# RDS

## Overview

This readme tries to provide some background on the hows and whys of RDS, and will hopefully help you find your way around the code.

In addition, please see this email about RDS origins: http://oss.oracle.com/pipermail/rds-devel/2007-November/000228.html

## RDS Architecture

RDS provides reliable, ordered datagram delivery by using a single reliable connection between any two nodes in the cluster. This allows applications to use a single socket to talk to any other process in the cluster - so in a cluster with N processes you need N sockets, in contrast to N*N if you use a connection-oriented socket transport like TCP.

RDS is not Infiniband-specific; it was designed to support different transports. The current implementation used to support RDS over TCP as well as IB.

The high-level semantics of RDS from the application's point of view are

- Addressing

  RDS uses IPv4 addresses and 16bit port numbers to identify the end point of a connection. All socket operations that involve passing addresses between kernel and user space generally use a struct sockaddr_in.

  The fact that IPv4 addresses are used does not mean the underlying transport has to be IP-based. In fact, RDS over IB uses a reliable IB connection; the IP address is used exclusively to locate the remote node's GID (by ARPing for the given IP).

  The port space is entirely independent of UDP, TCP or any other protocol.

- Socket interface

  RDS sockets work *mostly* as you would expect from a BSD socket. The next section will cover the details. At any rate, all I/O is performed through the standard BSD socket API. Some additions like zerocopy support are implemented through control messages, while other extensions use the getsockopt/ setsockopt calls.

  Sockets must be bound before you can send or receive data. This is needed because binding also selects a transport and attaches it to the socket. Once bound, the transport assignment does not change. RDS will tolerate IPs moving around (eg in a active-active HA scenario), but only as long as the address doesn't move to a different transport.

- sysctls

  RDS supports a number of sysctls in /proc/sys/net/rds

## Socket Interface

AF_RDS, PF_RDS, SOL_RDS

: AF_RDS and PF_RDS are the domain type to be used with socket(2) to create RDS sockets. SOL_RDS is the socket-level to be used with setsockopt(2) and getsockopt(2) for RDS specific socket options.

fd = socket(PF_RDS, SOCK_SEQPACKET, 0);

: This creates a new, unbound RDS socket.

setsockopt(SOL_SOCKET): send and receive buffer size

: RDS honors the send and receive buffer size socket options. You are not allowed to queue more than SO_SNDSIZE bytes to a socket. A message is queued when sendmsg is called, and it leaves the queue when the remote system acknowledges its arrival.

: The SO_RCVSIZE option controls the maximum receive queue length. This is a soft limit rather than a hard limit - RDS will continue to accept and queue incoming messages, even if that takes the queue length over the limit. However, it will also mark the port as "congested" and send a congestion update to the source node. The source node is supposed to throttle any processes sending to this congested port.

bind(fd, &sockaddr_in, ...)

: This binds the socket to a local IP address and port, and a transport, if one has not already been selected via the SO_RDS_TRANSPORT socket option

sendmsg(fd, ...)

Sends a message to the indicated recipient. The kernel will transparently establish the underlying reliable connection if it isn't up yet.

An attempt to send a message that exceeds SO_SNDSIZE will return with -EMSGSIZE

An attempt to send a message that would take the total number of queued bytes over the SO_SNDSIZE threshold will return EAGAIN.

An attempt to send a message to a destination that is marked as "congested" will return ENOBUFS.

recvmsg(fd, ...)

Receives a message that was queued to this socket. The sockets recv queue accounting is adjusted, and if the queue length drops below SO_SNDSIZE, the port is marked uncongested, and a congestion update is sent to all peers.

Applications can ask the RDS kernel module to receive notifications via control messages (for instance, there is a notification when a congestion update arrived, or when a RDMA operation completes). These notifications are received through the msg.msg_control buffer of struct msghdr. The format of the messages is described in manpages.

poll(fd)

RDS supports the poll interface to allow the application to implement async I/O.

POLLIN handling is pretty straightforward. When there's an incoming message queued to the socket, or a pending notification, we signal POLLIN.

POLLOUT is a little harder. Since you can essentially send to any destination, RDS will always signal POLLOUT as long as there's room on the send queue (ie the number of bytes queued is less than the sendbuf size).

However, the kernel will refuse to accept messages to a destination marked congested - in this case you will loop forever if you rely on poll to tell you what to do. This isn't a trivial problem, but applications can deal with this - by using congestion notifications, and by checking for ENOBUFS errors returned by sendmsg.

setsockopt(SOL_RDS, RDS_CANCEL_SENT_TO, &sockaddr_in)

This allows the application to discard all messages queued to a specific destination on this particular socket.

This allows the application to cancel outstanding messages if it detects a timeout. For instance, if it tried to send a message, and the remote host is unreachable, RDS will keep trying forever. The application may decide it's not worth it, and cancel the operation. In this case, it would use RDS_CANCEL_SENT_TO to nuke any pending messages.

`setsockopt(fd, SOL_RDS, SO_RDS_TRANSPORT, (int *)&transport ..), getsockopt(fd, SOL_RDS, SO_RDS_TRANSPORT, (int *)&transport ..)`

Set or read an integer defining the underlying encapsulating transport to be used for RDS packets on the socket. When setting the option, integer argument may be one of RDS_TRANS_TCP or RDS_TRANS_IB. When retrieving the value, RDS_TRANS_NONE will be returned on an unbound socket. This socket option may only be set exactly once on the socket, prior to binding it via the bind(2) system call. Attempts to set SO_RDS_TRANSPORT on a socket for which the transport has been previously attached explicitly (by SO_RDS_TRANSPORT) or implicitly (via bind(2)) will return an error of EOPNOTSUPP. An attempt to set SO_RDS_TRANSPORT to RDS_TRANS_NONE will always return EINVAL.

## RDMA for RDS

see rds-rdma(7) manpage (available in rds-tools)

## Congestion Notifications

see rds(7) manpage

## RDS Protocol

Message header

The message header is a 'struct rds_header' (see rds.h):

Fields:

h_sequence:

per-packet sequence number

h_ack:

piggybacked acknowledgment of last packet received

h_len:

length of data, not including header

h_sport:

source port

h_dport:

destination port

h_flags:

Can be:

| | |
|---|---|
| CONG_BITMAP | this is a congestion update bitmap |
| ACK_REQUIRED | receiver must ack this packet |
| RETRANSMITTED | packet has previously been sent |

h_credit:

indicate to other end of connection that it has more credits available (i.e. there is more send room)

h_padding[4]:

unused, for future use

h_csum:

header checksum

h_exthdr:

optional data can be passed here. This is currently used for passing RDMA-related information.

ACK and retransmit handling

One might think that with reliable IB connections you wouldn't need to ack messages that have been received. The problem is that IB hardware generates an ack message before it has DMAed the message into memory. This creates a potential message loss if the HCA is disabled for any reason between when it sends the ack and before the message is DMAed and processed. This is only a potential issue if another HCA is available for fail-over.

Sending an ack immediately would allow the sender to free the sent message from their send queue quickly, but could cause excessive traffic to be used for acks. RDS piggybacks acks on sent data packets. Ack-only packets are reduced by only allowing one to be in flight at a time, and by the sender only asking for acks when its send buffers start to fill up. All retransmissions are also acked.

Flow Control

RDS's IB transport uses a credit-based mechanism to verify that there is space in the peer's receive buffers for more data. This eliminates the need for hardware retries on the connection.

Congestion

Messages waiting in the receive queue on the receiving socket are accounted against the sockets SO_RCVBUF option value. Only the payload bytes in the message are accounted for. If the number of bytes queued equals or exceeds rcvbuf then the socket is congested. All sends attempted to this socket's address should return block or return -EWOULDBLOCK.

Applications are expected to be reasonably tuned such that this situation very rarely occurs. An application encountering this "back-pressure" is considered a bug.

This is implemented by having each node maintain bitmaps which indicate which ports on bound addresses are congested. As the bitmap changes it is sent through all the connections which terminate in the local address of the bitmap which changed.

The bitmaps are allocated as connections are brought up. This avoids allocation in the interrupt handling path which queues sages on sockets. The dense bitmaps let transports send the entire bitmap on any bitmap change reasonably efficiently. This is much easier to implement than some finer-grained communication of per-port congestion. The sender does a very inexpensive bit test to test if the port it's about to send to is congested or not.

## RDS Transport Layer

As mentioned above, RDS is not IB-specific. Its code is divided into a general RDS layer and a transport layer.

The general layer handles the socket API, congestion handling, loopback, stats, usermem pinning, and the connection state machine.

The transport layer handles the details of the transport. The IB transport, for example, handles all the queue pairs, work requests, CM event handlers, and other Infiniband details.

## RDS Kernel Structures

struct rds_message
: aka possibly "rds_outgoing", the generic RDS layer copies data to be sent and sets header fields as needed, based on the socket API. This is then queued for the individual connection and sent by the connection's transport.

struct rds_incoming
: a generic struct referring to incoming data that can be handed from the transport to the general code and queued by the general code while the socket is awoken. It is then passed back to the transport code to handle the actual copy-to-user.

struct rds_socket
: per-socket information

struct rds_connection
: per-connection information

struct rds_transport
: pointers to transport-specific functions

struct rds_statistics
: non-transport-specific statistics

struct rds_cong_map
: wraps the raw congestion bitmap, contains rbnode, waitq, etc.

## Connection management

Connections may be in UP, DOWN, CONNECTING, DISCONNECTING, and ERROR states.

The first time an attempt is made by an RDS socket to send data to a node, a connection is allocated and connected. That connection is then maintained forever -- if there are transport errors, the connection will be dropped and re-established.

Dropping a connection while packets are queued will cause queued or partially-sent datagrams to be retransmitted when the connection is re-established.

## The send path

rds_sendmsg()
- struct rds_message built from incoming data
- CMSGs parsed (e.g. RDMA ops)
- transport connection alloced and connected if not already
- rds_message placed on send queue
- send worker awoken

rds_send_worker()
- calls rds_send_xmit() until queue is empty

rds_send_xmit()
- transmits congestion map if one is pending
- may set ACK_REQUIRED
- calls transport to send either non-RDMA or RDMA message (RDMA ops never retransmitted)

rds_ib_xmit()
- allocs work requests from send ring
- adds any new send credits available to peer (h_credits)
- maps the rds_message's sg list
- piggybacks ack
- populates work requests
- post send to connection's queue pair

## The recv path

rds_ib_recv_cq_comp_handler()
- looks at write completions

- unmaps recv buffer from device
- no errors, call rds_ib_process_recv()
- refill recv ring

rds_ib_process_recv()

- validate header checksum
- copy header to rds_ib_incoming struct if start of a new datagram
- add to ibinc's fraglist
- if competed datagram:
  - update cong map if datagram was cong update
  - call rds_recv_incoming() otherwise
  - note if ack is required

rds_recv_incoming()

- drop duplicate packets
- respond to pings
- find the sock associated with this datagram
- add to sock queue
- wake up sock
- do some congestion calculations

rds_recvmsg

- copy data into user iovec
- handle CMSGs
- return to application

## Multipath RDS (mprds)

Mprds is multipathed-RDS, primarily intended for RDS-over-TCP (though the concept can be extended to other transports). The classical implementation of RDS-over-TCP is implemented by demultiplexing multiple PF_RDS sockets between any 2 endpoints (where endpoint == [IP address, port]) over a single TCP socket between the 2 IP addresses involved. This has the limitation that it ends up funneling multiple RDS flows over a single TCP flow, thus it is (a) upper-bounded to the single-flow bandwidth, (b) suffers from head-of-line blocking for all the RDS sockets.

Better throughput (for a fixed small packet size, MTU) can be achieved by having multiple TCP/IP flows per rds/tcp connection, i.e., multipathed RDS (mprds). Each such TCP/IP flow constitutes a path for the rds/tcp connection. RDS sockets will be attached to a path based on some hash (e.g., of local address and RDS port number) and packets for that RDS socket will be sent over the attached path using TCP to segment/reassemble RDS datagrams on that path.

Multipathed RDS is implemented by splitting the struct rds_connection into a common (to all paths) part, and a per-path struct rds_conn_path. All I/O workqs and reconnect threads are driven from the rds_conn_path. Transports such as TCP that are multipath capable may then set up a TCP socket per rds_conn_path, and this is managed by the transport via the transport privatee cp_transport_data pointer.

Transports announce themselves as multipath capable by setting the t_mp_capable bit during registration with the rds core module. When the transport is multipath-capable, rds_sendmsg() hashes outgoing traffic across multiple paths. The outgoing hash is computed based on the local address and port that the PF_RDS socket is bound to.

Additionally, even if the transport is MP capable, we may be peering with some node that does not support mprds, or supports a different number of paths. As a result, the peering nodes need to agree on the number of paths to be used for the connection. This is done by sending out a control packet exchange before the first data packet. The control packet exchange must have completed prior to outgoing hash completion in rds_sendmsg() when the transport is mutlipath capable.

The control packet is an RDS ping packet (i.e., packet to rds dest port 0) with the ping packet having a rds extension header option of type RDS_EXTHDR_NPATHS, length 2 bytes, and the value is the number of paths supported by the sender. The "probe" ping packet will get sent from some reserved port, RDS_FLAG_PROBE_PORT (in <linux/rds.h>) The receiver of a ping from RDS_FLAG_PROBE_PORT will thus immediately be able to compute the min(sender_paths, rcvr_paths). The pong sent in response to a probe-ping should contain the rcvr's npaths when the rcvr is mprds-capable.

If the rcvr is not mprds-capable, the exthdr in the ping will be ignored. In this case the pong will not have any exthdrs, so the sender of the probe-ping can default to single-path mprds.