illustrated in figure 7. If the client call message contains a large chunk of data (like an NFS **write** request), then the server will pull the chunk with an RDMA READ. If the reply contains a large chunk (like an NFS **read** reply), then the client will pull the file data from the server and direct place it into the destination buffer. The illustrations shown in figure 10 demonstrate the sequence in which an RPC message that contains a large chunk will be followed by an RDMA READ operation to move the chunk. The examples show a single chunk being moved, but it is possible that an RPC message might have multiple large chunks. Each large chunk in a message will elicit a corresponding RDMA READ.

4.3.1 RPC Done Message

In figure 10 the second sequence, showing a large chunk in the reply, concludes with a Done message from the client to the server. The purpose of this message is to notify the server that the client has completed its READ operations. The server cannot deregister and free the memory held by the chunks being READ until it receives the Done message. A Done message is not required for large chunks in the arguments of an RPC call message because the RPC reply is implicit notification that the server has completed its READ operations.

The Done message introduces no additional latency into an RPC transaction since the client does not need to wait for a response from the server. To avoid the possibility of a lost Done message from tying up the server's chunk buffers indefinitely, we attached a timeout to the wait for this message. If the Done message is not received within the time allowed, then the server assumes that the client has either completed the READ(s) and the Done message is lost, or for some reason the client was unable to initiate the READ(s) and complete the RPC transaction. The server frees the chunk buffers and any other state pending completion of the transaction. If the timeout was too short, then the client will receive an RDMA error in response to a READ attempt on a non-existent chunk buffer. In this case, the client can recover by re-establishing the RDMA connection and retransmitting the RPC call.

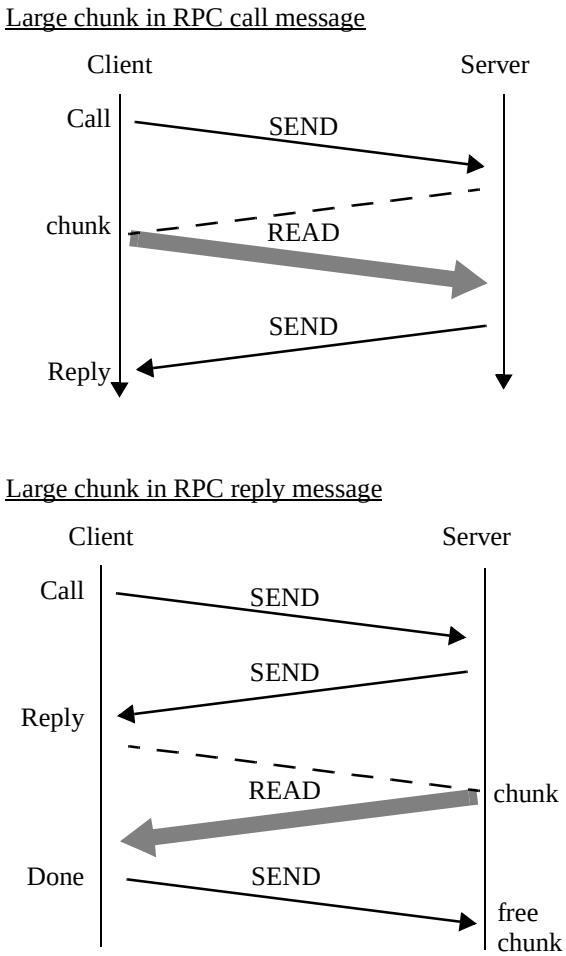

Figure 10. Chunks in RPC call or reply messages

4.4 RDMA Transport Header

The RDMA transport requires that additional data accompany the RPC message itself. If the message includes large chunks, then a chunk list must be transmitted with the dechunked RPC message. This additional data is prepended to the RPC message in an RDMA header which contains the following data:

- Transaction ID (XID) of the RPC message. Although the XID is also included in the RPC header, it is needed early in the message decode and more readily accessible at the front of the RDMA header.

- Version of the RDMA header. Allows subsequent extensions to the header while maintaining backward compatibility.
- Message Type. Has one of two values. If RDMA_MSG then a chunk list and RPC message follow the header. If RDMA_DONE then the RDMA header is a Done message.
- Chunk list as described in section 4.
- The RPC message with large chunks absent.

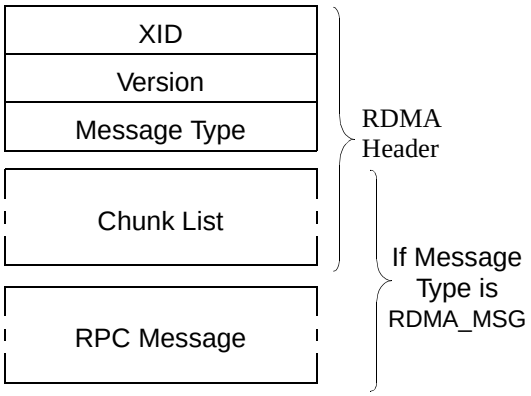

Figure 11. RDMA Transport Header.

The RDMA transport header is shown in figure 11. Since the components of the transport header are XDR encoded, the following XDR language specification describes the header:

```
struct rdma_msg {
  uint32       rc_xid;
  uint32       rc_vers;
  rdma_body    rc_body;
};

enum rdma_proc {
  RDMA_MSG     =0, /* RPC Call or Reply Message */
  RDMA_DONE    =1, /* Client signals completion */
};

union rdma_body switch (rdma_proc proc)
case RDMA_MSG:
  clist_type   rd_msg;
case RDMA_DONE:
  void;
};

struct clist_type {
  clist        *rc_clist; /* Chunk list */
  rpc_msg      rc_msg;    /* RPC message */
};

struct clist {
  uint32       c_xdroff;  /* XDR stream offset */
  uint32       c_len;     /* Chunk length in bytes */
  uint32       c_shandle; /* Source handle */
  uint64       c_saddr;   /* Source address */
  clist        *c_next;   /* Next chunk */
};
```

4.5 Sending the RDMA Message

The XDR encoding of an RDMA RPC message delivers a chunk list and the RPC message (without chunks) encoded in a separate buffer. Before it can be transmitted with the RPC message, the chunk list must be XDR encoded. The protocol requires that the RPC message follow the encoded chunk list when transmitted. This presents an interesting encoding problem because the chunk list is variable length – the number of chunks cannot be known until the RPC message has been encoded. Hence the correct offset to begin encoding the XDR stream for the RPC message cannot be known ahead of time.

The problem can be resolved by encoding into separate buffers which are then assembled into a contiguous RDMA transport header with a *gather* operation, which is supported by RDMA SEND. When an RDMA SEND is

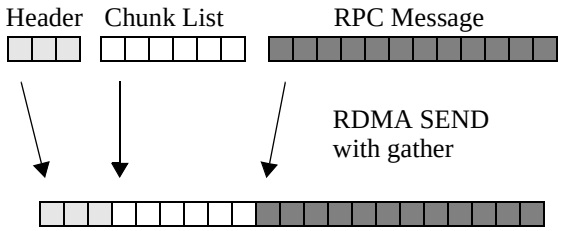

Figure 12. SEND gather assembles RDMA message

initiated, it takes a descriptor that can reference a number of discrete buffers that are transmitted, in order, to the receiving system as a contiguous sequence of bytes.

4.6 Long RPC Messages

NFS is typical of RPC protocols: most messages are quite short. Even very long messages associated with NFS **read** or write procedures become short if the file data in their payload is referenced by a chunk list entry and moved separately via an RDMA READ. Since these “de-chunked” messages are so short, some economy is possible in sizing receive buffers for these messages. RDMA SEND messages must fit within buffers posted on a receive queue. These posted buffers must be large enough for any SEND payload, yet not so excessively large that buffer capacity is wasted. Nearly all de-chunked NFS messages can easily fit within a 1 KB buffer. However, there are exceptions.

4.6.1 Messages without Large Chunks

The NFS **readdir** procedure is used to read directory contents. The RPC reply message contains a list of file names and fileids. The **readdirplus** procedure returns a larger reply with a complete set of file attributes for each directory entry. Even modest size directories can result in reply messages many kilobytes in length. These directory names and attributes are much smaller than the minimum chunk size, so they are not candidates for direct data placement via the chunk list.

The **readdir** procedure is infrequently used. The SPEC SFS 2.0 benchmark mix of operations uses **readdir** or **readdirplus** only 10% of the time. If the client posted receive buffers large enough to contain **readdir** replies, then 90% of received messages would occupy only a tiny fraction of the receive buffer space.

The NFS version 4 protocol presents a slightly different problem with message sizes. Its **compound** procedure provides clients with a lot of flexibility in constructing RPC messages. The **compound** procedure and its reply are of indeterminate length. Our observation is that most implementation will use modest sized **compound** requests most of the time. But occasionally, a very large request or reply may be justified.

A more efficient strategy is to post receive buffers large enough to contain the bulk of RPC messages typical of the protocol, and find another way to move the minority of large messages. Fortunately, the chunk list already provides this mechanism.

4.6.2 The RPC Message as a Chunk

The `xdr_sizeof` routine allows the encoded size of the RPC message to be determined in advance. This routine goes through the encoding process without moving any data. It just tallies the sizes of all of the objects. The implementation of `xdr_sizeof` in the XDR RDMA stream ignores chunks that meet the minimum chunk size. They are not included in the overall total because they take no space in the receive buffer. If the encoded size of the RPC message would exceed the size of the posted receive buffer, then the entire message can be encoded into a large, registered buffer and sent as a chunk. This message chunk must occupy the first entry in the chunk list and its XDR offset value must be zero.

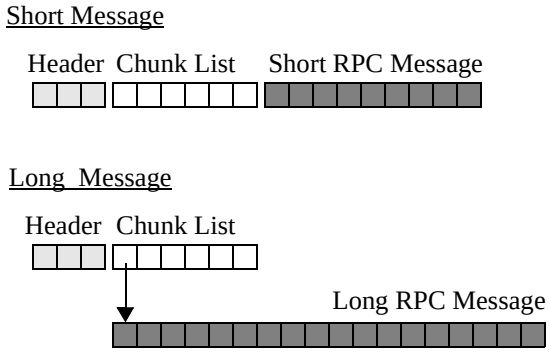

Figure 13. A long RPC message is transferred as a chunk with an XDR offset of zero.

Although a long message must be the first chunk in the chunk list, subsequent entries in the list may refer to other large chunks that require direct placement.

The RDMA transport header of a long message will have a chunk list but no in-line RPC message. The receiver of a long message will decode the chunk list first, but it must check the first chunk list entry to see if the RPC header follows. If the XDR offset value in this first entry is zero, then the receiver must allocate and register a buffer to hold the long RPC message, then initiate an RDMA READ operation to load the message. Once the message is loaded, the decode of the RPC message can then proceed normally.

5 Security

A transport like RDMA that exchanges memory addresses and provides direct access to host memory by remote systems would appear to invite attack. Hence, it is important to be able to protect data on this transport from prying eyes and disruption from unauthorized data modification.

RPC features a security framework called RPCSEC_GSS (RFC 2203 [14]) which is compatible with security mechanisms available though the GSS-API (RFC 2743[16]). Using a security mechanism like Kerberos version 5, RPCSEC_GSS can provide message authentication, integrity or privacy or any combination of these. Since RPC can provide its own security, it does not rely on connection-based security provided at the RDMA transport layer.

Message authentication requires only the use of a secure credential within the RPC header. Message integrity protection requires that a checksum be computed across the body of the RPC message which is then encrypted and transmitted with the message. Message privacy requires that the entire message body be encrypted.

Both integrity and privacy protection require that the RPC message body be *wrapped*. The message data is first encoded into a large buffer. If the message is to be integrity protected, a checksum is computed across the buffer. If privacy is required, then the buffer is encrypted. Then the buffer is re-encoded as a large, opaque chunk before it is transmitted as an RPC message. At the receiver, an *unwrap* function first verifies the integrity of the received message or decrypts it before subjecting it to a normal XDR decode.

When XDR encoding for the transport, only a large, opaque buffer is visible to the XDR routines. Although the large buffer will possibly meet the minimum chunk size requirement and be transmitted via the chunk list, no direct placement of any embedded chunks will be possible.

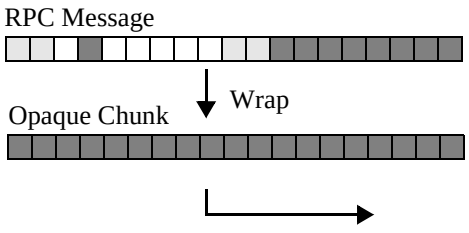

Figure 14. The wrap function of RPC integrity or privacy protection reduces an RPC message to an opaque chunk.

Although RPC integrity and privacy can be used with the RDMA transport, the checksum and encryption processes

exact a toll in host CPU utilization and data movement. While the RDMA transport may provide some performance advantage through protocol offloading into RDMA hardware, there is no hope of direct data placement for large data chunks in a wrapped message.

5.1 Connection based Security

Connection based security protects all data running over a connection from interference or scrutiny. Such schemes as SSL, SSH and IPsec are ubiquitous and commonly implemented in network hardware. IPsec (RFC 2401 [15]) seems particularly suitable for implementation within RDMA NICs. IPsec implemented within an RDMA NIC can provide a similar level of integrity or privacy protection as RPCSEC_GSS while offloading the processing involved and preserving the direct data placement capability.

However, connection based security does not meet the authentication requirement of NFS. Only a single security principal can be used for the entire connection.This is not sufficient for NFS, since a single connection may be shared by multiple users. A compromise is to use the per-call RPC credential to authenticate each user, while protecting data and privacy with connection-based security.

The NFS version 4 working group within the IETF is considering a new RPCSEC_GSS mechanism called CCM: The Credential Cache GSS Mechanism. It allows a client and server to detect the presence of a secure connection and avoid the use of RPC level integrity and privacy protection.

6 Solaris Implementation

Our first implementation of NFS/RDMA was developed using the Solaris Remote Shared Memory (RSM) API [7] running over the Dolphin SCI interconnect on a Sun Cluster. Our goal was to validate the approach of using RDMA as a new RPC transport to benefit NFS.

Using our experience with RSM, we moved to a kernel implementation using kVIPL API [8] driving the Emulex GN9000/VI NIC. This NIC supports two drivers: a kVIPL driver for RDMA and a DLPI driver that allows the host TCP stack to use the gigabit Ethernet. Since our implementation of the RDMA as an RPC transport is entirely contained within the kernel, we still need TCP-based RPC to handle the NFS MOUNT protocol, which is implemented at user-level in the UNIX mount command and mount daemon. We verified that the NFS/RDMA implementation is functionally complete by successfully running the Connectathon test suite [9] upon an RDMA NFS mount.

Concurrent with the kVIPL implementation work, we developed an Infiniband version of the RDMA transport that runs over Mellanox Gamla (2.5 Gb/sec) and Tavor (10 Gb/sec) cards. We have not yet begun performance measurements on our Infiniband implementation.

Underlying our implementation is a desire to make the use of RDMA transports invisible not only to applications using NFS mounts, but also to system administrators. Just as it is difficult to tell whether an NFS mount is using UDP or TCP, we expect the RDMA transport to be noticed only by virtue of its exceptional performance. System administrators should not need to modify automounter maps or update *fstab* or *vfstab* files.

7 Performance

Our performance measurement strategy was to compare the performance of the new RDMA transport with a conventional TCP transport. Initially, we have focused on sequential read throughput and CPU utilization.

7.1 Configuration

We used two SunBlade 1000 systems for the client and server. Each system had two 750 Mhz UltraSparc-III CPUs (SPECint2000 396). On the client we disabled one CPU to simplify measurements of CPU utilization. While the client was installed with 512 MB of memory, we installed 4 GB on the server to allow the caching of large files in memory. On both systems we installed Solaris 9 along with a new kernel RPC module containing the code to implement the RDMA transport.

These systems were connected back-to-back with two gigabit Ethernet fibre connections. One connection used Sun GigaSwift Ethernet adapters. The other, Emulex GN9000/VI adapters for RDMA traffic.The SunBlade 1000 has four 64-bit 33 Mhz PCI slots and one 64-bit 66 Mhz slot. Concerned that the 33 Mhz slot might be a bottleneck, we measured the performance of the GigaSwift and Emulex cards only when occupying the 66 Mhz slot.

In both the GigaSwift and Emulex drivers we enabled the use of Jumbo frames which extend the normal Ethernet MTU of 1500 bytes up to 9000 bytes – 6 times normal size. We evaluated the GN9000/VI performance with normal Ethernet frames and observed a 40% drop in throughput. Since our goal was to evaluate RDMA performance rather than the performance of TCP stacks, we felt justified in choosing Jumbo frames.

On the server we NFS-exported a directory containing a one gigabyte file. On the client we mounted the exported directory using either of two hostnames: one mapping to the address of the server GigaSwift card and the other to its Emulex card. For the Emulex mount we used the mount

option `proto=rdma` to force the use of the RDMA transport. For all measurements, we used the NFS version 3.

7.2 Tunables

The NFS version 3 protocol allows the client and server to negotiate a suitable **read** size. We configured the server to accept read sizes up to one megabyte. The Solaris client default is to generate 32 KB **read**s. However, we were interested in the effect on performance of a variety of **read** sizes. So we used the Solaris `adb` debugger to modify the kernel variables `nfs3_bsize` and `nfs3_max_transfer_size` in a range from 512 byte reads to 1 MB.

Another NFS performance tunable is the number of reads ahead. When the client detects that an application is accessing a file sequentially, it initiates asynchronous kernel threads to read further ahead in the file in anticipation of future application access. Ideally, the client will generate sufficient reads-ahead to use all the reply bandwidth, without overloading the NFS server. The Solaris client default is one read-ahead. We were interested in determining the optimum number of reads ahead for each transport, so we varied the kernel variables `nfs3_nra` and `nfs3_max_threads` in a range from 0 reads ahead to 16.

7.3 Measuring Read Throughput

We wanted to find the combination of read size and read-ahead values that yielded the best read throughput for each transport. To do this, we set up a benchmark script that used a small C program, `simple_read`, that read the file sequentially. We averaged measurements over three runs of `simple_read` to achieve consistent numbers. We found less than 3% variation in the numbers. The throughput was computed by dividing the file size (1 GB) by the run-time. We ran the benchmark script collecting throughput and CPU utilization for a range of read sizes and read-ahead.

To avoid cache effects, the script mounted the filesystem prior to each run of `simple_read`, then unmounted it immediately after. To eliminate the effects of disk I/O on the server, we warmed the server memory cache with a blind run of `simple_read` before capturing any measurements. Since the server was endowed with 4 GB of memory, we were confident that the entire 1 GB file would be cached.

Our measurements of read throughput using the GigaSwift connection are shown in figure 15. Throughput appears to peak at 60 MB/sec with 256 KB reads and 4 reads-ahead. The same test using the RDMA transport via an Emulex mount yielded the results shown in figure 16. Peak throughput is achieved with 256 KB reads and 8 reads-ahead, and the RDMA throughput is more than 60% greater at 102 MB/sec.

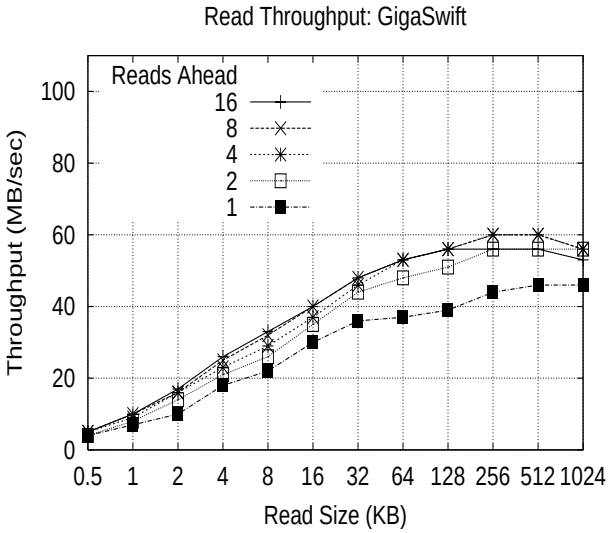

Figure 15. GigaSwift Throughput

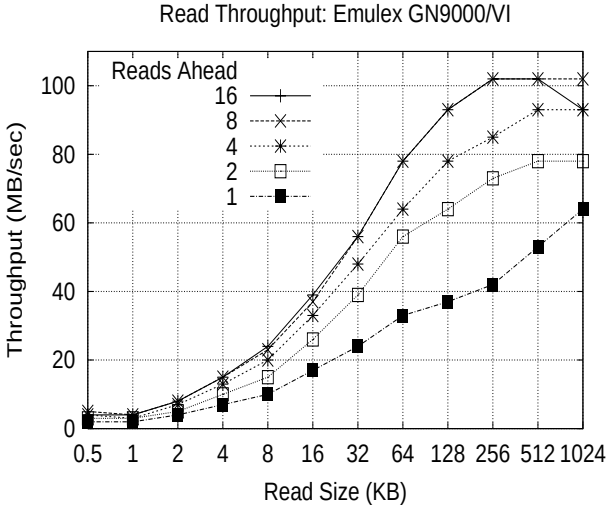

Figure 16. Emulex RDMA Throughput

We were interested in the CPU utilization during these runs. A direct measurement using the Unix `time` command of the `simple_read` process would not account for the CPU used by the asynchronous read-ahead threads. Instead, we used a tool called `statit`, described in [4] that allow aggregate CPU utilization to be measured for the duration of the `simple_read` run. In figure 17 we plotted CPU utilization against achieved throughput values using the data obtained from 4 reads-ahead for both GigaSwift and Emulex RDMA. This plot shows that reading the file over GigaSwift, even at low throughput from smaller reads, requires almost all of the CPU, while at similar throughput the Emulex RDMA requires significantly less CPU. We suspect that much of the reduction in CPU use is due to off-load of network processing into the Emulex NIC. However, there is still host CPU and mem-

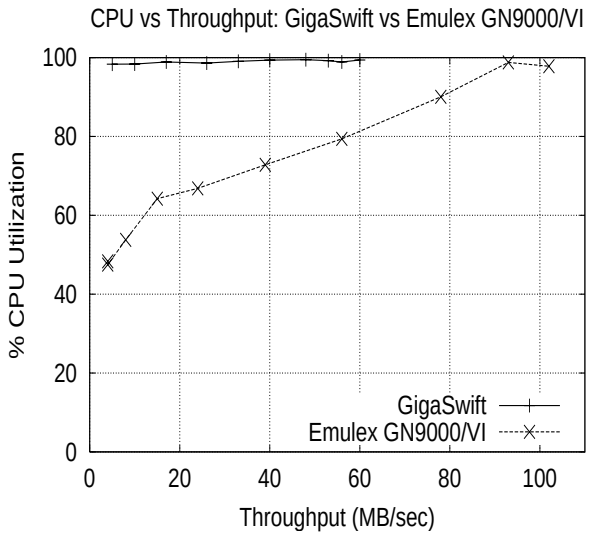

Figure 17. CPU vs Throughput

ory bus involvement in copying read data from the kernel to the user buffer, so CPU demands climb with throughput.

To confirm our comparison of CPU utilization, we measured the CPU accumulated by the `simple_read` process. To eliminate the effects of asynchronous read-ahead threads, we set `nfs3_nra` to zero – effectively turning off all read-ahead. Hence, all system CPU used to read the file would be accounted for by `simple_read`. In figure 18 we show the system CPU used by `simple_read` for a range of read sizes. These plots show the Emulex RDMA

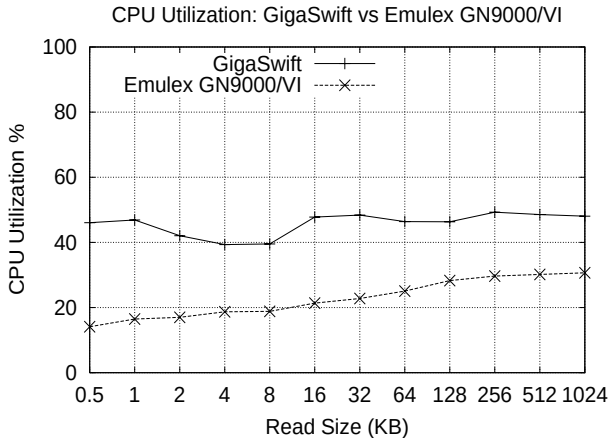

Figure 18. CPU Utilization: no read-ahead

mount using between 30% and 60% of the CPU that the GigaSwift mount uses for the same NFS transfer size.

7.4 Measurements of Write Throughput

We attempted to measure the throughput of a simple_write program to a memory filesystem on the server. We encountered some performance bottlenecks that we are currently working to resolve. Since the RDMA protocol that moves data in the NFS write direction is not significantly different from the read direction, we are confident that we will ultimately measure NFS write performance over RDMA equivalent to that of read.

8 Related Work

A team at Fujitsu Prime Software Technologies Ltd. [5] modified the Linux NFS version 2 client and server to use RDMA with the VIPL API over Emulex cLAN cards. They modified the NFS read, write and readdir code with VIPL calls to move data directly via RDMA. No changes were made in the RPC layer. They achieved NFS throughput of 90 MB/sec for read and write.

The DAFS protocol [1,6] shares many features of the NFS version 4 protocol. In addition to NFS read, write, readdir, getattr and open calls, that move data in-line, DAFS adds direct versions of these operations that pass registered buffer handles and addresses, allowing data to be moved via RDMA READ or WRITE. DAFS is a new file access protocol that uses RDMA. In contrast, NFS/RDMA preserves the existing NFS protocol, adding RDMA as a new RPC transport.

DAFS makes more explicit use of RDMA features. RDMA READ and WRITE are initiated by the DAFS server. The server returns large results to the client via an RDMA WRITE to a client buffer posted in the RPC call. This avoids the need for an RDMA_DONE message since the server can free its buffer as soon as the WRITE is complete. The NFS/RDMA protocol does not currently use RDMA WRITE at all, since it assumes the client will send the address of a correctly sized reply buffer in the RPC call message. Knowledge of the size of results is not easily accessible to the RPC layer without hints provided by a modified NFS client.

9 Future Work

Although we are pleased that we have demonstrated the utility of RDMA as an RPC transport to benefit NFS. We believe that much work yet lies ahead to fully realize the benefits.

9.1 User Level RPC

Since NFS is almost universally implemented as a kernel-level RPC service, we have focused on a kernel implementation. However, the vast majority of protocols that use ONC RPC have user-level clients and servers that access RPC through a library. Some of these RPC applications may also benefit from an RDMA transport. We would like to enhance the RPC library with an RDMA transport.

There may also be some performance benefits in providing an RDMA transport for a user-level NFS client. Such a client could access a remote server through the RPC library

by interacting directly with RDMA hardware – without going through the kernel. This would not only avoid some user-kernel context switching for I/O, but also avoid user-kernel data copies. RDMA hardware can access data directly in the application buffer. Such an NFS client could be provided as a library available to applications that need low-overhead, high-performance access to NFS data.

9.2 More Application Level Control

In making RDMA available as an RPC transport we have provided the advantages of RDMA available to a large number of RPC based applications and protocols without requiring changes to those applications and protocols. We were helped by the transport independence of ONC RPC and its Solaris implementation that made it straightforward to *plug in* RDMA as a new transport. However, we have faced some limitations in the use of RDMA features as a result of this strategy. For instance, we make no use of RDMA WRITE to return results from the server directly to a reply buffer pre-allocated by the client. The Solaris RPC API makes no provision for the client to control reply buffers at this level. We would like to extend the RPC API so that selected applications can exercise more control over data buffers when using the RDMA transport. It should be possible for the NFS client to register a large buffer for return of NFS Read data and use that buffer for multiple NFS Read calls without requiring the RPC layer to register/de-register the buffer on every call.

9.3 Copy Avoidance

Although the RDMA transport can read data in any virtual address, or perform direct placement, we do not yet fully utilize this feature in our implementation. Since the NFS client is kernel-resident, applications doing read and write via the system call interface must do a buffer copy into or out of kernel context. Although this copy can be avoided by providing a user-level NFS client library, more applications can benefit if we are able to reduce the cost of the user-kernel data copy. One possibility is to map aligned data pages between user and kernel contexts, avoiding a copy. Such techniques are described in [1] and [2].

9.4 Connection Management

An RDMA transport places some strict requirements on buffer allocation. For instance, a receive buffer must be posted for each SEND operation. If no buffer is available, or if it is too small, then a fatal error on the connection is a likely result. Our implementation uses fixed 1 KB buffers and maintains a queue of three receive buffers – replacing each buffer as it is consumed in interrupt context. However, that will not be sufficient for general support of RPC protocols and higher performance.

Currently, the NFS server is responsible for posting buffers of the correct size. Using an NFS **fsinfo** call, a client can determine this buffer size. However, the concept of short and long messages discussed in section 4.6 introduces the notion of an optimal buffer size that will accept most RPC messages – except for long ones.

One possibility is to provide calls into the RPC layer that allow the application to set/query the optimal and maximum buffers sizes for RDMA transports. But unless there is agreement between the client and server implementation on these sizes, interoperability may suffer. A better scheme may be to provide a connection management protocol that allows the client and server to have better control over the number of posted buffers and their size. This protocol may exist either as a distinct RPC protocol, or as an extension to an existing RPC protocol.

9.5 Improved RDMA Support

Our prototype uses a kernel RPC layer that uses the kVIPL driver with the Emulex GN9000/VI. We are also working with an Infiniband implementation using a Mellanox Tavor cards. We will benefit from a single API that can provide access to a multiplicity of RDMA transports including Infiniband and RDDP. Industry supported APIs like DAPL from the DAT Collaborative [17] and the Interconnect Transport API from the Open Group's Interconnect Software Consortium [18] appear to be good candidates for this single API.

9.6 Protocol Standards

If this new RDMA transport for RPC is to be inter-operable between multiple client and server implementations, then there must be a published standard that describes it. Currently the ONC RPC standards are hosted by the IETF. These standards describe UDP and TCP transports. A new IETF working group has begun work on a Remote Direct Data Placement Protocol (RDDP) [19] that will support RDMA semantics over SCTP and TCP connections. The NFS version 4 working group [20] has extended its charter to investigate the use of RDMA for NFS.

10 Conclusions

Our experience with the prototype has been encouraging. Not only does it appear that RDMA can be successfully integrated as an RPC transport layer, but that we can do so while leveraging RDMA network processing off-load and direct data placement. We expect that for many applications, ONC RPC will provide an easier path to RDMA interconnects than direct use of an API specific to RDMA.

The performance benefits and CPU savings will be of immediate interest to NFS users, particularly those with large, application servers co-located with NFS fronted storage. Computing systems will come with embedded RDMA hardware that uses Infiniband or gigabit Ethernet

based interconnects. We expect NFS/RDMA to take full advantage of the jump to 10 gigabit performance and deliver an improved level of application and system performance to existing and future applications.

11 Acknowledgments

The authors are grateful to Shantanu Mehendale, Helen Chao and Nagakiran Rajashekar for their review and comments on drafts of this paper. The authors also thank the anonymous reviewers for their helpful feedback, and finally, thanks to Prasenjit Sarkar for his help as shepherd of this paper.

References

[1] Kostas Magoutis, Salimah Addetia, Alexandra Fedorova, Margo I. Seltzer, Jeffrey S. Chase, Andrew J. Gallatin, Richard Kisley, Rajiv Wickremesinghe, Eran Gabber: Structure and Performance of the Direct Access File System. *USENIX Annual Technical Conference*, General Track 2002: 1-14

[2] H. J. Chu. Zero-Copy TCP in Solaris. In *Proc. of USENIX Technical Conference*, San Diego, CA, January 1996.

[3] LSO, Large Send Offload - also known as TCP Segmentation Offload. Described in: `http://www.microsoft.com/whdc/hwdev/tech/network/taskoffload.mspx`

[4] Jim Mauro, Richard McDougall. The statit program. In *Solaris Internals: Core Kernel Architecture*, Prentice Hall PTR.

[5] Fujitsu Prime Software Technologies Ltd, Technical Report. *An Adaptation of VIA to NFS on Linux*, `http://pst.fujitsu.com/english/nfs`

[6] DAFS Collaborative. *Direct Access File System Protocol, Version 1.0*, September 2001. `http://www.dafscollaborative.org`.

[7] Remote Shared Memory API (RSMAPI) is a shared memory AP for Solaris Clusters that uses Dolphin SCI (Scalable Coherent Interface) to provide low latency access to remote memory. Further details on the API may be found in: `http://docs.sun.com`

[8] Intel Kernel Virtual Interface Provider Library (KVIPL) Addendum, March 25 1999. `http://www.intel.com/design/servers/vi/developer/ia_imp_guide.htm`

[9] The Connectathon test suite. A suite of C programs used to test NFS implementations at Connectathon events and downloadable from: `http://www.connectathon.org`

[10] B. Pawlowski, C. Juszczak, P. Staubach, C. Smith, D. Lebel, and D. Hitz. NFS Version 3 Design and Implementation. In *Proc. of USENIX TechnicalConference*, Boston, MA, June 1994.

[11] S. Shepler, B. Callaghan, D. Robinson, R. Thurlow, C. Beame, M. Eisler, and D. Noveck. "NFS Version 4 Protocol," RFC 3530, April 2003. `http://www.ietf.org/rfc/rfc3530.txt`

[12] Srinivasan, R. "RPC: Remote Procedure Call Protocol Specification Version 2," RFC 1831, August 1995. `http://www.ietf.org/rfc/rfc1831.txt`

[13] Srinivasan, R. "XDR: External Data Representation Standard," RFC 1832, August 1995. `http://www.ietf.org/rfc/rfc1832.txt`

[14] M. Eisler, A. Chiu, L. Ling, "RPCSEC_GSS Protocol Specification," RFC 2203, September 1997. `http://www.ietf.org/rfc/rfc2203.txt`

[15] S. Kent, R. Atkinson, "Security Architecture for the Internet Protocol," RFC 2401, November 1998. `http://www.ietf.org/rfc/rfc2401.txt`

[16] J. Linn, "Generic Security Service Application Program Interface Version 2, Update 1," RFC 2743, January 2000. `http://www.ietf.org/rfc/rfc2743.txt`

[17] DAT Collaborative: Direct Access Transport. An industry group set up to define transport-independent, platform-independent APIs for RDMA: `http://www.datcollaborative.org`

[18] The Interconnect Software Consortium (ICSC). An industry consortium hosted by The Open Group set up to promote APIs for fast interconnects such as Infiniband: `http://www.opengroup.org/icsc`

[19] Remote Direct Data Placement Protocol (RDDP). IETF working group charter: `http://www.ietf.org/html.charters/rddp-charter.html`

[20] NFS version 4 Protocol. IETF working group charter: `http://www.ietf.org/html.charters/nfsv4-charter.html`