NetApp™

Go further, faster™

NFS/RDMA Update and Future

Tom Talpey
tmt@netapp.com

NFS Connectathon
May 2008

NFS-RDMA is…

- Good-old-NFS, over RDMA. It's faster.
- Allows any NFS version to use any RDMA network
- A transparent solution (binary compatible and admin-friendly) for applications, NFS protocol features, and NFS users
- A significant performance boost to clients
  - Reduces client CPU overhead
  - Utilizes high-bandwidth, low-latency fabrics
- A single-wire host cluster solution

Why NFS/RDMA?

- *Reduced client-side overhead*
  - Achieved primarily through elimination of data copies
  - Secondarily, through efficient offloaded RDMA I/O model
- *Access to bandwidth*
  - CPU is no longer a limiting factor
  - Full benefit of low fabric latency
- Result: NFS performance similar to local filesystem

Roughly…

- Full wire bandwidth at minimal client CPU
  - Helen Chen OFA07 presentation
    - http://www.openfabrics.org/archives/spring2007sonoma/Tuesday May 1/Helen Chen NFS over RDMA – IB and iWARP-5.pdf
  - Alexandros Bataskis FAST08 paper
    - http://www.usenix.org/events/fast08/tech/batsakis.html
- Easily scales with interconnect speed
  - Provided the backend storage is also scaled

What it *doesn't* provide

- Does not, by itself, increase bandwidth
  - *"I hooked up my old machine to this firehose, why doesn't it spray like a firehose?"*
  - ► Many NFS limits are not due to the wire
  - Server limits are usually out of disk, or out of filesystem ops
  - Client limits typically stem from parallelism, and buffered read/write policies
- Does not increase server performance
  - Unless the server is out of CPU (a different problem – get a bigger server to match)

What it *does* provide

- The benefits are, nonetheless, evident
  - Low client CPU
    - Single-digit cpu% at full wire bandwidth
    - NFS at full wire bandwidth!!
  - High end-to-end throughput
    - Full wire bandwidth
    - Rock-bottom metadata latency

Standardization

NFS-RDMA is standardized in the IETF NFSv4 Working Group:

http://www.ietf.org/html.charters/nfsv4-charter.html

In two layered specifications:

- RDMA Transport for ONC RPC
  - Describes the RPC-RDMA protocol for sending RPC messages on an RDMA transport (IB, iWARP)
- NFS Direct Data Placement
  - Describes the NFSv2/v3/v4 mapping to RPC-RDMA operations

Standardization

- Documents are clearing IETF Last Call
  - Soon to be approved for RFC standardization
- Port number
  - Unofficially using port 2050
    - Additional port required to support NFSv3/iWARP
  - Official assignment soon

Open Source Implementations

- Linux
  - http://nfs-rdma.sourceforge.net/
  - Infiniband and iWARP
  - Client – in 2.6.24-
  - Server – in 2.6.25-
  - Also as server product from SGI
- OpenSolaris
  - Infiniband
  - Client and server
  - In upcoming release
- Yes, they interoperate

Enabling server side Linux

- When RDMA adapters configured and present:
  - modprobe svcrdma
  - echo rdma nnnn > /proc/fs/nfsd/portlist
  - In NFSD startup scripts or manually
- Server listens on all RDMA adapters

Enabling client side Linux

- mount server:/fs /mnt [-i] –o rdma,port=nnnn
  - Client chooses RDMA adapter automatically
    - rdma_connect(server)
- Well, almost that simple
  - May have to "modprobe xprtrdma"
  - Linux mount API
  - Need nfs_utils v1.1 or better
    - mount.nfs server:/fs /mnt –o rdma,port=nnnn
- Port requirement will no longer be needed
  - when IANA port assignment is received

Connectathon testing status

- Implementations:
  - All testing on Infiniband 4X SDR (10Gbps)
  - NetApp: Linux 2.6.26-rc1+
    - Client and server on Infinihost III
  - RedHat: Linux Fedora 9 (2.6.25-14.fc9)
    - Client and server on ConnectX
  - Sun: OpenSolaris 11
    - Client and server on Infinihost III
  - SGI: SGI ISSP 1.2
    - Client and server on Infinihost III
- No iWARP testing yet

Week's results so far

- Linux testing (NetApp and RedHat)
  - Successful basic and general tests
  - Server issue in special locking test 7
    - Server assertion or server fault, being tracked
  - No client issues identified
- Some Infiniband interoperability issues:
  - Failure to connect from Infinihost III->ConnectX
    - Works fine when roles reversed
  - Switch connectivity issues

Week's results so far - 2

- Protocol implementation quirks
  - XDR padding for reads of length %4 != 0
  - Detected by special/holey
    - Client reads 4096 bytes of 4321 byte file @4096
    - Server returns 225 bytes via RDMA
    - Pad bytes not sent, which client must account for
      - Client spec violation!
- Credit overflow / lost connections
  - Possibly client or server implementation
  - Just as possibly, fabric issues
  - Only under load (basic/test5 and special/test7)
  - Infiniband RNR retry? Firmware? Server timing?

Week's results so far - 3

- Linux client/server on Fedora 9 is able to run with no additional package installed (yay)
- 5 steps to NFS/RDMA:
  - (Boot Fedora 9)
  - modprobe ipoib
  - ifconfig ib0 <addr> up
  - Client:
    - modprobe xprtrdma
    - mount –o rdma …
  - Server
    - modprobe svcrdma
    - echo "rdma nnnn" >/proc/fs/nfsd/portlist

Week's results so far - 4

- SGI testing with Linux fully successful (!)
  - NetApp/Linux client to SGI server passed first time (yay)
  - SGI client to NetApp/Linux server passes at 32KB and 64KB (double yay)
  - Fails at 128KB, as expected due to limited scatter/gather of server
- Hope to complete more testing w/Solaris today
  - NetApp/Linux – followup testing
  - SGI – first testing
- Other vendor reports?

NetApp™

Go further, faster™

NFS/RDMA futures

(mainly Linux and Linux/RDMA)

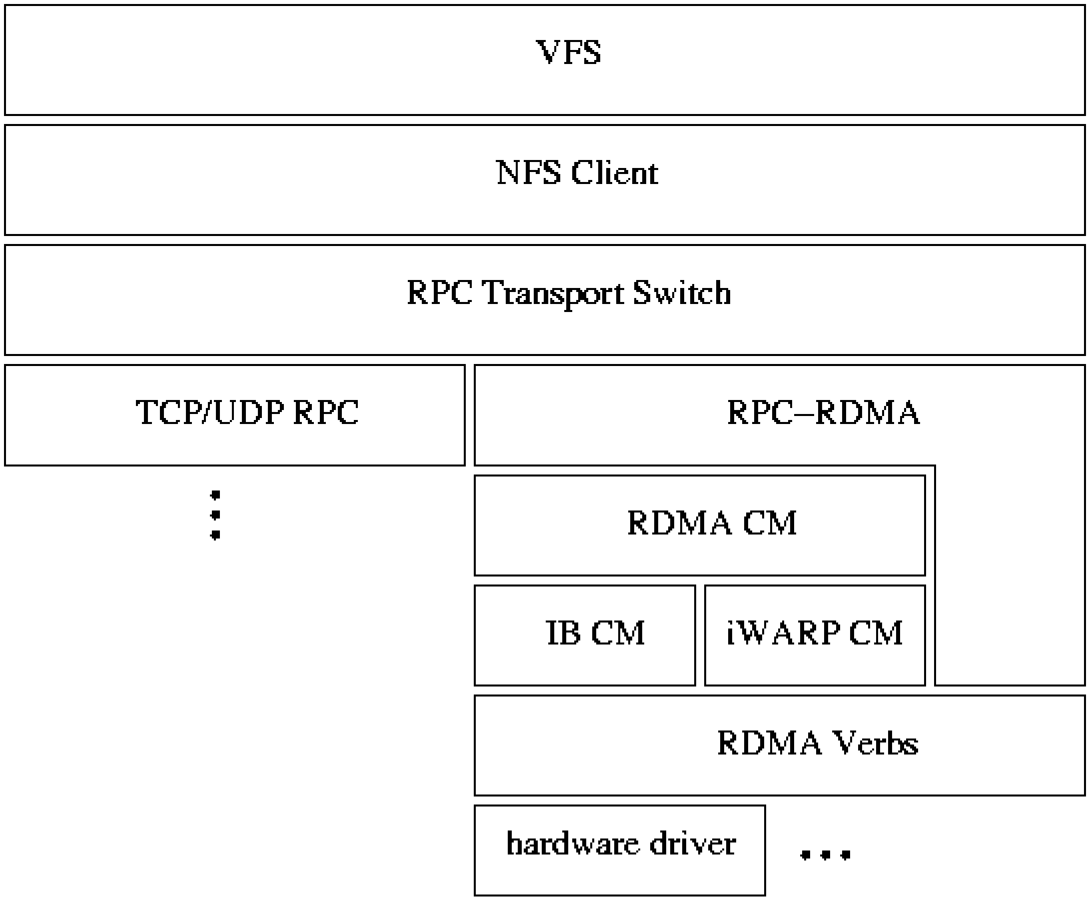
An aside…

- Storage over RDMA is an inflection point
- Used to be…
  – The **storage** was faster than the **wire**
- With NFS/RDMA…
  – The **wire** is faster than the **storage**
- "Interesting" things have appeared
  – Cache algorithms
  – "dd if=/dev/zero of=/mnt/nfs-rdma" (cached writes) produces surprising results
  – Many Linux FS and VM changes have resulted, more are to come

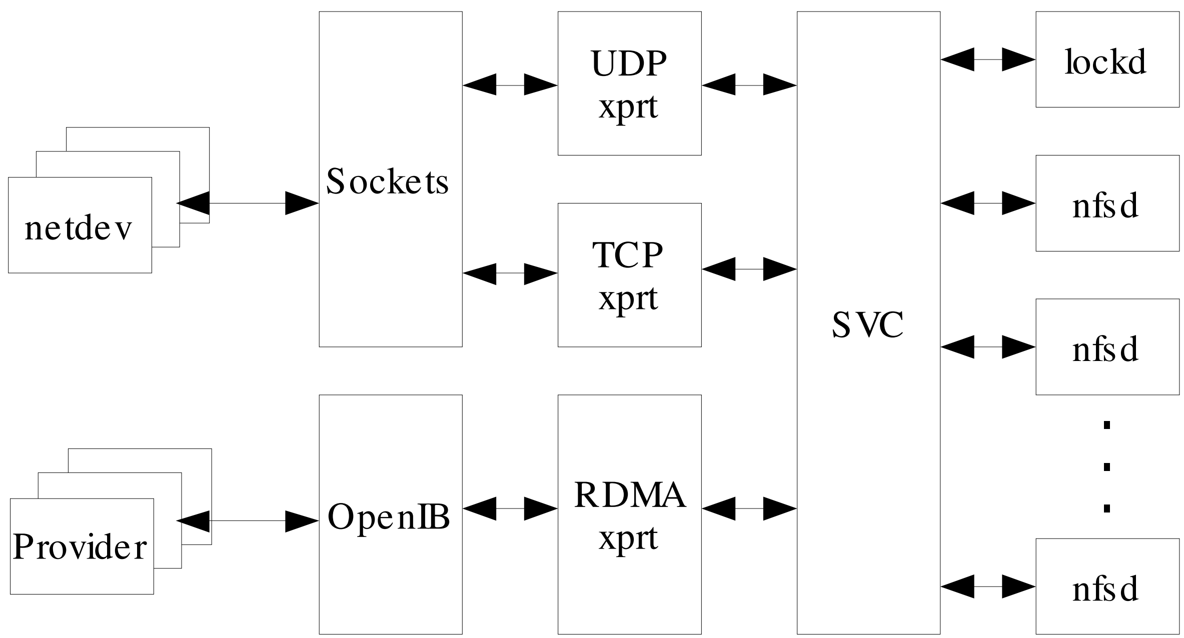
Client Direct I/O

- Direct I/O through NFS layer is fully supported
- User pages are pinned / mapped by the VFS
- Passed down to RPC as pagelist
- Pagelist is simply mapped to the adapter
- Zero-copy, zero-touch (send or receive)

Client Architecture



Server Architecture



RPC Server

- All RDMA operations (READ/WRITE) initiated by the server
- NFS Server is unaware of transport, processes only RPC
- Multiple concurrent RPC may be processed
- Credit based flow-control (advertised in RDMA header) used to back-pressure client
- Server registers all of physical memory to avoid registration overhead
  – Safe on Infiniband, slight risk on iWARP RDMA Read

Wire parallelism

- NFS highly parallel on-the-wire
  - Linux RPC slot table typically 16
  - NFS/RDMA defaults to 32, can go to 128+
- Cached I/O uses kernel helpers
  - Readahead
  - Writebehind
- Application can issue in parallel
  - Async I/O, multiple threads, etc
  - iozone –H# (aio) or –t# (threads)
- AIO+DIO (asyncio/directio) rocks!
  - Iozone –I –H# …

I/O size

- NFS default (historical) is 32KB
- Recent Linux kernels support up to 2MB
- Only some servers support this
  - Many stop at 64KB due to atomicity
- Large I/Os typically benefit the *server*
  - Some benefit from larger contiguous writes
    - Better sequencing, larger stripes (~128KB)
  - Fewer ops to the server FS

I/O size - client

- Current NFS/RDMA client limited to 32KB
  - Bookkeeping limits – accounting for memregs
  - Reduces RDMA chunks on the wire
  - Small size is not a major impact, due to high wire parallelism
- On plan to increase to ~128KB
  - To enable gains at the server
  - Decreasing gains above this size

Resource usage

- Client – memory segment accounting
  - Currently a fixed list
  - Want a dynamic list
  - To be addressed in conjunction with memreg
    - Because memreg strategy greatly impacts it
- Server – scatter/gather handling
  - Number of RDMA Read/Write ops needed
  - Depends on client memreg, and server memreg
  - Also ability of adapter to coalesce
- Server – receive memory usage
  - Shared receive queue (tbd)

Client Memory Management

- Multiple memory management algorithms are possible. Five have been implemented:
  - Pure Inline (debug, “TOE-like”)
  - Memory Region
  - Synchronous Memory Window
  - Asynchronous Memory Window
  - Fast Memory Region (FMR)
  - Persistent (“experimental”)
- But, which to choose?

Client memreg strategies

	Mode	IB	IWarp	Safe?	Fast?	SGE segs/op	Stalls?	Irpts?	Addressing, protection
No memreg (copy to bounce buffers)	0	Yes	Yes	Yes	No	Any, small (1KB - 4KB)	No	No	N/A
ib_reg_phys_mr	1	Yes	Yes	Yes	No	>= 4	No	No	Virt, byte
ib_bind_mw	2,3	No	No	Yes	Semi	1	Yes	Yes	Virt, byte
ib_map_phys_fmr	4	Some	No	Semi (NO if pools)	Semi	>=32	?	No	Phys, page
ib_get_dma_mr	5	Yes	Most (some< 2GB)	NO!	Yes	1	No	No	Phys, none

Key: RED=bad ORANGE=issue

But all I want is an R_Key!

- So why does client code have to decide what to do?
- Currently, the client chooses mode 5
  - All-physical
  - Because it's the only fast mode that (almost) always works
- The server code takes a similar approach
  - But is working on a better iWARP method

The ideal RDMA memory registration

- Has one API that works on all adapters
  - Upper layers shouldn't have to care
  - Users and admins really shouldn't have to care
- Protects byte-ranges
- Scatters/gathers many (32+) segments
- Completes asynchronously
- Doesn't stall the queue (optionally)

Other memreg features

- E.g.
  – Send w/invalidate
- Not interested unless they're:
  – Really good
  – Widely useful/supported
- Not worth it to write adapter- or transport-specific code
  – And maintain it
  – And tell users how best to use it
- New memreg strategy(ies) under development
  – Watch OFA space

Misc issues

- Lots of RDMA Read responder resources
  – Needed at client only – server is requestor
- rdma_cm (Connection Manager)
  – Responder resources mismatch
  – IPv6
  – Source port selection

Simplicity

- Supporting all features is good
- But it needs to be usable - without having to consult *an encyclopedia of adapters*, fabrics, etc. to decide which to use
- And without writing even more code to support new schemes

Other bottlenecks

- Linux Server:
  - Memory registration
  - Large I/O
  - Thread service model
  - VFS interface (synchronous, data copies)
- Linux Client:
  - Memory registration
  - Large I/O (to help the server)
  - Buffer cache VM write behavior

NFS/RDMA is a success if…

- Users can use it without knowing how RDMA, and each different RDMA adapter, work
- NFS actually works (well) over it
- Without “too much” effort
- Without breaking (corrupting data, stopping jobs)
- If users’ issues are reduced to NFS issues
- So far, so good! ☺

Resources

- Tom Talpey (NFS/RDMA client maintainer):
  - tmt@netapp.com
- Tom Tucker (NFS/RDMA server maintainer):
  - tom@opengridcomputing.com
- NFS-RDMA project:
  - http://sourceforge.net/projects/nfs (howto)
  - nfs-rdma-devel@lists.sourceforge.net
- Linux-NFS:
  - linux-nfs@vger.kernel.org

NetApp™

Go further, faster™

Backup

But all I want is an R_Key!

- So why do I have to write client code like these 4 slides?

```
int
rpcrdma_register_external(struct rpcrdma_mr_seg *seg,
                                        int nsegs, int writing, struct rpcrdma_xprt *r_xprt)
{
        ...
        switch (ia->ri_memreg_strategy) {

        case RPCRDMA_ALLPHYSICAL:
                rpcrdma_map_one(ia, seg, writing);
                seg->mr_rkey = ia->ri_bind_mem->rkey;
                seg->mr_base = seg->mr_dma;
                seg->mr_nsegs = 1;
                nsegs = 1;
                break;
```

MTHCA FMR’s

```
/* Registration using fast memory registration */
case RPCRDMA_MTHCAFMR:
        {
        u64 physaddrs[RPCRDMA_MAX_DATA_SEGS];
        int len, pageoff = offset_in_page(seg->mr_offset);
        seg1->mr_offset -= pageoff;          /* start of page */
        seg1->mr_len += pageoff;
        len = -pageoff;
        if (nsegs > RPCRDMA_MAX_DATA_SEGS)
                nsegs = RPCRDMA_MAX_DATA_SEGS;
        for (i = 0; i < nsegs;) {
                rpcrdma_map_one(ia, seg, writing);
                physaddrs[i] = seg->mr_dma;
                len += seg->mr_len;
                ++seg;
                ++i;
                /* Check for holes */
                if ((i < nsegs && offset_in_page(seg->mr_offset)) ||
                    offset_in_page((seg-1)->mr_offset+(seg-1)->mr_len))
                        break;
        }
        nsegs = i;
        rc = ib_map_phys_fmr(seg1->mr_chunk.rl_mw->r.fmr,
                                physaddrs, nsegs, seg1->mr_dma);
        ...
        seg1->mr_rkey = seg1->mr_chunk.rl_mw->r.fmr->rkey;
        seg1->mr_base = seg1->mr_dma + pageoff;
        seg1->mr_nsegs = nsegs;
        seg1->mr_len = len;
        }
        break;
```

Memory Windows

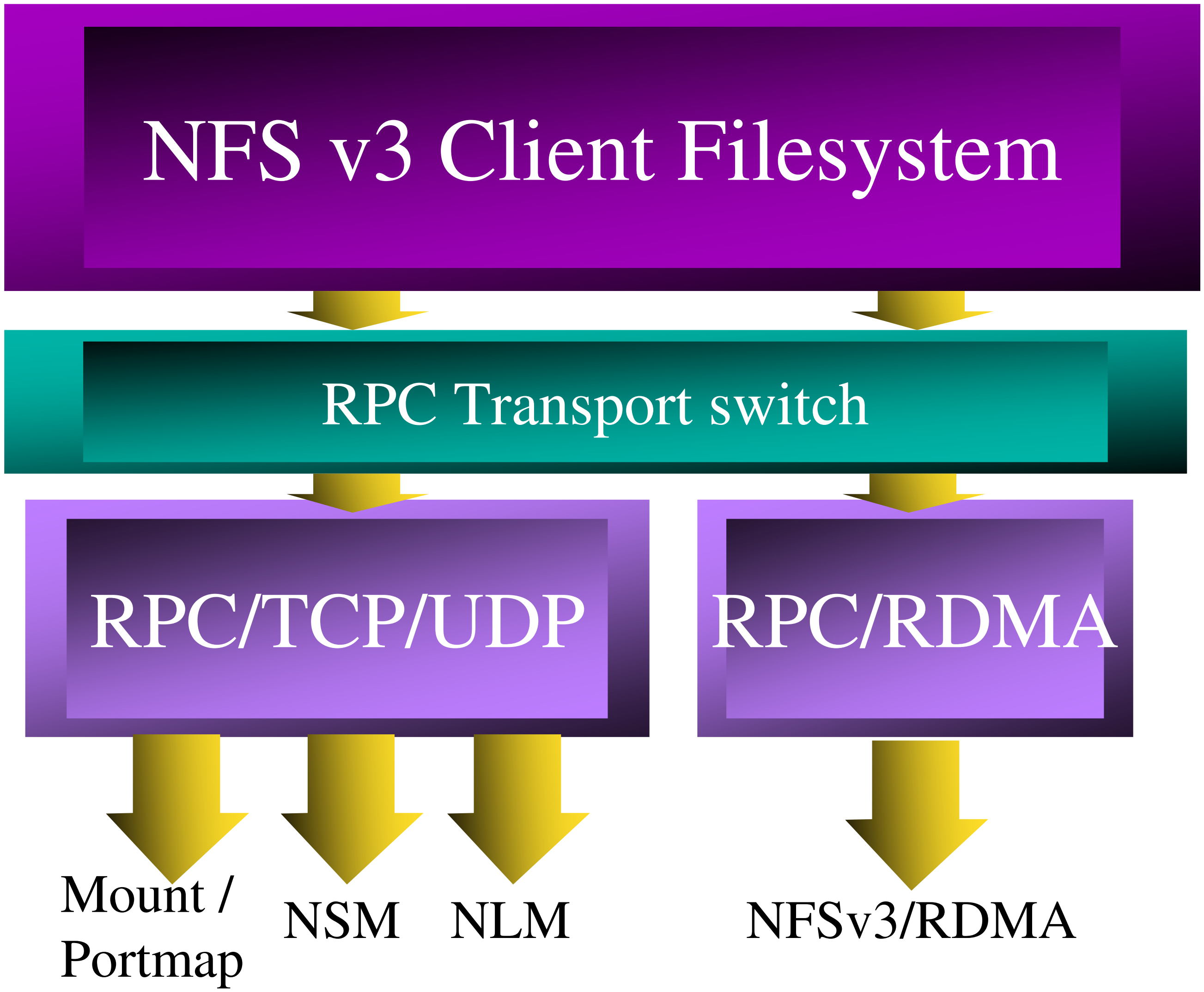
```
/* Registration using memory windows */
case RPCRDMA_MEMWINDOWS_ASYNC:
case RPCRDMA_MEMWINDOWS:
        {
        struct ib_mw_bind param;
        rpcrdma_map_one(ia, seg, writing);
        param.mr = ia->ri_bind_mem;
        param.wr_id = 0ULL;    /* no send cookie */
        param.addr = seg->mr_dma;
        param.length = seg->mr_len;
        param.send_flags = 0;
        param.mw_access_flags = mem_priv;

        rc = ib_bind_mw(ia->ri_id->qp, seg->mr_chunk.rl_mw->r.mw, &param);
        ...
        seg->mr_rkey = seg->mr_chunk.rl_mw->r.mw->rkey;
        seg->mr_base = param.addr;
        seg->mr_nsegs = 1;
        nsegs = 1;
        }
        break;
```

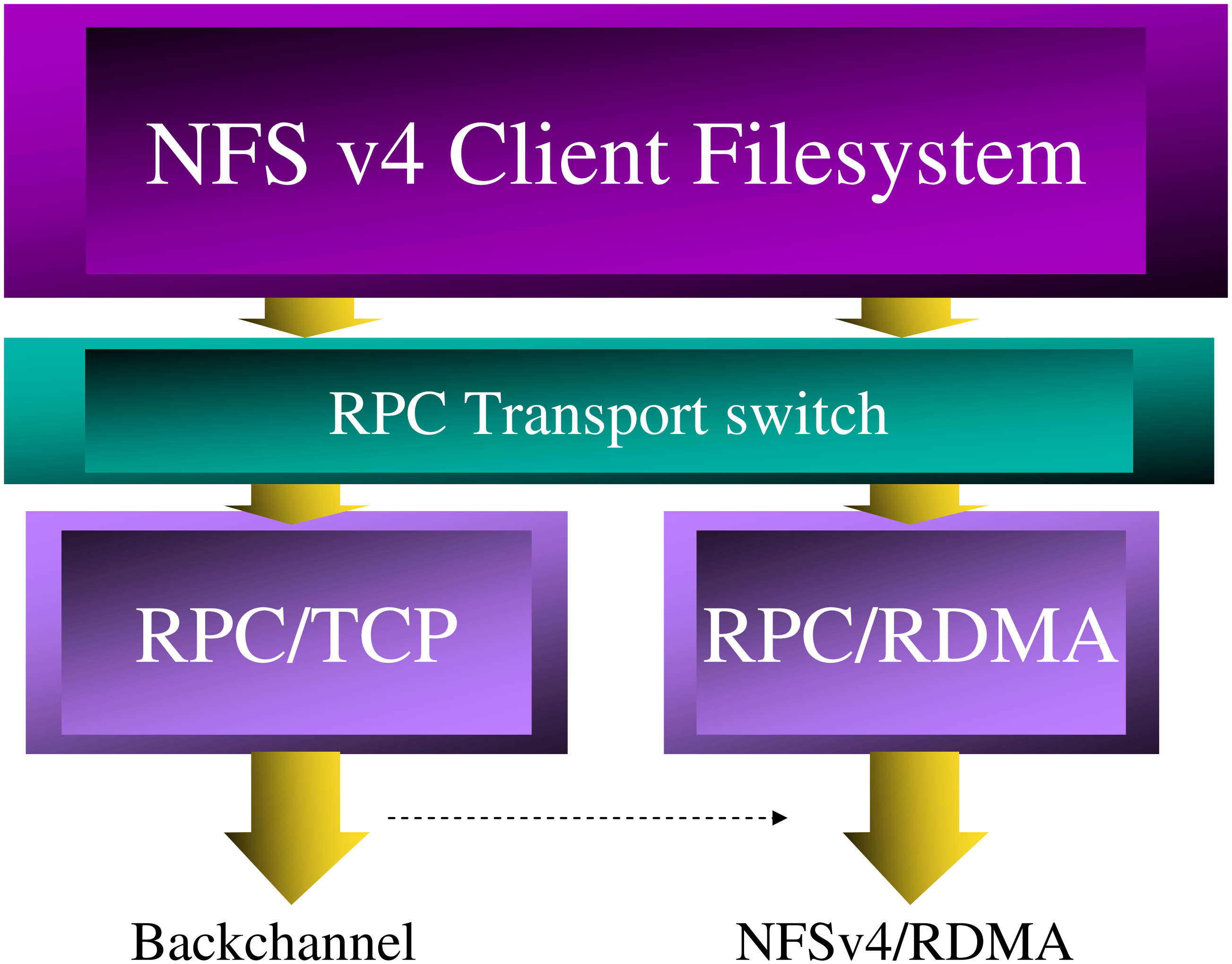
Hardway

```
/* Default registration each time */
default:
            {
            struct ib_phys_buf ipb[RPCRDMA_MAX_DATA_SEGS];
            int len = 0;
            if (nsegs > RPCRDMA_MAX_DATA_SEGS)
                        nsegs = RPCRDMA_MAX_DATA_SEGS;
            for (i = 0; i < nsegs;) {
                        rpcrdma_map_one(ia, seg, writing);
                        ipb[i].addr = seg->mr_dma;
                        ipb[i].size = seg->mr_len;
                        len += seg->mr_len;
                        ++seg;
                        ++i;
                        /* Check for holes */
                        if ((i < nsegs && offset_in_page(seg->mr_offset)) ||
                          offset_in_page((seg-1)->mr_offset+(seg-1)->mr_len))
                                    break;
            }
            nsegs = i;
            seg1->mr_base = seg1->mr_dma;
            seg1->mr_chunk.rl_mr = ib_reg_phys_mr(ia->ri_pd,
                                                ipb, nsegs, mem_priv, &seg1->mr_base);
            ...
            seg1->mr_rkey = seg1->mr_chunk.rl_mr->rkey;
            seg1->mr_nsegs = nsegs;
            seg1->mr_len = len;
            }
            break;
}
```

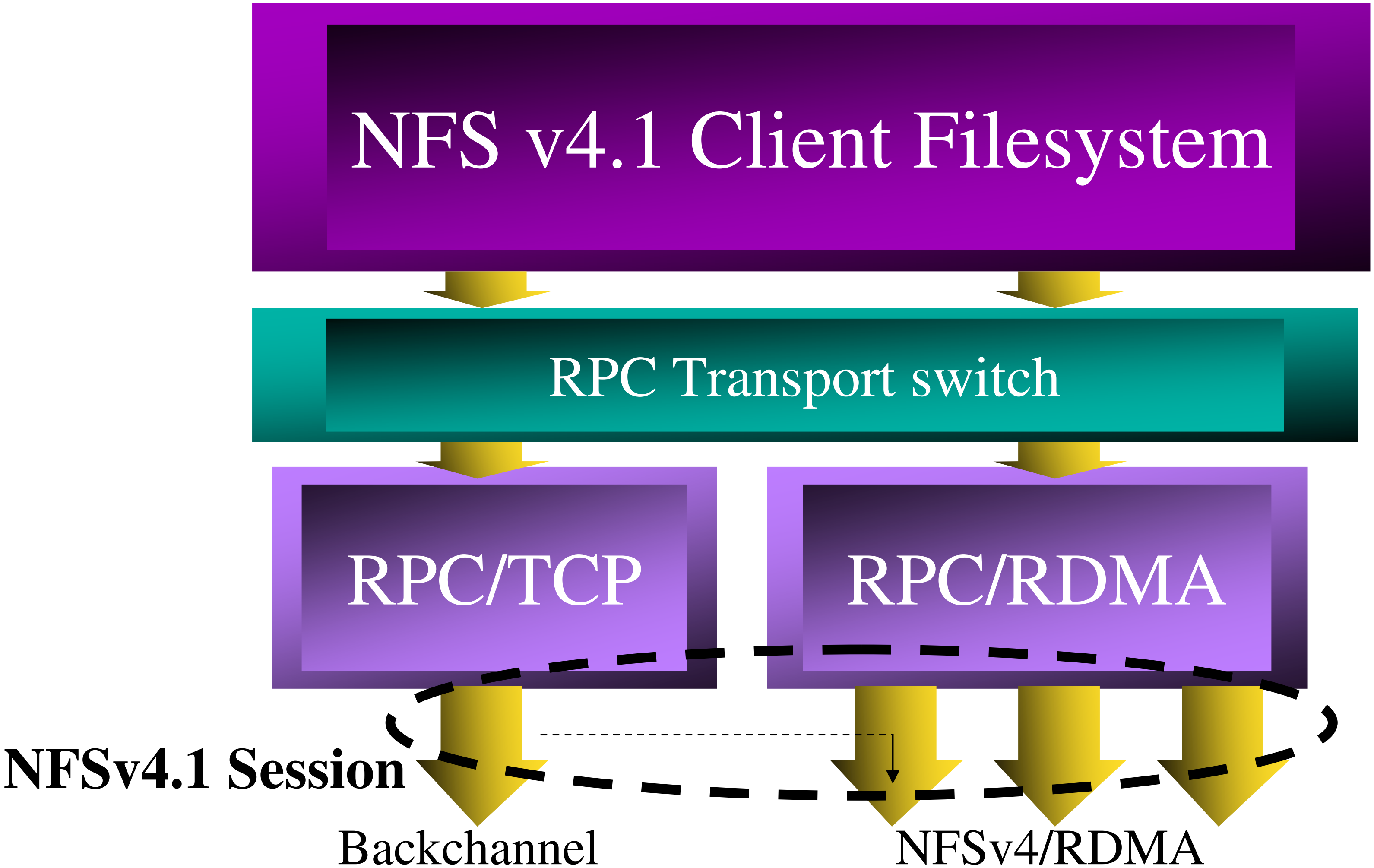
NFSv3/RDMA stack (RDMA/TCP/UDP)



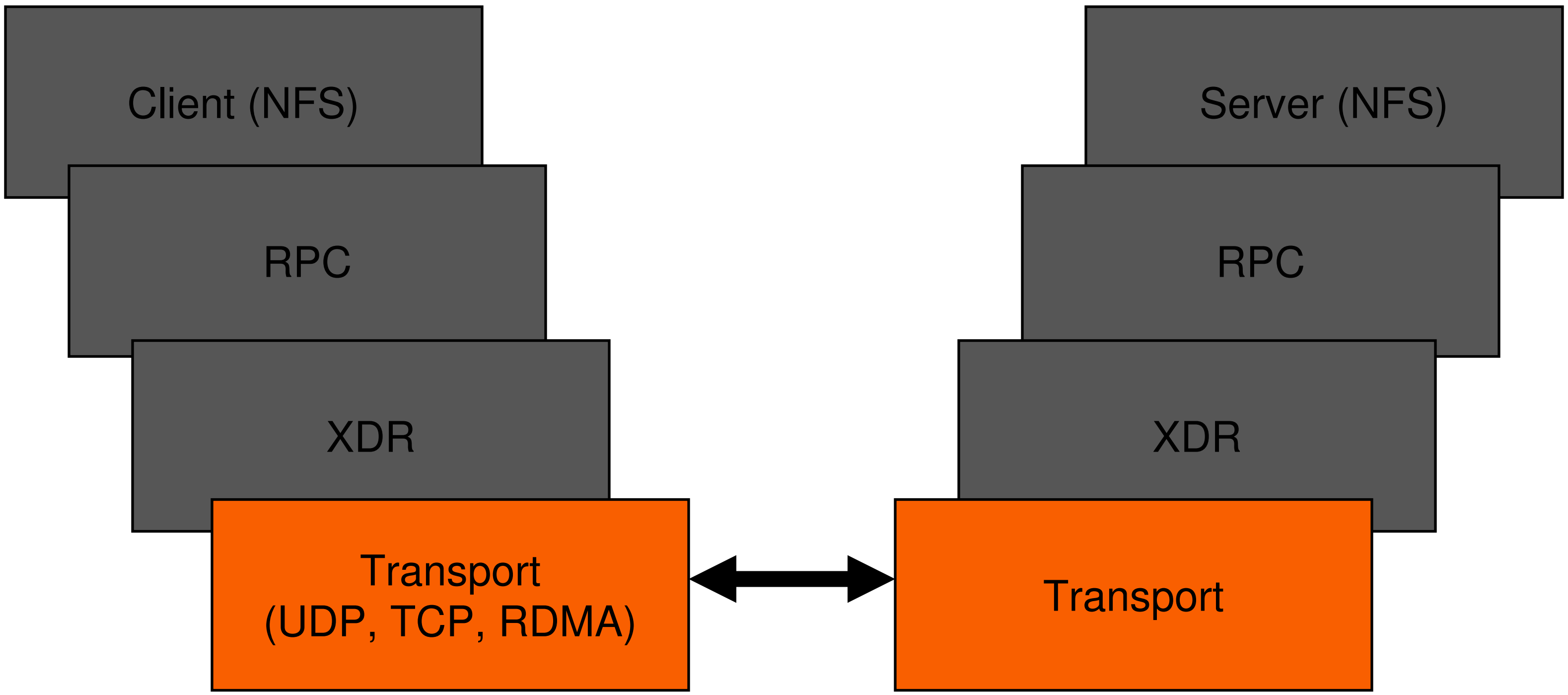
NFSv4/RDMA stack (RDMA/TCP)



NFSv4.1(pNFS)/RDMA stack (all RDMA)



RPC layering model

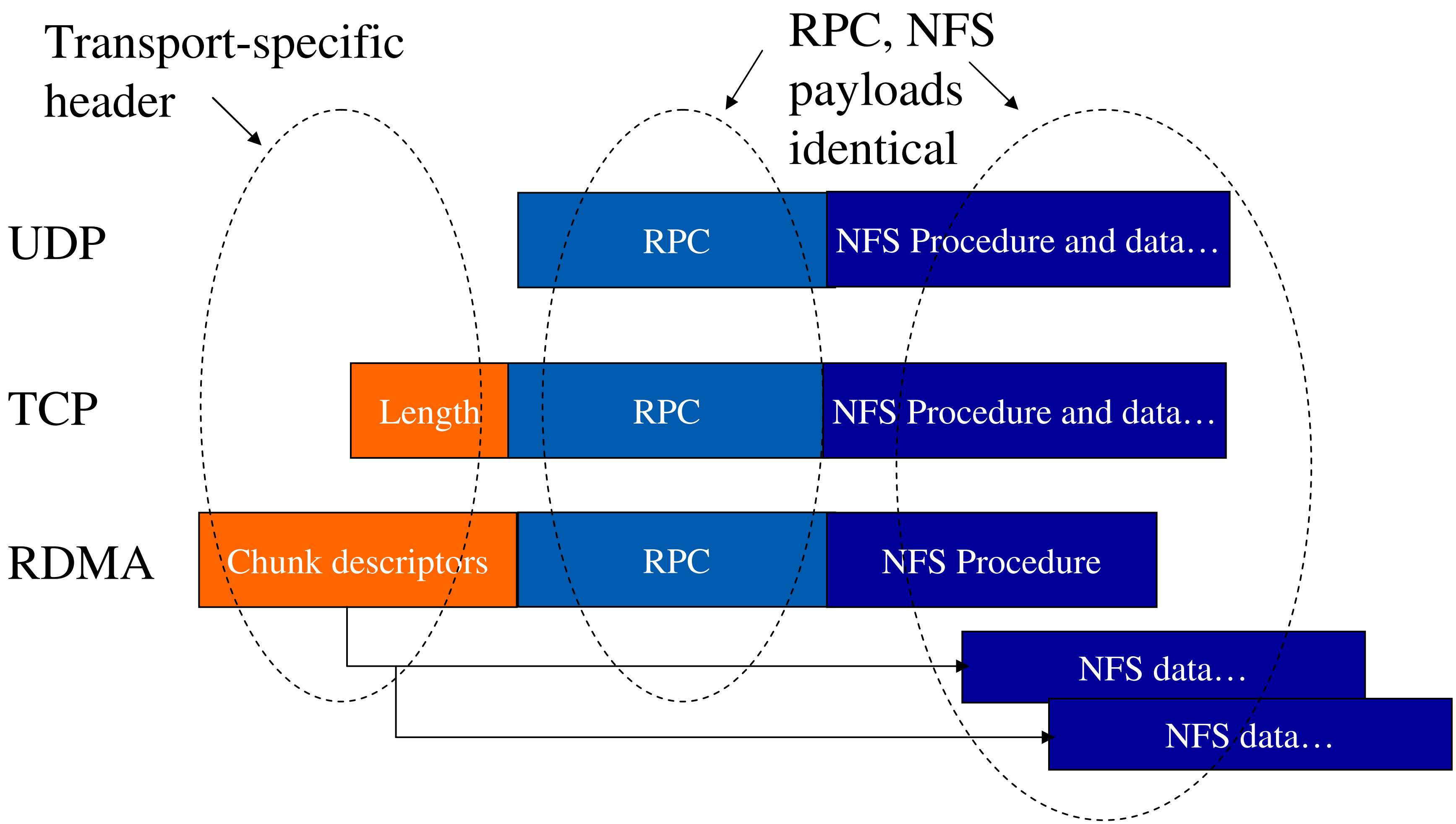


- RPC/RDMA only changes transport
- No upper layer changes required
- Some RPC implementation changes may be desirable

RPC/RDMA as RPC Transport

- RDMA, as a transport, has unique properties
  - Reliable (like TCP)
  - Atomic (preserves boundaries) (like UDP)
  - Messages are sequenced and ordered
  - Supports direct transfer (RDMA)
    - Using handle/length/offset “triplets”
- Naturally leads to defining a new transport type

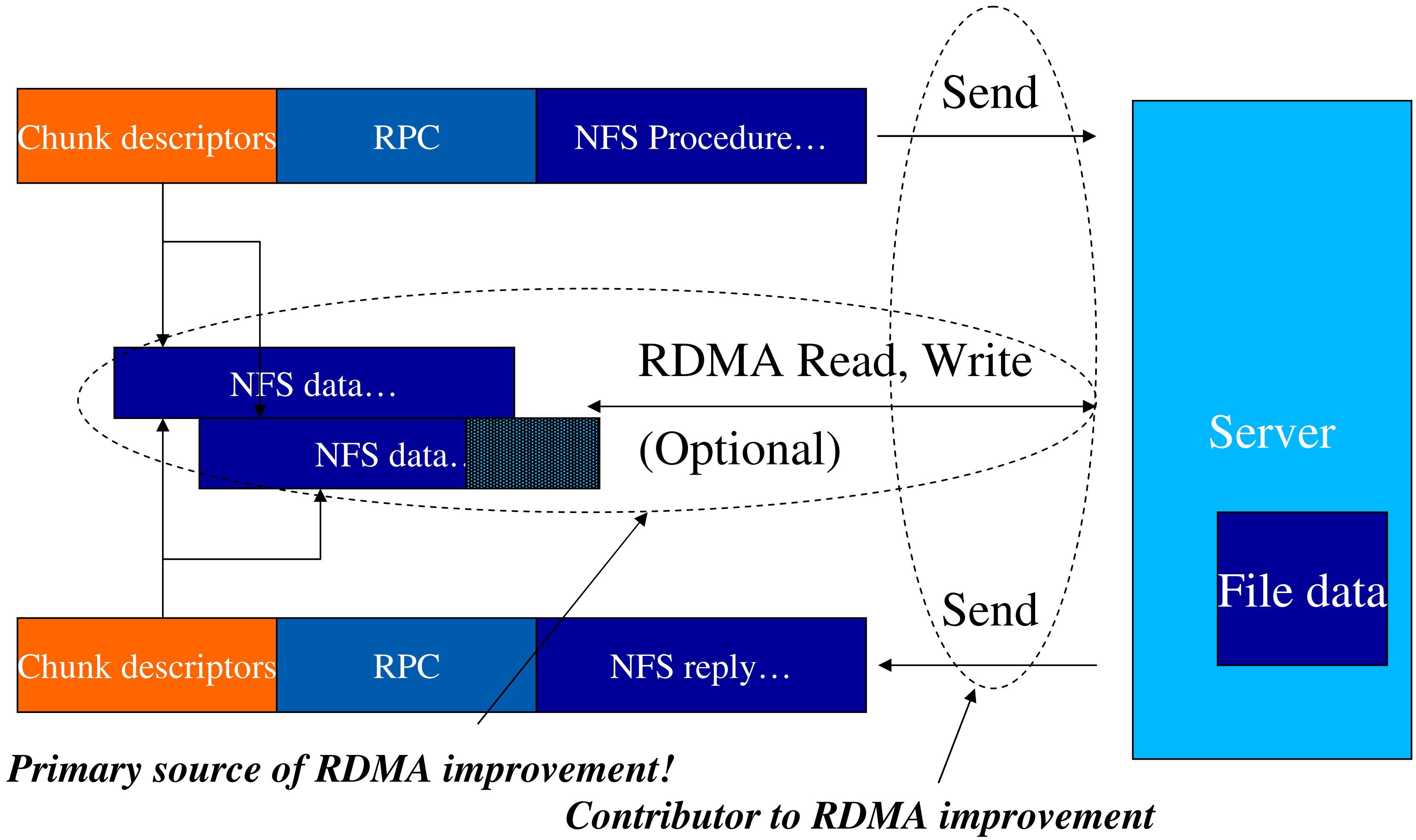
Transport RPC format



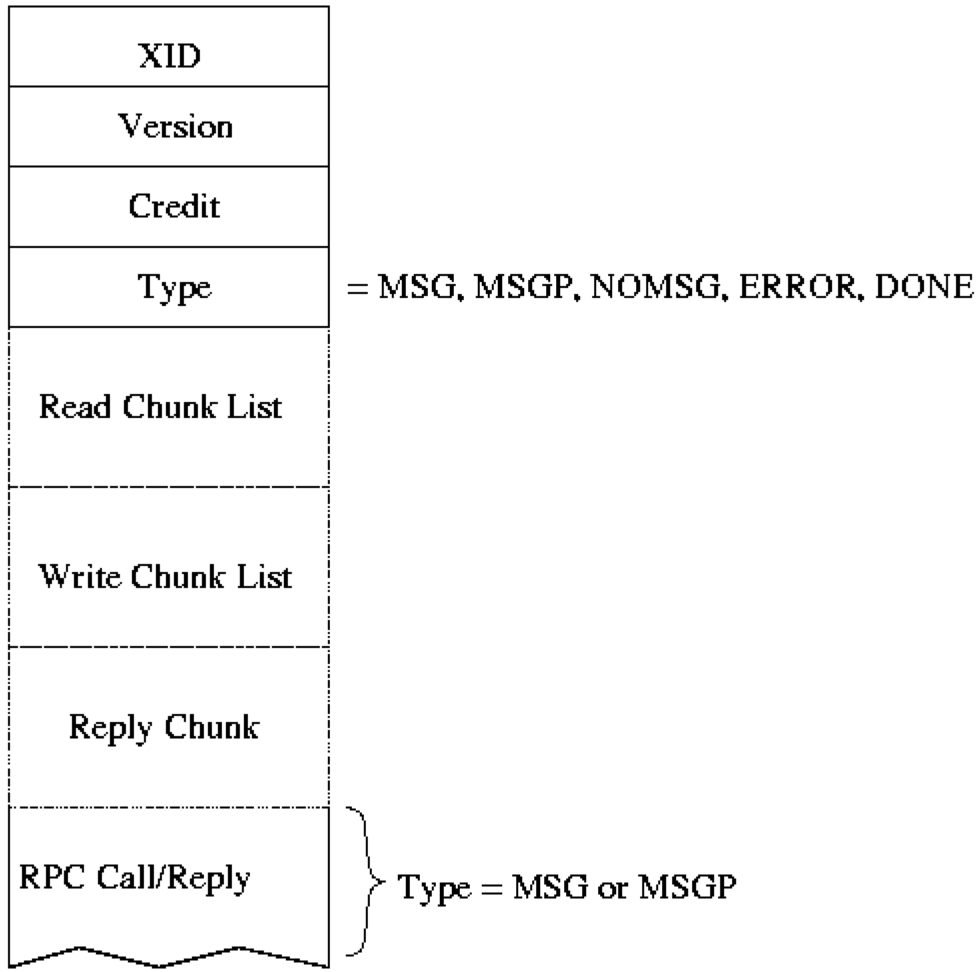
RPC-RDMA Operations

- RPC-RDMA can transfer RPC packets using
  - RDMA Send
  - RDMA Read/Write
- NFS-RDMA restricts RDMA Read/Write usage
  - only the server can use RDMA Read/Write

RPC/RDMA Transfer

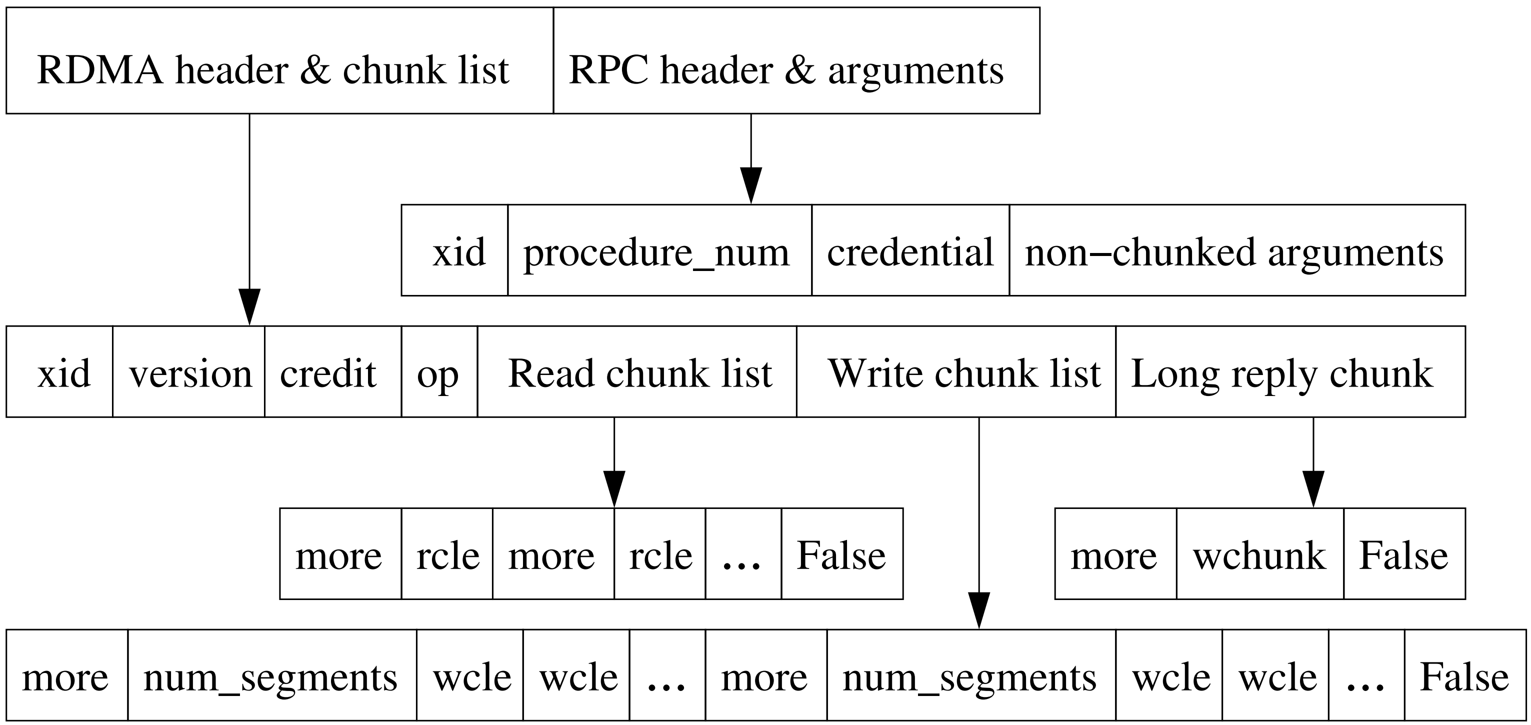


RPC-RDMA transport header (simplified)



RPC/RDMA transport header (detail)

RPC Call Format



more: A boolean variable that indicates whether there is a chunk list element following

rcle: Read chunk list element

wcle: Write chunk list element

wchunk: Write chunk for long reply

Server Control

- All RDMA is initiated by the *Server*
  - Improves correctness
    - Server does not expose its memory to clients
  - Improves performance
    - Server optimizes or avoids dynamic registration
    - Server paces data flow as it can accept it
- Server controls client Send credits
  - Client requests desired number
  - Server reallocates (grants) at each exchange
  - Optimizes server resources
    - Good for clients

Server XDR argument chunk decode

- Each chunk is a “pointer” to data not present in the message, but logically present in the stream
- When decode reaches an XDR position referenced by a chunk, the data source switches from the message to the (possibly remote) chunk
- RDMA transfer is used to process each such phase of the decode

Server XDR result chunk encode

- When RDMA-eligible data (dictated by the upper layer binding e.g. NFS/RDMA) is reached, the next eligible chunk is used
  - If no chunk, the data is sent inline
  - Otherwise, RDMA transfer moves the data, and the chunk is returned with its length and position for client decode
    - Each chunk results on one RDMA op on the wire
- Net result at the client RPC is a fully-decoded and present RPC message