developer.skatelescope.org Documentation

Release 0.1.0-beta

Marco Bartolini

Dec 15, 2020

Home

CHAPTER 1

Overview

C project for testing sending and receiving blocks of data as messages over RDMA.

RDMA technology implementations include InfiniBand, internet Wide Area RDMA Protocol (iWarp), and RDMA over Converged Ethernet (RoCE). This project has been tested using RoCE version 2, which implements RDMA over UDP.

RDMA has the following communication options:

- SEND/RECV sender issues a send operation specifying memory for retrieving message and before arriving the receiver has already issued a recv operation specifying where in memory to place message payload,
- READ pulls data from remote virtual memory having only receiver active, with the sender previously providing permissions to read from its memory,
- WRITE pushes data into remote virtual memory having only sender active, with the receiver previously providing permissions to write to its memory.
- Atomic extensions to fetch and add, or compare and swap, at a specified virtual address.

This project has chosen to use SEND/RECV communication (with additional immediate 4 byte values) so that the sender and receiver can each separately control what memory regions get utilised on its end.

RDMA has the following tranport modes:

- Reliable connection (RC), where messages are reliably delivered in order to one receiver,
- Unreliable connection (UC), where packets comprising a message may be lost and error handling is left to the application,
- Unreliable datagram (UD), where individual (MTU) packet messages are tranmitted and/or multicast.

This project has chosen to use UC, so if a packet is lost en route the message is not delivered. Furthermore, if the receiver has not posted a receive request in its completion queue before the sender's message is received the message will be silently dropped.

RDMA SEND/RECV allows for each end to track completions of its (send or receive) work requests either by:

- busy polling where the completion queue is repeatedly polled until a completion occurs
- waiting for completion events, where the application thread blocks until notified of a completion.

This project has chosen to use waiting for completion events, to reduce the CPU processor load. As completion events incur an overhead the application sender only signals completions for the final message in a region of contiguous messages.

Further information on RDMA can be obtained from https://www.mellanox.com/related-docs/prod_software/RDMA_Aware_Programming_user_manual.pdf

CHAPTER 2

Software and hardware requirements

The ethernet adapter card on the sender and receiver need to both be configured for RDMA, such as RoCE V2. For a ConnectX-4/4LX/5 adapter cards (which may show as two RDMA device each with a single RDMA port 1) recommendations are available at https://community.mellanox.com/s/article/recommended-network-configuration-examples-for-roce-deployment#jive_content_id_Recommended_Configurations. For an older ConnectX-3Pro card (which may show as RDMA device "mlx_4" with RDMA ports 1 and 2) there are some archived instructions at https://community.mellanox.com/s/article/howto-configure-roce-v2-for-connectx-3-pro-using-mellanox-switchx-switches.

Then the instructions at https://www.rdmamojo.com/2014/11/08/working-rdma-ubuntu/ were followed to install the RDMA packages for Ubuntu (some packages such as libdapl2, rds-tools and libibcommon1 were non longer available, and libopensm5 was replaced with libopensm5a). Firmware updates for the card and up to date packages are described at https://www.mellanox.com/pdf/prod_software/Ubuntu_18_04_Inbox_Driver_User_Manual.pdf.

CHAPTER 3

Building the project

The project only has basic C code, and does not use cmake. So it is build simply by make:

```
make all
```

which includes compiler optimisations.

A debug build can be produced using:

```
make debug
```

which is useful for tools such as Valgrind. Please be aware that this only is a very basic makefile and doesn't separate the debug and release builds in separate folders.

CHAPTER 4

Making an RDMA library

The RDMAapi, RDMAlogger, and RDMAmetrics can optionally be bundled together into a library for use in other projects. In Linux once they are compiled into their respective .o files either a new shared library called librdmaapi.so can be created:

```
gcc -shared -o librdmaapi.so RDMAapi.o RDMAmetricreporter.o  RDMAmemorymanager.o␣
↪RDMAexchangeidcallbacks.o RDMAlogger.o
```

or else a new static library called librdmaapi.a can be created:

```
ar -rcs librdmaapi.a RDMAapi.o RDMAmetricreporter.o  RDMAmemorymanager.o␣
↪RDMAexchangeidcallbacks.o RDMAlogger.o
```

The shared or static library should be made available at runtime in a default folder such as /usr/lib:

```
sudo cp librdmaapi.so /usr/lib
sudo chmod 755 /usr/lib/librdmaapi.so
sudo ldconfig -v -n /usr/lib
```

and the corresponding header files copied to a folder such as /usr/include:

```
sudo cp RDMAapi.h /usr/include
sudo cp RDMAexchangeidcallbacks.h /usr/include
sudo cp RDMAlogger.h /usr/include
sudo cp RDMAmemorymanager.h /usr/include
sudo cp RDMAmetricreporter.h /usr/include
```

Then the library can be included as a C library in other projects, such as in C++:

```
#ifdef __cplusplus
    extern "C"
    {
#endif
#include <RDMAapi.h>
#ifdef __cplusplus
```

(continues on next page)

(continued from previous page)

```
    }
#endif
```

and included along with its library dependencies when building other projects:

```
-lrdmaapi -lrdmacm -libverbs -lcurl -lpthread -lm
```

On at least one installation some of the RDMA packages required by librdmaapi have not been built using position independent code, resulting in the RDMA api failing to create some of the necessary RDMA resources (eg event queues). In this case the static librdmaapi.a library should be used instead of the shared librdmaapi.so library.

CHAPTER 5

Running the project

The project has various command line options:

- -l to specify the log level output to the console using RFC5424 (either 0=LOG_EMERG, 1=LOG_ALERT, 2=LOG_CRIT, 3=LOG_ERR, 4=LOG_WARNING, 5=LOG_NOTICE, 6=LOG_INFO, 7=LOG_DEBUG, default is LOG_NOTICE).
- -m to specify the individual message size in bytes (default is 65536).
- -b to specify how many memory region blocks to allocate (default is 1).
- -c to specify how many contiguous messages to have in each memory region, to help encourage utilisation of transparent huge pages and reduce the number of completion events signalled by the sender (default is 1). Each memory region is allocated to hold this number of messages.
- -f to specify a filename to use for either the receiver to save the memory block data or else sender to load the memory block data (note .0, .1, ... get appended to the filename for each memory block).
- -t to specify the total number of message intended to send or receive (default is the total number of messages across all memory regions). Note that this should be a multiple of the number of contiguous messages as all the messages in a memory region are sent/received as a linked list of work requests for efficiency. If the total is greater than the number of messages on the sender or receiver it simply loops back to repeat earlier work requests. Note the receive will block until it successfully receives the intended number of messages (so hang if it received any messages while it could not keep up with the sender and had its receive request queue emptied).
- -d to specify the time to delay after each send or receive (default is 0 for no delay). This option is intended to throttle the sender so the receiver can better handle the message rate, or simulate delays due to processing on the receiver.
- -r to specify the RDMA device name (default is NULL to let the application itself find a suitable RDMA device).
- -p to specify the device port (default is 1). Note when using a sender with localhost that some RDMA devices offer a second RDMA port which can be used for testing on localhost without only local loopback.
- -g to specify a specific index in the GID table to use (default is to let the application try to find a GID that supports IPv6 or else fall back to IPv4). Note the choice of GID will determine which RDMA technology (such as version of RoCE) will be utilised.

- -x to specify an exchange identifier filename for exchanging PSN,QPN,GID,LID identifiers rather than via stdio (note .send or .recv suffix gets appended to filename which should be on shared file system accessible to both sender and receiver).
- -o to specify an IP address to which InfluxDB metrics should be pushed via HTTP POST requests.
- -v to specify approximate number of messages over which to average metrics before they get pushed (default is to output each time after all memory regions transferred).
- -s to have the project run in send rather than receive mode.

Using the -s option changes the receiver to instead behave as a sender of RDMA messages.

```
./receive [-l log level 0..6] [-m message size in bytes] [-b num memory blocks] [-c␣
↪num contig messages per block] [-f data filename] [-t total num messages] [-d delay␣
↪microsec time per message] [-r RDMA device name] [-p device port] [-g gid index] [-
↪x exchange identifier filename] [-o metric url] [-v metric averaging] [-s]
```

For example, to start an application that will receive 8k messages with memory buffers holding 32 contiguous messages per buffer and a total of 16 memory buffers, to receive a default total of 16*32=512 messages:

```
./receive -m8192 -b16 -c32
```

To start an equivalent application that will send 512 messages including a 10 microsecond delay after each one (to throttle the sending a bit) to the RDMA device at localhost:

```
./receive -m8192 -b16 -c32 -d10 -s
```

The sender will then display its packet sequence number (PSN), its queue pair number (QPN), its 16 byte global identifier (GID), and its local identifier (LID, typically 0). The receiver will query for these to be entered, and then will display its PSN, QPN, GID, LID which must also be entered for the receiver.

On the receiver a output similar to the following should be observed:

```
************************************************************************
Please enter the remote sender packet sequence number (psn): 13569919
Please enter the remote sender queue pair number (qpn): 1373
Please enter the remote sender global identifier as 16 bytes (gid): 254-128-0-0-0-0-0-
↪0-238-13-154-255-254-38-42-64
Please enter the remote sender local identifier (lid): 0
Please pass to the remote sender: psn = 10107130
Please pass to the remote sender: qpn = 1372
Please pass to the remote sender: gid = 254-128-0-0-0-0-0-0-238-13-154-255-254-38-42-
↪64
Please pass to the remote sender: lid = 0
************************************************************************
Receiver wall time duration since first message received is 1 milliseconds and CPU␣
↪clock time duration is 0 milliseconds (82% utilisation)
Receiver bandwidth is 30.98 Gbps
Receiver detected 0 missing messages from 512 total (0% missing)
########################################################################
Memory contents:
  Memory region 0000:  00 01 02 03 04 05 06 07 08 09 ... f6 f7 f8 f9 fa fb fc fd fe ff
  Memory region 0001:  00 01 02 03 04 05 06 07 08 09 ... f6 f7 f8 f9 fa fb fc fd fe ff
  Memory region 0002:  00 01 02 03 04 05 06 07 08 09 ... f6 f7 f8 f9 fa fb fc fd fe ff
  Memory region 0003:  00 01 02 03 04 05 06 07 08 09 ... f6 f7 f8 f9 fa fb fc fd fe ff
  Memory region 0004:  00 01 02 03 04 05 06 07 08 09 ... f6 f7 f8 f9 fa fb fc fd fe ff
  Memory region 0005:  00 01 02 03 04 05 06 07 08 09 ... f6 f7 f8 f9 fa fb fc fd fe ff
  Memory region 0006:  00 01 02 03 04 05 06 07 08 09 ... f6 f7 f8 f9 fa fb fc fd fe ff
```

(continues on next page)

(continued from previous page)

```
  Memory region 0007:  00 01 02 03 04 05 06 07 08 09 ... f6 f7 f8 f9 fa fb fc fd fe ff
  Memory region 0008:  00 01 02 03 04 05 06 07 08 09 ... f6 f7 f8 f9 fa fb fc fd fe ff
  Memory region 0009:  00 01 02 03 04 05 06 07 08 09 ... f6 f7 f8 f9 fa fb fc fd fe ff
  Memory region 0010:  00 01 02 03 04 05 06 07 08 09 ... f6 f7 f8 f9 fa fb fc fd fe ff
  Memory region 0011:  00 01 02 03 04 05 06 07 08 09 ... f6 f7 f8 f9 fa fb fc fd fe ff
  Memory region 0012:  00 01 02 03 04 05 06 07 08 09 ... f6 f7 f8 f9 fa fb fc fd fe ff
  Memory region 0013:  00 01 02 03 04 05 06 07 08 09 ... f6 f7 f8 f9 fa fb fc fd fe ff
  Memory region 0014:  00 01 02 03 04 05 06 07 08 09 ... f6 f7 f8 f9 fa fb fc fd fe ff
  Memory region 0015:  00 01 02 03 04 05 06 07 08 09 ... f6 f7 f8 f9 fa fb fc fd fe ff
########################################################################
```

And likewise on the sender:

```
************************************************************************
Please pass to the remote receiver: psn = 13569919
Please pass to the remote receiver: qpn = 1373
Please pass to the remote receiver: gid = 254-128-0-0-0-0-0-0-238-13-154-255-254-38-
↪42-64
Please pass to the remote receiver: lid = 0
Please enter the remote receiver packet sequence number (psn): 10107130
Please enter the remote receiver queue pair number (qpn): 1372
Please enter the remote receiver global identifier as 16 bytes (gid): 254-128-0-0-0-0-
↪0-0-238-13-154-255-254-38-42-64
Please enter the remote receiver local identifier (lid): 0
************************************************************************
########################################################################
Memory contents:
  Memory region 0000:  00 01 02 03 04 05 06 07 08 09 ... f6 f7 f8 f9 fa fb fc fd fe ff
  Memory region 0001:  00 01 02 03 04 05 06 07 08 09 ... f6 f7 f8 f9 fa fb fc fd fe ff
  Memory region 0002:  00 01 02 03 04 05 06 07 08 09 ... f6 f7 f8 f9 fa fb fc fd fe ff
  Memory region 0003:  00 01 02 03 04 05 06 07 08 09 ... f6 f7 f8 f9 fa fb fc fd fe ff
  Memory region 0004:  00 01 02 03 04 05 06 07 08 09 ... f6 f7 f8 f9 fa fb fc fd fe ff
  Memory region 0005:  00 01 02 03 04 05 06 07 08 09 ... f6 f7 f8 f9 fa fb fc fd fe ff
  Memory region 0006:  00 01 02 03 04 05 06 07 08 09 ... f6 f7 f8 f9 fa fb fc fd fe ff
  Memory region 0007:  00 01 02 03 04 05 06 07 08 09 ... f6 f7 f8 f9 fa fb fc fd fe ff
  Memory region 0008:  00 01 02 03 04 05 06 07 08 09 ... f6 f7 f8 f9 fa fb fc fd fe ff
  Memory region 0009:  00 01 02 03 04 05 06 07 08 09 ... f6 f7 f8 f9 fa fb fc fd fe ff
  Memory region 0010:  00 01 02 03 04 05 06 07 08 09 ... f6 f7 f8 f9 fa fb fc fd fe ff
  Memory region 0011:  00 01 02 03 04 05 06 07 08 09 ... f6 f7 f8 f9 fa fb fc fd fe ff
  Memory region 0012:  00 01 02 03 04 05 06 07 08 09 ... f6 f7 f8 f9 fa fb fc fd fe ff
  Memory region 0013:  00 01 02 03 04 05 06 07 08 09 ... f6 f7 f8 f9 fa fb fc fd fe ff
  Memory region 0014:  00 01 02 03 04 05 06 07 08 09 ... f6 f7 f8 f9 fa fb fc fd fe ff
  Memory region 0015:  00 01 02 03 04 05 06 07 08 09 ... f6 f7 f8 f9 fa fb fc fd fe ff
########################################################################
Sender wall time duration is 1 milliseconds and CPU clock time duration is 0␣
↪milliseconds (19% utilisation)
Sender bandwidth is 27.24 Gbps
```

For the SKA DP System Demo 8.6 where a longer run was used with RDMA metrics output to a separate InfluxDB/Grafana instance (at 156.62.60.71) the sender was run via a script:

```
./receive -b128 -t10000000 -v500000 -xtemprun -s
rm -f temprun.*
```

and the receiver likewise via the script:

```
./receive -b128 -t10000000 -v25000 -xtemprun -o156.62.60.71
rm -f temprun.*
```

CHAPTER 6

Developer notes for using the RDMA api

The `RDMAapi.c` file has the necessary functions for querying available RDMA devices, opening them, creating and modifying RDMA queue pairs, allocating necessary RDMA resources such as a protection domain and completion queues, setting up an RDMA connection for a chosen device, waiting for completion queue events, and later cleaning up the RDMA resources. These can be used to build an application-specific RDMA send/receive workflow.

It also has functions `sendWorkRequests` and `receiveWorkRequests` for sending or receiving a specified number of RDMA messages, where the sender labels each message's immediate data with an unsigned 32-bit ordinal number (starting at 0 and wrapping around at `UINT32_MAX`), so the receiver can track missing messages, while also optionally providing metrics to an InfluxDB instance. For convenience these methods can be called via `rdmaTransferMemoryBlocksWithDefaultDevice` or `rdmaTransferMemoryBlocks` to have the sending/receiving performed in a separate thread (advisable as at high data rates the thread needs to keep the RDMA queues sufficiently populated to reduce the risk of losing messages, so should not perform any other time-demanding tasks). These convenience functions are passed a `MemoryRegionManager` which can track the progress of the transfers, and eventually the `waitForRdmaTransferCompletion` needs to be called to block until the thread has completed.

The file `ReceiveVisibilities.c` demonstrates how the RDMA api might be used via the following steps:

- Allocate memory blocks that will hold the data sent/received. Each of the `numMemoryBlocks` memory blocks can hold `numContiguousMessages>=1` contiguous RDMA messages, in case an application may find using large blocks of memory convenient for caching. For convenience, the RDMA api function `allocateMemoryBlocks` can be called to allocate page-aligned memory blocks in RAM:

```
uint64_t memoryBlockSize = messageSize * numContiguousMessages;
void **memoryBlocks = allocateMemoryBlocks(memoryBlockSize, &numMemoryBlocks);
```

  Instead, the application might itself want to handle the memory allocation, for example allocating memory in CUDA that is page-locked for a GPU device:

```
for (unsigned int blockIndex=0; blockIndex<numMemoryBlocks; blockIndex++)
{
    cudaHostAlloc(&memoryBlocks[blockIndex], numInputBytes*numContiguousMessages,␣
↪0);
}
```

Note that if `numTotalMessages>numMemoryBlocks*numContiguousMessages` then the memory blocks will get reused until all messages are transferred. In this case a sending application is responsible for repopulating the memory blocks with further data before they get enqueued again, and a receiving application is responsible for utilising the received data within each block before it gets overwritten with further data.

- Create a `MemoryRegionManager` that will coordinate the RDMA transfers with the allocated memory blocks, and prepare them as RDMA memory *regions*, optionally specifying whether the `MemoryRegionManager` should be checked on the sender to ensure each memory region is populated with data before it gets enqueued to send and on the receiver each memory region is not (still) populated with data before it is enqueued to receive:

```
MemoryRegionManager* manager = createMemoryRegionManager(memoryBlocks,␣
↪messageSize, numMemoryBlocks, numContiguousMessages, numTotalMessages, false);
```

- Optionally create a custom exchange identifier callback if standard I/O is not to be used for exchanging the four RDMA identifiers.
- Prepare the RDMA device and perform the RDMA transfers either in `SEND_MODE` or in `RECV_MODE`, optionally using a non-null URL for an InfluxDB instance to where transfer metrics get reported:

```
rdmaTransferMemoryBlocks(mode, manager, messageDelayTime,␣
↪identifierExchangeFunction, rdmaDeviceName, rdmaPort, gidIndex, metricURL,␣
↪numMetricAveraging);
```

- Optionally monitor and respond as messages are transferred, particularly if the memory blocks are to be reused during the transfer:

```
/* process the message data as they get transferred */
uint64_t expectedTransfer = 0;
uint32_t currentRegionIndex = 0;
uint32_t currentContiguousIndex = 0;
while (expectedTransfer < numTotalMessages)
{
    /* delay until the current memory location has its data transferred */
    uint64_t currentTransfer = getMessageTransferred(manager, currentRegionIndex,␣
↪currentContiguousIndex);
    while (currentTransfer==NO_MESSAGE_ORDINAL || currentTransfer
↪<expectedTransfer)
    {
        /* expected message transfer has not yet taken place */
        microSleep(10); /* sleep current thread before checking currentTransfer␣
↪again */
        currentTransfer = getMessageTransferred(manager, currentRegionIndex,␣
↪currentContiguousIndex);
    }
    /* process the current message transfer */
    if (currentTransfer > expectedTransfer)
    {
        /* messages between expectedTransfer and currentTransfer-1 inclusive have␣
↪been dropped during the transfer */
        /* HERE: do something about the missing messages */
    }
    /* note message data for currentTransfer is now available in␣
↪memoryBlocks[currentRegionIndex] at start index␣
↪currentContiguousIndex*messageSize */
    /* HERE: do some processing with the current message transfer */
    /* once done processing the current message transfer move along to the next␣
↪memory location */
```

(continues on next page)

(continued from previous page)

```
    expectedTransfer = currentTransfer + 1;
    currentContiguousIndex++;
    if (currentContiguousIndex >= manager->numContiguousMessages)
    {
        /* move along to the start of the next memory region */
        setMemoryRegionPopulated(manager, currentRegionIndex, false); /* finished␣
↪processing this memory region so it can now be reused */
        currentContiguousIndex = 0;
        currentRegionIndex++;
        if (currentRegionIndex >= manager->numMemoryRegions)
        {
            /* wrap around to reuse the same memory regions */
            currentRegionIndex = 0;
        }
    }
}
```

- Ensure the RDMA transfers have completed by making the application thread wait:

```
waitForRdmaTransferCompletion(manager);
```

- Destroy the `MemoryRegionManager`:

```
destroyMemoryRegionManager(manager);
```

- Free the allocated memory blocks. If the `allocateMemoryBlocks` convenience function was used to create them then the `freeMemoryBlocks` convenience function can be used to free them:

```
freeMemoryBlocks(memoryBlocks, numMemoryBlocks);
```

  Otherwise the application is itself responsible for freeing any allocated memory, for example in CUDA to free host memory:

```
for (unsigned int blockIndex=0; blockIndex<numMemoryBlocks; blockIndex++)
{
    cudaFreeHost(memoryBlocks[blockIndex]);
}
```

CHAPTER 7

Using a custom exchange identifier callback function

By default the RDMA api uses standard I/O to present and receive the four identifiers that must be exchanged beween each of the sender and receiver with the other:

- Packet Sequence Number (PSN), which is a 32-bit unsigned integer. Note the RDMA api uses a random number generator to produce a random 24-bit PSN on each end (as is typical for software that utilises RDMA), but this number could instead be fixed at each end to avoid its exchange.
- Queue Pair Number (QPN), which is a 32-bit unsigned integer. Note that this is provided by the RDMA device when the queue pair is created. Current RDMA devices appear to sequentially increment its value by one each time a new queue pair is created on the device.
- Global Identifer (GID), which is a 128-bit unsigned integer (represented in C as a `union ibv_gid`), whose first 64 bits give a subnet prefix and remaining 64 bits give an interface ID. Amongst other information the GID encapsulates the IP address for the device. The RDMA device may offer a table of GID address options (by default the RDMA api will default to choosing an available IPv6 GID over an IPv4 if a preferred GID index is not specified). In Linux the available GIDs are in folders `/sys/class/infiniband/$dev/ports/$port/gids` which can be displayed using the Mellanox script `show_gids`.
- Local Identifier (LID), which is a 16-bit unsigned integer. Note this is an attribute of the RDMA device port at layer 2 of the Infiniband protocol stack, which current RoCE implementations set to 0.

The default standard I/O exchange of identifiers is handled in the `RDMAexchangeidcallbacks.c` callback function `exchangeViaStdIO`. This C function takes a `bool` parameter stating whether the RDMA api is running as a sender (`true`) or receiver (`false`), along with the four local identifiers followed by pointers for storing the four remote identifiers when obtained. It has return type `enum exchangeResult` returning `EXCH_SUCCESS` (int value 0) if the exchange has been successful, or non-zero if the exchange was not successful.

A custom callback can instead be used by the RDMA api to exchange the identifiers provided it has the same signature:

```
enum exchangeResult (*identifierExchangeFunction)(bool isSendMode,
    uint32_t packetSequenceNumber, uint32_t queuePairNumber, union ibv_gid gidAddress,
↪ uint16_t localIdentifier,
    uint32_t *remotePSNPtr, uint32_t *remoteQPNPtr, union ibv_gid *remoteGIDPtr,␣
↪uint16_t *remoteLIDPtr),
```

One example of this which exchanges the identifiers using a shared file system is the `RDMAexchangeidcallbacks.c` callback function `exchangeViaSharedFiles`.

Todo:

- Insert todo's here

CHAPTER 8

Package-name documentation

This section describes requirements and guidelines.

8.1 Subtitle

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

8.1.1 Public API Documentation

Functions

Classes

CHAPTER 9

Project-name documentation HEADING

These are all the packages, functions and scripts that form part of the project.