NAAICE API for RMA-Basic-Communication

Florian Mikolajczak, Max Schrötter, Bettina Schnor
NAAICE AP1 (E1.B)
University of Potsdam

Contents

1 Introduction **2**

2 Host-NAA Communication Protocol **3**

3 API **12**

3.1 Client API 17

3.2 Software-NAA API 27

1 Introduction

This documentation concerns the low-level API, which is used a basis for remote direct memory access (RDMA) based communication between an HPC node and a network attached accelerator (NAA) on a field programmable gate array (FPGA). As part of the NAAICE project, RDMA communication between the two is used to facilitate fast data transfer for the offloading of specific calculations. The offloading of specific calculations onto an NAA can lead to a reduction in overall runtime and energy consumption of a given HPC job.
The API is written in `C` and makes use of two libraries for user-space based RDMA communication. These libraries are `libibverbs` and `librdmacm`, which are both part of the `rdma-core` library. `libibverbs` and `librdmacm` are implementations of the Infiniband standard. Within the project, the used network protocol is RoCEv2 (RDMA over Converged Ethernet 2). RoCEv2 encapsulates Infiniband packets within UDP/IP packets, thus making them routeable within an IP-based HPC center. `libibverbs` provides the functionalities for setting up Infiniband communication structures, queue pairs (QP), as well as other structures such as queues for sending and receiving operations as well as a completion queue, where finished operations are noted. It encapsulates operations for sending and receiving messages or data in a two-sided (`IBV_SEND/IBV_RECV`) or one-sided `IBV_WRITE/IBV_READ` style. `librdmacm` provides a method for setting up an Infiniband-based connection.

As a short introduction, the key structures and components of an Infiniband connection will be discussed. The reader is referred to the Infiniband standard for more detailed introduction [2]. Chapter 3 of the standard gives a broad introduction to the overall architecture. Generally, knowledge of socket-based communication is assumed, but can be found in [4]. In Infiniband communication, queue pairs (QPs) are analogous to sockets. Communication operations include two-sided ones like `SEND` and `RECEIVE` operations. These explicitly synchronize the two communication partners and a `SEND` operations cannot be done before the receiving side issues a `RECEIVE`. Additionally, one-sided communication is available as well, using `WRITE` or `READ` operations, as well as `ATOMICS`. These one-sided operations are also known as remote direct memory access operations (RDMA), since the CPU or user-space application on the receiving side is not involved in the communication. A schematic of RDMA operations as they will be used within the NAAICE project using RoCEv2 is shown in figure 1.

All communication requests are handled using work queues. These make up the aforementioned queue pairs. The different queues are: send queues and receive queues as well as completion queues. Send and receive queues are used to post work requests that will be handled by the network interface card (NIC). Finished work requests (or failures) are reported in the completion queue. Additionally, memory buffers that are used for the transfer of data exist. These are called memory regions. Memory regions have to be allocated and registered to the device. After registration, the NIC can access these buffers without involvement of the CPU, a prerequisite for RDMA operations. Memory regions are identified by an address and length. Accessing memory regions is restricted through a set of a local and remote key (`rkey, lkey`). A memory region can be split into different memory windows (MW) with their own keys. Many memory regions can be associated with a single queue pair. As an overarching structure, protection domains also exist. This structure combines memory regions and queue pairs and is used to

provide a mechanism for access management. A memory region is registered to a protection domain and a queue pair allocated to a protection domain. Any given key set of a memory region is only valid on queue pairs from the same protection domain [2, p. 107]. Figure 1 summarizes the relationship of the different components of an Infiniband connection. All features mentioned above except memory windows are used so far within the API for AP 1.

2 Host-NAA Communication Protocol

Prior to implementing a low-level API, the communication pattern between an HPC node and an NAA was discussed and explored. For this, a communication sequence diagram was created by HHI, ZIB and UP. Within discussions for the communication sequence diagram, a protocol for exchanging memory region metadata was designed as well. The sequence diagram is shown in figure 2. The communication between the HPC node and the NAA on the FPGA is analogous to the general client-server model. The HPC node acts as the client and initiates the connection. By transferring data and a function code (AP 2), it also initiates calculations on the FPGA in the style of an asynchronous RPC. The FPGA takes on the role of the server, waits for connection requests and responds to an RPC with the result data. The communication can be split up in four different sections: The connection establishment, the setup of memory regions, the data transfer and the connection termination.

Connection Establishment

The connection management is handled by functions from the library `librdmacm`. Much like in TCP, connection establishment is done by a three-way handshake. The client sends a connection request (`REQ`), to which the server replies with a Reply message `REP`. Finally, the connection is fully established with a Ready-to-Use (`RTU`) message by the

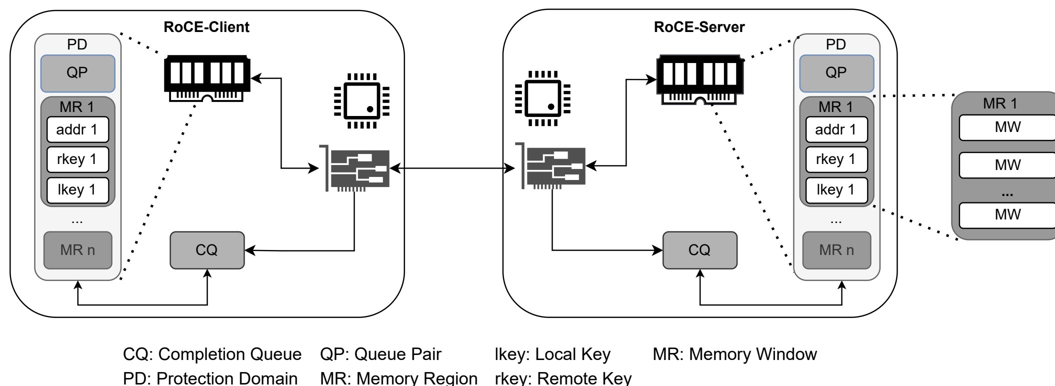


Figure 1: Schematic figure of RDMA communication between two hosts. As in the source code for AP 1, one host takes on the role of a RoCE client, initiating a connection as depicted in figure 2 and requesting the calculation of a given problem by the RoCE server in an RPC-like manner through the transfer of input data. The data structures within the memory, e.g. memory regions are registered to the NIC, such that no CPU involvement is necessary during the data transfer.

client. Other message types for unsuccessful connections can be found in the Infiniband Standard chapter 12.6 [2]. All message types for connection establishment defined in the Infiniband standard are also implemented on the FPGA. During AP1 the IP address of the server is a user argument for the client program. The server listens for connection requests on a specific and known port. In future developments, the user can ask the resource management system of the computer center for an FPGA with the desired capabilities and will receive information on its IP address and the port number for connection establishment.

Memory Region Setup Protocol (MRSP)

Before data can be transferred, meta information about the memory regions, i.e., buffers registered to the NIC, has to be exchanged. Within the project, a protocol for memory region setup has been developed. Generally, the client announces memory regions for the transfer of results from the server to the client and requests memory regions on the server for its input data. The advertisement for a memory region includes the address of the region as well as the length and the remote key for access control. The request for a memory region includes a valid physical FPGA address, the size and memory region flags. These are unused so far, but can in the future be used to request regions on the FPGA that will not be used for data transfer.
The MRSP is done using two messages. All memory regions are announced and requested by the client in one message. The server than handles this announcement and request and returns the virtual addresses and `rkeys` for the requested memory regions in a single message back to the client.

Earlier versions of the MRSP exist: First, a static approach was used. A single memory region is announced through a single message and the amount of requested memory on the server is requested through one single message as well. With this approach, the number of messages scales linearly with the amount of memory regions to exchange, requiring n advertisements and 1 request from the client and n advertisements from the server. As a second approach, a dynamic exchange of memory regions was developed. Herein, all announced memory regions are sent within a single message of variable size. Thus, both communication partners can announce the meta-information for all registered memory regions in one message per direction (excluding acknowledgements). Within the same message or another message type, the HPC node also requests an amount of memory to be registered in the server for data transfer. The different message types will be discussed in detail in the next section. The server then responds with the advertisement for the memory region(s) it has allocated in accordance with the overall size requested by the client. Therefore, the exchange of memory regions is done in 2 messages (excluding acknowledgements) regardless of the number of requested memory regions.

In another update, the user now announces host memory regions and FPGA memory regions of the same size in the same message. This is necessary, since the memory management of the FPGA will be outsourced to a service of the resource management system (RMS) or compute cluster. The user will receive valid FPGA memory addresses through this service and request these addresses on the FPGA. The current implementation might change in the future, based on the performance of different data transfer styles. This if further discussed in section 2.

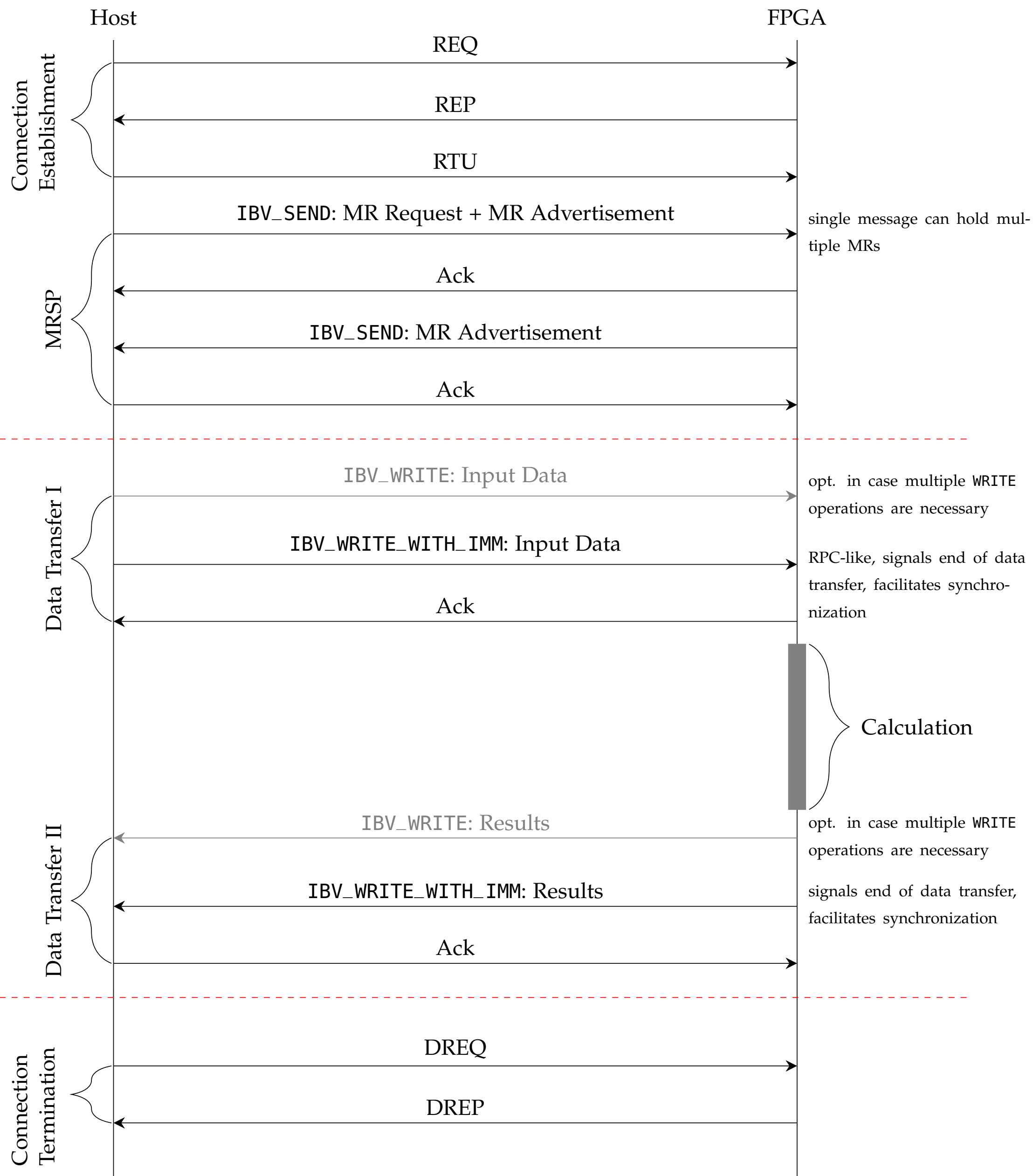


Figure 2: Communication sequence diagram between an HPC node and an NAA on an FPGA. The communication pattern was designed as part of AP 1. The communication follows a server-client model, where the HPC node is the initiating client. The FPGA takes on the role of the server and responds to the connection. Similarly, the communication is RPC-like, where the end of the data transfer signals the FPGA that calculation or work on the request can now begin. In the reverse direction, the end of data transfer back to the client signals the end of the RPC. The actual synchronization is explicitly facilitated by the RDMA operation used: `IBV_WRITE_WITH_IMM`.

The communication pattern for the exchange of memory region information is two-sided. Thus, `IBV_SEND` operations are used, for which a prior `IBV_RECV` work request has to be posted by the receiver. If the receiver has not posted such a request, the transfer

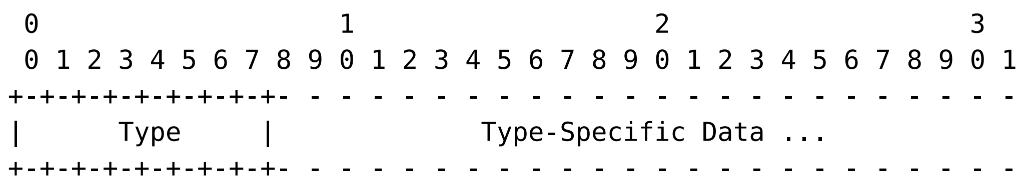
will not work. Depending on the configuration, the sender will retry the transfer a number of times after a short pause. Note, that unsuccessful transfers are not easily spotted in network traffic recorded by tools like `wireshark`. In any case, the receiver will respond with an `ack` message. However, in unsuccessful transfers the `ack`-specific header includes a `syndrome` value of 32 instead of 0 for successful transfers.

Message Types

For the new memory setup protocol, specific message types were defined. This document limits itself to most recent implementation including the message type of the dynamic memory region setup protocol with FPGA addresses. The memory exchange protocol requires a total of 3 different messages so far.

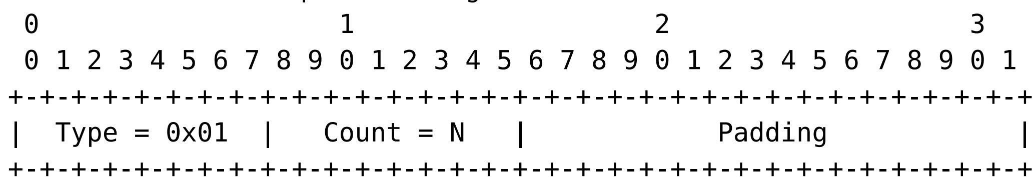
- 0x00 Error
  A generic message including an error code to communicate errors during the MRSP.
- 0x01 Advertisement and Request
  Used by the client to announce its memory regions and request a specific amount of memory on the server
- 0x02 Advertisement
  Used by the server to announce its memory regions after a request. *Note: This message type could also be used by the client, requiring a specific request message. This is already defined, but not used.*

All messages share a similar structure as shown below:



where type denotes the message types defined in the list above.
An Advertisement and Request message can include up to 256 memory regions, limited only be the `count` variable, which indicates the number of announced memory regions. This variable has 8 bits but can be expanded to use up to 24 bits (including 16 bits now used for padding). However, the current limit of 256 memory regions to be exchanged vastly outnumbers the amount of memory regions the FPGA can handle. In the current implementation, the FPGA is only able to handle up to 32 memory regions. Note: For regular compute nodes using `ibverbs`, the amount of memory available for registration is hardware-specific as well. In practice, the size of a single memory region is effectively limited by the maximum message transfer size defined by the InfiniBand standard to 2 GB [3, Chapter 9.3.3.3, C9-9].

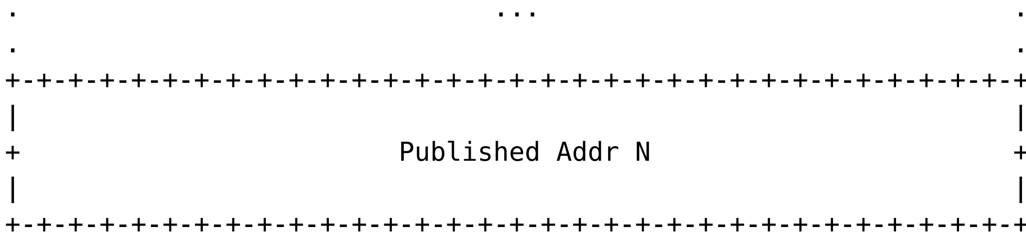
`Advertisement and Request Message`



```
|   MR Flags    |           Requested                           |
+-+-+-+-+-+-+-+-+                                               +
|                          FPGA address  1                      |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
+                         Published Addr 1                      +
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                         Published RKey 1                      |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                         Published Size 1                      |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
.                                                               .
.                               ...                             .
.                                                               .
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|   MR Flags    |           Requested                           |
+-+-+-+-+-+-+-+-+                                               +
|                          FPGA address  N                      |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
+                         Published Addr N                      +
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                         Published RKey N                      |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                         Published Size N                      |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

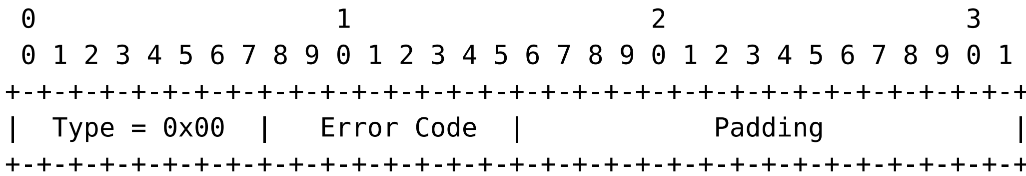
The Advertisement and Request message includes the metadata for each memory region to be announced, i.e. the starting address, size and the remote key, which needs to be included in each RDMA request later on. The Advertisement message is a part of the Advertisement and Request message, excluding the request part and is, up to now, solely used by the RDMA server. Its structure is depicted below:

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Type = 0x02  |   Count = N   |            Padding            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
+                         Published Addr 1                      +
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                         Published RKey 1                      |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                         Published Size 1                      |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
.                                                               .
```



```
.                              ...                              .
.                                                               .
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
+                        Published Addr N                       +
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Lastly, the error message only includes an error code:



```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Type = 0x00  |   Error Code  |            Padding            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

The 8-bit variable allows 256 different error codes and can be extended if more codes are necessary.

The first set of error codes have already been defined. The error message is self will only be used for errors related to the exchange of memory region. For the connection management, messages signaling errors are already included and error handling is part of the `librdmacm` library. Error in data transfer are reported in the completion queue and do not need a specific error messages. Errors during the calculation are reported by the server as the immediate value within the `IBV_WRITE_WITH_IMM` message.

The error codes defined for the exchange of memory regions so far are:

- 0x01: Not enough memory available (address + size exceeds the memory boundary)
- 0x02: invalid address (address is out of bounds)
- 0x03: Too many regions requested

Data Transfer (and calculations)

Data transfer occurs twice. First the HPC node sends input data to the NAA. Data will be using one-sided RDMA operations. If the data transfer can be done in one operation, i.e. only one MR is transferred, an `IBV_WRITE_WITH_IMM` (Write with immediate data) will be used. The immediate data is a 32-bit value that is transferred with the data. This immediate data is used to transfer the function code, i.e. which calculation is to be done by the FPGA. If $n \mid n > 1$ operations are used, $n - 1$ `IBV_WRITE` operations are done and a single last `IBV_WRITE_WITH_IMM`. Unlike for simple `IBV_WRITE` operations, the receiver has to post a receive work request, as seen above in case of `IBV_SEND` operations. Using this, explicit synchronization after the last data transfer between sender and receiver is done. The communication style of the data transfer is further discussed in section 2.
Next, the NAA does calculations, while the HPC node waits for response. Whether explicit waiting or polling is used by the HPC node is outside the scope of this document and will be handled in AP 2. After calculations are done, the FPGA sends results data back in the same way it received data. Again, the last transfer operation will be an `IBV_WRITE_WITH_IMM`, synchronizing both communication partners. An immediate

value of 0 indicates success. Errors are reported with a positive integer as the immediate value. The error codes defined so far are:

- 0x01: Socket not available
- 0x02: Kernel timeout
- 0x03-0x0f: reserved
- 0x10 - 0x7f: Application / calculation errors

Connection Termination

Connection termination is again done through methods provided by `librdmacm`. The HPC node (client) sends a disconnect request (`DREQ`), message to the FPGA (server). The FPGA replies with a disconnect reply (`DREP`) message, terminating the connection. More information on connection management can be found in the Infiniband standard chapter 12.6 [2]. All message types for connection termination defined in the Infiniband standard are also implemented on the FPGA.

Data Transfer: Number of Memory Regions

The main goal of the API is to allow the offloading of tasks on to an NAA in an RPC-like fashion. For this, the data transfer should be as performant as possible. Taking message overhead into account it can be safely assumed that the performance of fewer and larger messages is the best. The actual style of the data transfer, i.e. the number and size of messages sent is dependent on many factors however:

- A single message can only hold 2 GB, which is also the maximum size for a memory region right now
- There is an overhead for multiple memory regions on the FPGA[1].
  - The management on the FPGA (e.g. mapping of virtual to physical addresses and the verification of the `rkeys`) is easier with a smaller number of memory regions
  - The resource cost (number of registers, amount of memory) for each supported queue pair scales with the number of regions.

  However, the performance of the FPGA with multiple DDR DIMMS (double data rate dual inline memory modules), will have a faster internal read/write performance if multiple memory regions on multiple DIMMS are used.
- From the user perspective, zero-copy data transfer is preferred (at least for large data chunks), i.e. each RPC parameter should have its own memory region.

At the start we have identified two distinct setups regarding memory regions: A symmetric and an asymmetric setup with one memory region on the FPGA. Other asymmetric memory region setups exist e.g. with two memory regions on the FPGA. However, these only represent intermediate stages between the two outlined border cases. The symmetric and asymmetric memory region setup with only one memory region on the FPGA are outlined in Figure 3 and 4. Currently, the symmetric approach is used. Large

parameters receive their own memory region to allow for zero-copy data transfer. However, smaller parameters will be collected in a single memory region, for which copying of data on the client side is justifiable. In a later stage of the project, different approaches will be compared to find the most performant approach.
To justify this setup, a closer look will be taken at the `ibverbs`-provided semantics of writing data. Data is written using `ibv_post_send()`, which takes the associated queue pair and a list of work requests as paramters. For a detailed look at the parameters the user is referred to the manpage of `ibv_post_send()`. Every work request includes (among other less important parameters) an opcode. For writing data with RDMA, either an `RDMA_WRITE` or `RDMA_WRITE_WITH_IMM` opcode is used. Additionally,

[1] According to Fraunhofer Heinrich-Hertz-Institute

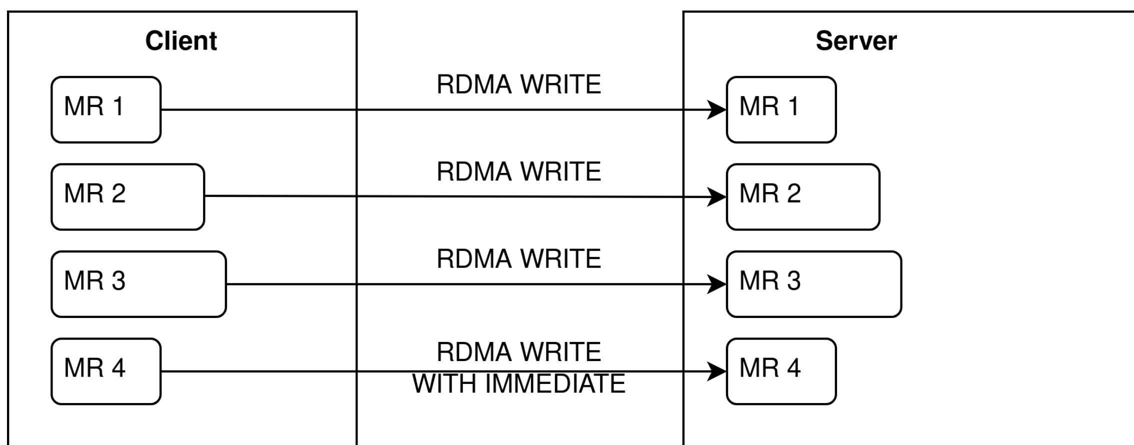


Figure 3: Schematic figure of symmetric memory region setup host-based client and an NAA-server. Both have the same amount of memory regions with equal sizes on both sides. The data is transferred using N operations for N memory regions.

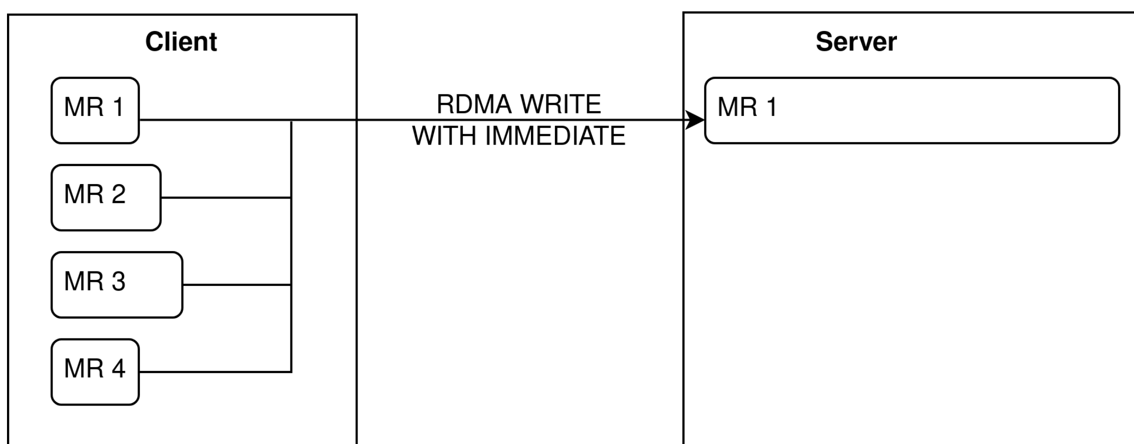


Figure 4: Schematic figure of asymmetric memory region setup host-based client and an NAA-server. The NAA only has one large memory region into which everything must be written. This can be done individually with a known offset using N operations for N memory regions or in 1 operation (up to 1 GB in total) using scatter-gather-elements.

each RDMA write request includes a remote address (to write to) and its accompanying `rkey`. Thus, the remote address has to be registered as part of a memory region the remote host as well. The memory to be written to the remote memory region is identified by a list of scatter-gather-elements (SGEs). Each SGE is made up of a starting address and its accompanying `lkey` and the amount of memory to write. Thus, memory to write to the remote host has to be registered as part of a memory region as well.
When using multiple SGEs, the data is written into contiguous memory on the remote side and interpreted as one single blob of memory, i.e. the remote is not aware of the fact that the data comes from different memory regions and might describe different parameters. Thus, when using SGEs, usually metadata has to be transferred as well for the remote side to distinguish the original meaning of the data. Additionally, Dotan Barak (Mellanox/NVIDIA), a main developer of `ibverbs` as part of the Linux Kernel, hints that using many SGEs is not performant[2]. When using multiple SGEs, the NIC collects the data from the different memory regions and transfers it to the remote side. It is yet unclear whether this leads to internal copying of data.
Lastly, first experiments suggest that using multiple write operations with multiple memory regions of 1 MB or larger provide optimal throughput [1]. To summarize, we use the symmetric approach for now, because:

- It is easier to implement
- The sizes of most parameters of the RPC can be inferred by the memory region size
- We don't expect large performance issues

[2]https://www.rdmamojo.com/2013/06/08/tips-and-tricks-to-optimize-your-rdma-code/

3 API

The documentation of the API includes the structures as well as all methods. The API specifies methods for a client and server. The client is always a host machine, requesting some RPC to be done on a remote machine. The remote part can be taken on by a host machine as well, for which we provide software, or any other machine such as an FPGA, which is compatible with the proposed API. The software for host clients and servers is modelled as a state machine.

State Machine

A state machine is used to facilitate communication as depicted in Figure 2. Our state machine is defined with the following states:

```
enum naaice_communication_state
{
    INIT              = 00,
    READY             = 01,
    CONNECTED         = 02,
    DISCONNECTED      = 03,
    MRSP_SENDING      = 10,
    MRSP_RECEIVING    = 11,
    MRSP_DONE         = 12,
    DATA_SENDING      = 20,
    CALCULATING       = 21,
    DATA_RECEIVING    = 22,
    FINISHED          = 30,
    ERROR             = 40,
};
```

An exemplary flow chart of the state machines for a client and server without errors are given in Figure 5. Except for `CALCULATING`, both communication partners run through all states.

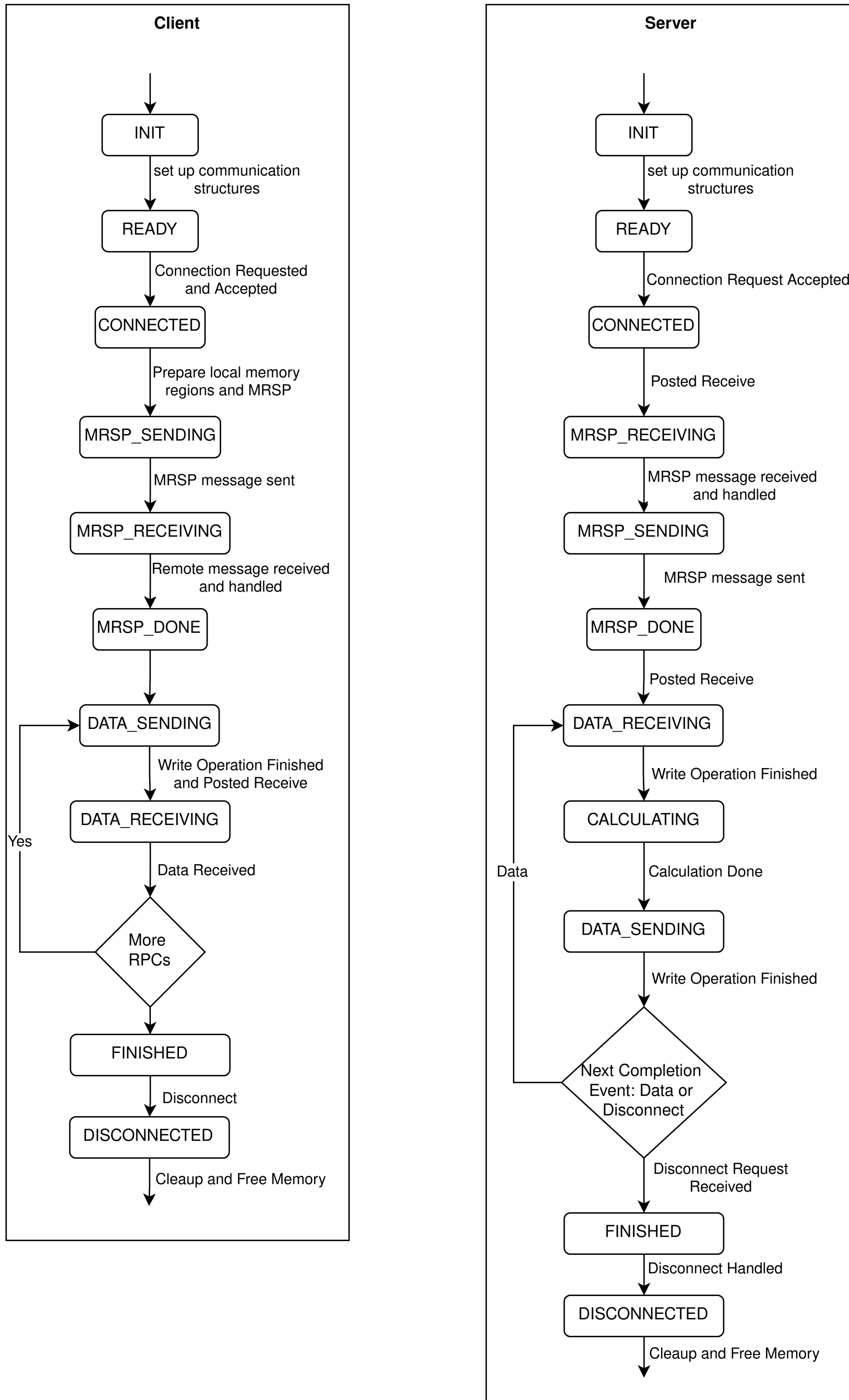


Figure 5: Flow chart of the state machine for client and server. An error-free communication with 1+ RPC invocations is depicted.

Structures

Structures for Handling Messages

```
struct naaice_mr_hdr{
  uint8_t type;
  };
struct naaice_mr_dynamic_hdr{
  uint8_t count;
  uint8_t padding[2];
};
struct naaice_mr_advertisement_request{
  uint8_t mrflags;
  uint8_t fpgaaddress[7];
  uint64_t addr;
  uint32_t rkey;
  uint32_t size;
};
struct naaice_mr_advertisement
{
  uint64_t addr;
  uint32_t rkey;
  uint32_t size;
};

struct naaice_mr_error{
    uint8_t code;
    uint8_t padding[2];
  };
```

Different structures for the parsing of messages exist. All defined messages have in common that the first 8 bit signify the message type. All messages but error messages then include 8 bits for a count variable and 16 bits for padding. An Advertisement and Request includes the five variables of metadata: 8 bits for MR flags, which signify the type of MR: regular or FPGA-only; 56 bits for a requested FPGA address, and the address (64), rkey (32) and length (32) of the advertised MR. Error messages simply consist of the error code and padding. The different structures are used in parsing messages according to the message type.

Memory Region Structures

```
struct naaice_mr_peer{
  uint64_t addr;
  uint32_t rkey;
  uint32_t size;
};

struct naaice_mr_local{
```

```
  struct ibv_mr *ibv;
  char *addr;
  size_t size;
  bool write;
};
struct naaice_mr_internal{
  char *addr;
  uint32_t size;
};
```

Three different structures for memory regions exist. Peer memory regions, i.e. those of the remote host are identified by the address, rkey and size, much like the advertisement message structure. Local memory region structures consist of a memory region structure from the `libibverbs` library and an address pointer and the size (also available via the ibv_mr member). The memory region structure `ibv_mr` also includes the local and remote key and size of the memory region, as well as the address. Local memory regions also have an associated bool variable that states, whether the given MR will be written in the next write operation. The different memory regions are available as an array that is part of the overall communication structure `naaice_communication_context`, which holds all information relevant for the communication. Internal memory regions are only available on the FPGA and are used for storing calculation results/temporary results.

```
struct naaice_communication_context
{
  // Basic connection properties.
  struct rdma_cm_id *id;                    // Communication ID.
  struct rdma_event_channel *ev_channel;    // Event channel.
  struct ibv_context *ibv_ctx;              // IBV context.
  struct ibv_pd *pd;                        // Protection domain.
  struct ibv_comp_channel *comp_channel;    // Completion channel.
  struct ibv_cq *cq;                        // Completion queue.
  struct ibv_qp *qp;                        // Queue pair.

  // Current state.
  enum naaice_communication_state state;

  // Local memory regions.
  struct naaice_mr_local *mr_local_data;
  uint8_t no_local_mrs;

  // Index indicating which local memory region is the return
     region.
  // Set when the return address is set, in naaice_set_metadata.
  // Should be in range [1, no_local_mrs].
  uint8_t mr_return_idx;

  // Array of peer memory regions, i.e. information about
     memory regions of the
  // commuication partner.
```

```
    // Includes only symmetric memory regions, i.e. only MRs
       representing parameters
    // and not internal MRs used on the NAA just for computation.
    struct naaice_mr_peer *mr_peer_data;
    uint8_t no_peer_mrs;

    // Used for MRSP.
    struct naaice_mr_local *mr_local_message;

    // Array of internal memory regions, i.e. information about
       memory regions
    // on the NAA used only for computation which are not
       communicated during
    // data transfer.
    struct naaice_mr_internal *mr_internal;
    uint8_t no_internal_mrs;

    // Function code indicating which NAA routine to be called.
    uint8_t fncode;

    // Keeps track of number of writes done to NAA.
    uint8_t rdma_writes_done;
};
enum naaice_communication_state
{
    INIT            = 00,
    READY           = 01,
    CONNECTED       = 02,
    DISCONNECTED    = 03,
    MRSP_SENDING    = 10,
    MRSP_RECEIVING  = 11,
    MRSP_DONE       = 12,
    DATA_SENDING    = 20,
    CALCULATING     = 21,
    DATA_RECEIVING  = 22,
    FINISHED        = 30,
    ERROR           = 40,
};
```

Additionally, all `ibverbs`-specific structures such as the queue pair, and work request queues are part of the communication context structure. Each communication partner is modeled as a state machine. This is done since the communication works with events that signal the current state of the connection, such as establishment of a connection or the completion of sending/writing operations. All unsigned integer members are used for internal bookkeeping.

3.1 Client API

Setup and Infiniband Work Completions

```
/**
* naaice_init_communication_context:
*  Initializes communication context struct.
*  After a call to this function, the provied communication
   context struct is
*  ready to be passed to all other AP1 functions.
*
* params:
*  naaice_communication_context **comm_ctx: (return param)
*    Double pointer to communication context struct to be
   initialized.
*    Should not point to an existing struct; the struct is
   allocated
*    and returned by this function.
*  unsigned int *param_sizes:
*    Array of sizes (in bytes) of the provided routine
   parameters.
*  char **params:
*    Array of pointers to parameter data. Should be
   preallocated by
*    the host application.
*  unsigned int params_amount:
*    Number of params. Used to index param_sizes and params, so
   their lengths
*    should correspond to params_amount.
*  uint8_t fncode:
*    Function code specifying which NAA routine to be called.
*  const char *remote_ip:
*    String specifying remote address, ex. "10.3.10.135".
*  uint16_t port:
*    Value specifying connection port, ex. 12345.
*
* returns:
*  0 if sucessful, -1 if not.
*/

/**
* naaice_handle_work_completion:
*  Handles a single work completion from the completion queue.
*  These represent memory region writes from host to NAA or NAA
   to host.
*
* params:
*  naaice_communication_context *comm_ctx:
*    Pointer to struct describing the connection.
*
```

```
* returns:
*  0 if sucessful, -1 if not.
*/
int naaice_handle_work_completion(struct ibv_wc *wc,
 struct naaice_communication_context *comm_ctx);

/**
* naaice_poll_cq_nonblocking:
*  Polls the completion queue for any work completions, and
   handles them if
*  any are recieved using naaice_handle_work_completion.
*
*  Subsequently, comm_ctx->state is updated to reflect the
   current state
*  of the NAA connection and routine.
*
* params:
*  naaice_communication_context *comm_ctx:
*    Pointer to struct describing the connection.
*
* returns:
*  0 if sucessful (regardless of whether any work completions
   are recieved),
* -1 if not.
*/
int naaice_poll_cq_nonblocking(struct
   naaice_communication_context *comm_ctx);

/**
* naaice_poll_cq_blocking:
*  Polls the completion queue for any work completion once and
   blocks until any completion
*  is available. Then it handles them using
   naaice_handle_work_completion.
*
*  Subsequently, comm_ctx->state is updated to reflect the
   current state
*  of the NAA connection and routine.
*
* params:
*  naaice_communication_context *comm_ctx:
*    Pointer to struct describing the connection.
*
* returns:
*  0 if sucessful (regardless of whether any work completions
   are recieved),
* -1 if not.
*/
int naaice_poll_cq_blocking(struct naaice_communication_context
```

```
    *comm_ctx);
/**
* naaice_init_rdma_resources
*  Allocates a protection domain, completion channel,
    completion queue, and
*  queue pair.
*
* params:
*  naaice_communication_context *comm_ctx:
*    Pointer to struct describing the connection.
*
* returns:
*  0 if successful, -1 if not.
*/
int naaice_init_rdma_resources(struct
    naaice_communication_context *comm_ctx);
```

Connection Management

```
/**
* naaice_poll_connection_event:
*  Polls for a connection event on the RDMA event channel
    stored in the
*  communication context. If a connection event is recieved,
    stores it
*  in the provided event pointer.
*
* params:
*  naaice_communication_context *comm_ctx:
*    Pointer to struct describing the connection to be polled.
*  struct rdma_cm_event ev: (return param)
*    Pointer to event recieved.
*
* returns:
*  0 if sucessful (regardless of whether an event is recieved),
    -1 if not.
*/
int naaice_poll_connection_event(struct
    naaice_communication_context *comm_ctx,
                                 struct rdma_cm_event *ev,
                                 struct rdma_cm_event *ev_cp);

/**
* connection event handlers:
*  These functions each handle a specific connection event. If
    the provided
*  event's type matches the event type of the handler function,
    it executes
```

```
*  the necessary logic to handle it.
*
*  Subsequently, flags in the communication context are updated
   to represent
*  the current status of connection establishment.
*
*  The events handled by these functions are, in order:
*  RDMA_CM_EVENT_ADDR_RESOLVED
*  RDMA_CM_EVENT_ROUTE_RESOLVED
*  RDMA_CM_EVENT_CONNECT_ESTABLISHED
*
*  Also, the following are handled by naaice_handle_error:
*  RDMA_CM_EVENT_ADDR_ERROR, RDMA_CM_EVENT_ROUTE_ERROR,
*  RDMA_CM_EVENT_CONNECT_ERROR, RDMA_CM_EVENT_UNREACHABLE,
*  RDMA_CM_EVENT_REJECTED, RDMA_CM_EVENT_DEVICE_REMOVAL,
*  RDMA_CM_EVENT_DISCONNECTED.
*
* params:
*  naaice_communication_context *comm_ctx:
*    Pointer to struct describing the connection with events to
   handle.
*  struct rdma_cm_event *ev:
*    Pointer to event to be checked and possibly handled.
*
* returns:
*  0 if sucessful (i.e. either the event was the matching type
   and was handled
*  successfully, or the event was not the matching type), -1 if
   not.
*/
int naaice_handle_addr_resolved(
  struct naaice_communication_context *comm_ctx,
  struct rdma_cm_event *ev);
int naaice_handle_route_resolved(
  struct naaice_communication_context *comm_ctx,
  struct rdma_cm_event *ev);
int naaice_handle_connection_established(
  struct naaice_communication_context *comm_ctx,
  struct rdma_cm_event *ev);
int naaice_handle_error(
  struct naaice_communication_context *comm_ctx,
  struct rdma_cm_event *ev);
int naaice_handle_other(
  struct naaice_communication_context *comm_ctx,
  struct rdma_cm_event *ev);

/**
* naaice_poll_and_handle_connection_event:
*  Polls for a connection event on the RDMA event channel
```

```
   stored in the
*  communication context and handles the event if one is
   recieved.
*  Simply uses the poll and handle functions above.
*
* params:
*  naaice_communication_context *comm_ctx:
*    Pointer to struct describing the connection.
*
* returns:
*  0 if sucessful (regardless of whether an event is recieved),
   -1 if not.
*/
int naaice_poll_and_handle_connection_event(
  struct naaice_communication_context *comm_ctx);

/**
* naaice_setup_connection:
*  Loops polling for and handling connection events until
   connection setup
*  is complete. Simply uses
   naaice_poll_and_handle_connection_event.
*
*
* params:
*  naaice_communication_context *comm_ctx:
*    Pointer to struct describing the connection.
*
* returns:
*  0 if sucessful, -1 if not (due to timeout).
*/
int naaice_setup_connection(struct naaice_communication_context
   *comm_ctx);
/**
* naaice_disconnect_and_cleanup:
*  Terminates the connection and frees all communication
   context memory.
*
* params:
*  naaice_communication_context *comm_ctx:
*    Pointer to struct describing the connection.
*
* returns:
*  0 if sucessful, -1 if not.
*/
int naaice_disconnect_and_cleanup(
  struct naaice_communication_context *comm_ctx);
```

Memory Region Exchange

```
/**
* naaice_register_mrs:
*  Registers local memory regions as IBV memory regions using
   ibv_reg_mr.
*  This includes memory regions corresponding to input and
   output params,
*  the single metadata memory region, and the single memory
   region used
*  for MRSP.
*  If an error occurs, the remote peer is notified via
   naaice_send_message.
*
* params:
*  naaice_communication_context *comm_ctx:
*    Pointer to struct describing the connection and memory
   regions.
*
* returns:
*  0 if sucessful, -1 if not.
*/
int naaice_register_mrs(struct naaice_communication_context
   *comm_ctx);

/**
* naaice_set_metadata:
*  Sets the fields of the metadata memory region.
*  Specifically, this sets the return_addr field, which
   specifies which
*  memory region the NAA should write results back to.
*
*  Should only be called once. Each call to this function
   overwrites the
*  previous call.
*
*
* params:
*  naaice_communication_context *comm_ctx:
*    Pointer to struct describing the connection.
*  uintptr_t return_addr:
*    Address of memory region to be used as the return param
   for the RPC.
*
* returns:
*  0 if sucessful, -1 if not.
*/
int naaice_set_metadata(struct naaice_communication_context
   *comm_ctx,
```

```
    uintptr_t return_addr);

/**
* naaice_set_internal_mrs
*  Adds information about internal memory regions to the
   communication
*  context. Such memory regions exist only on the NAA side and
   are used for
*  computation. The contents of these memory region are not
   communicated
*  during data transfer.
*
*  Must be called before naaice_init_mrsp. The internal memory
   regions will
*  then be included in the memory region announcement message,
   indicating that
*  they should be allocated by the NAA.
*
*  Should only be called once. Each call to this function
   overwrites the
*  previous call.
*
* params:
*  naaice_communication_context *comm_ctx:
*    Pointer to struct describing the connection.
*  unsigned int n_internal_mrs:
*    Number of internal memory regions.
*  uintptr_t *addrs:
*    Array of addresses of the internal memory regions in NAA
   memory space.
*    These addresses will be requested of the NAA during MRSP.
*  uint32_t *sizes:
*    Sizes of the internal memory region, in bytes.
*
* returns:
*  0 if sucessful, -1 if not.
*/
int naaice_set_internal_mrs(struct naaice_communication_context
   *comm_ctx,
  unsigned int n_internal_mrs, uintptr_t *addrs, size_t *sizes);

/**
* naaice_init_mrsp:
*  Starts the MRSP. That is, sends advertise/request packets
   and posts a
*  recieve for the response.
*
* params:
*  naaice_communication_context *comm_ctx:
```

```
*     Pointer to struct describing the connection and memory
   regions.
*
* returns:
*  0 if sucessful, -1 if not.
*/
int naaice_init_mrsp(struct naaice_communication_context
   *comm_ctx);

/**
* naaice_send_message:
*  Sends an MRSP packet to the remote peer. Done with a
   ibv_post_send using
*  opcode IBV_WR_SEND.
*
* params:
*  naaice_communication_context *comm_ctx:
*     Pointer to struct describing the connection.
*  enum message_id message_type:
*     Specifies the packet type. Should be one of MSG_MR_ERR,
   MSG_MR_AAR,
*     or MSG_MR_A.
*  uint8_t errorcode:
*     Error code send in the packet, if message_type was
   MSG_MR_ERR.
*     Unused for other message types.
*
* returns:
*  0 if sucessful, -1 if not.
*/
int naaice_send_message(struct naaice_communication_context
   *comm_ctx,
 enum message_id message_type, uint8_t errorcode);

/**
* naaice_post_recv_mrsp
*  Posts a recieve for an MRSP message.
*  A recieve request is added to the queue which specifies the
   memory region
*  to be written to (in this case, the MRSP region).
*
* params:
*  naaice_communication_context *comm_ctx:
*     Pointer to struct describing the connection.
*
* returns:
*  0 if successful, -1 if not.
*/
int naaice_post_recv_mrsp(struct naaice_communication_context
```

```
    *comm_ctx);

/**
* naaice_do_mrsp
*  Does all logic for the MRSP in a blocking fashion.
*
* params:
*  naaice_communication_context *comm_ctx:
*    Pointer to struct describing the connection.
*
* returns:
*  0 if successful, -1 if not.
*/
int naaice_do_mrsp(struct naaice_communication_context
    *comm_ctx);
```

Data Transfer

```
/**
* naaice_init_data_transfer:
*  Starts the data transfer. That is, posts the write for
    memory regions to
*  the NAA.
*
* params:
*  naaice_communication_context *comm_ctx:
*    Pointer to struct describing the connection and memory
    regions.
*
* returns:
*  0 if sucessful, -1 if not.
*/
int naaice_init_data_transfer(struct
    naaice_communication_context *comm_ctx);

/**
* naaice_write_data:
*  Writes memory regions (metadata and input parameters) to the
    NAA. Done
*  with a ibv_post_send using opcode IBV_WR_RDMA_WRITE or
*  IBV_WR_RDMA_WRITE_WITH_IMM (for the final memory region).
*
* params:
*  naaice_communication_context *comm_ctx:
*    Pointer to struct describing the connection.
*  uint8_t fncode:
*    Function Code for NAA routine. Positive, 0 on error.
```

```
*
* returns:
*  0 if sucessful, -1 if not.
*/
int naaice_write_data(struct naaice_communication_context
   *comm_ctx,
 uint8_t fncode);

/**
* naaice_post_recv_data
*  Posts a recieve for a memory region write.
*  It is only necessary to post a recieve for the final memory
   region to be
*  written, that is, the write with an immediate value. RDMA
   writes without
*  an immediate simply occur without consuming a recieve
   request in the queue.
*
*  The memory region specified in the recieve request is the
   MRSP region;
*  this is just a dummy value, as the region written to is
   specified by the
*  sender.
*
* params:
*  naaice_communication_context *comm_ctx:
*    Pointer to struct describing the connection.
*
* returns:
*  0 if successful, -1 if not.
*/
int naaice_post_recv_data(struct naaice_communication_context
   *comm_ctx);

/**
* naaice_do_data_transfer
*  Does all logic for the data transfer, including writing data
   to the NAA,
*  wating for the NAA calculation, and receiving the return
   data back, in a
*  blocking fashion.
*
* params:
*  naaice_communication_context *comm_ctx:
*    Pointer to struct describing the connection.
*
* returns:
*  0 if successful, -1 if not.
*/
```

```c
int naaice_do_data_transfer(struct naaice_communication_context
    *comm_ctx);
```

3.2 Software-NAA API

Setup and Infiniband Work Completions

```c
/**
* naaice_init_communication_context:
*  Initializes communication context struct.
*
*  The dummy software NAA reuses the communication context
   struct from the
*  host-side API implementation, but doesn't use all the fields
   in the
*  exact same way. Most importantly, the size and number of
   parameters
*  is not known (and related fields are not populated) until
   after the
*  MRSP is complete.
*
* params:
*  naaice_communication_context *comm_ctx: (return param)
*    Pointer to communication context struct to be initialized.
*    Should not point to an existing struct; the struct is
   allocated
*    and returned by this function.
*  const char *port:
*    String specifying connection port, ex. "12345".
*
* returns:
*  0 if sucessful, -1 if not.
*/
int naaice_swnaa_init_communication_context(
  struct naaice_communication_context **comm_ctx, uint16_t
     port);
/**
* naaice_swnaa_handle_work_completion:
*  Handles a single work completion from the completion queue.
*  These represent memory region writes from host to NAA or NAA
   to host.
*
* params:
*  naaice_communication_context *comm_ctx:
*    Pointer to struct describing the connection.
*
* returns:
*  0 if sucessful, -1 if not.
```

```
*/
int naaice_swnaa_handle_work_completion(struct ibv_wc *wc,
  struct naaice_communication_context *comm_ctx);
/**
* naaice_swnaa_poll_cq_nonblocking:
*  Polls the completion queue for any work completions, and
   handles them if
*  any are received using naaice_handle_work_completion.
*
*  Subsequently, comm_ctx->state is updated to reflect the
   current state
*  of the NAA connection and routine.
*
* params:
*  naaice_communication_context *comm_ctx:
*    Pointer to struct describing the connection.
*
* returns:
*  0 if sucessful (regardless of whether any work completions
   are received),
* -1 if not.
*/
int naaice_swnaa_poll_cq_nonblocking(
  struct naaice_communication_context *comm_ctx);
```

Connection Management

```
/**
* naaice_swnaa_setup_connection:
*  Loops polling for and handling connection events until
   connection setup
*  is complete. Unlike the base naaice version, does not
   require handling
*  the address or route resolution events, but does handle the
   connection
*  requests complete event which is not handled on the host
   side.
*
* params:
*  naaice_communication_context *comm_ctx:
*    Pointer to struct describing the connection.
*
* returns:
*  0 if sucessful, -1 if not (due to timeout).
*/
int naaice_swnaa_setup_connection(
  struct naaice_communication_context *comm_ctx);
/**
```

```
* swnaa connection event handlers:
*  These functions each handle a specific connection event. If
   the provided
*  event's type matches the event type of the handler function,
   it executes
*  the necessary logic to handle it.
*
*  Subsequently, flags in the communication context are updated
   to represent
*  the current status of connection establishment.
*
*  The events handled by these functions are, in order:
*  RDMA_CM_EVENT_CONNECTION_REQUEST
*  RDMA_CM_EVENT_CONNECT_ESTABLISHED
*
*  Also, the following are handled by naaice_handle_error:
*  RDMA_CM_EVENT_ADDR_ERROR, RDMA_CM_EVENT_ROUTE_ERROR,
*  RDMA_CM_EVENT_CONNECT_ERROR, RDMA_CM_EVENT_UNREACHABLE,
*  RDMA_CM_EVENT_REJECTED, RDMA_CM_EVENT_DEVICE_REMOVAL,
*  RDMA_CM_EVENT_DISCONNECTED.
*
* params:
*  naaice_communication_context *comm_ctx:
*    Pointer to struct describing the connection with events to
   handle.
*  struct rdma_cm_event *ev:
*    Pointer to event to be checked and possibly handled.
*
* returns:
*  0 if sucessful (i.e. either the event was the matching type
   and was handled
*  successfully, or the event was not the matching type), -1 if
   not.
*/
int naaice_swnaa_handle_connection_requests(struct
   naaice_communication_context *comm_ctx,
                                   struct rdma_cm_event *ev);
int naaice_swnaa_handle_connection_established(
    struct naaice_communication_context *comm_ctx, struct
       rdma_cm_event *ev);

/**
* naaice_swnaa_poll_and_handle_connection_event:
*  Polls for a connection event on the RDMA event channel
   stored in the
*  communication context and handles the event if one is
   received.
*  Simply uses the poll and handle functions above.
*
```

```
* params:
*  naaice_communication_context *comm_ctx:
*    Pointer to struct describing the connection.
*
* returns:
*  0 if sucessful (regardless of whether an event is received),
   -1 if not.
*/
int naaice_swnaa_poll_and_handle_connection_event(
    struct naaice_communication_context *comm_ctx);

/**
* naaice_swnaa_disconnect_and_cleanup:
*  Terminates the connection and frees all communication
   context memory.
*
* params:
*  naaice_communication_context *comm_ctx:
*    Pointer to struct describing the connection.
*
* returns:
*  0 if sucessful, -1 if not.
*/
int naaice_swnaa_disconnect_and_cleanup(
 struct naaice_communication_context *comm_ctx);
```

Memory Region Exchange

```
/**
* naaice_swnaa_init_mrsp:
*  Starts the MRSP on the NAA side. That is, posts a recieve
   for MRSP
*  packets expected from the host.
*
* params:
*  naaice_communication_context *comm_ctx:
*    Pointer to struct describing the connection and memory
   regions.
*
* returns:
*  0 if sucessful, -1 if not.
*/
int naaice_swnaa_init_mrsp(struct naaice_communication_context
   *comm_ctx);

/**
* naaice_swnaa_post_recv_mrsp
```

```
*  Posts a recieve for an MRSP message.
*  A recieve request is added to the queue which specifies the
   memory region
*  to be written to (in this case, the MRSP region).
*
* params:
*  naaice_communication_context *comm_ctx:
*     Pointer to struct describing the connection.
*
* returns:
*  0 if successful, -1 if not.
*/
int naaice_swnaa_post_recv_mrsp(struct
   naaice_communication_context *comm_ctx);
/**
* naaice_swnaa_handle_mr_announce,
* naaice_swnaa_handle_mr_announce_and_request:
*
*  Handlers for MRSP packets.
*  Processes the contents of a received MRSP packet of the
   corresponding type,
*  populating relevant values in the communication context.
*
* params:
*  naaice_communication_context *comm_ctx:
*     Pointer to struct describing the connection.
*
* returns:
*  0 if sucessful, -1 if not.
*/
int naaice_swnaa_handle_mr_announce(
  struct naaice_communication_context *comm_ctx);
int naaice_swnaa_handle_mr_announce_and_request(
  struct naaice_communication_context *comm_ctx);

/**
* naaice_swnaa_send_message:
*  Sends an MRSP packet to the remote peer. Done with a
   ibv_post_send using
*  opcode IBV_WR_SEND.
*
* params:
*  naaice_communication_context *comm_ctx:
*     Pointer to struct describing the connection.
*  enum message_id message_type:
*     Specifies the packet type. Should be one of MSG_MR_ERR,
   MSG_MR_AAR,
*     or MSG_MR_A.
*  uint8_t errorcode:
```

```
*     Error code send in the packet, if message_type was
   MSG_MR_ERR.
*     Unused for other message types.
*
* returns:
*   0 if sucessful, -1 if not.
*/
int naaice_swnaa_send_message(struct
   naaice_communication_context *comm_ctx,
  enum message_id message_type, uint8_t errorcode);/**
* naaice_swnaa_do_mrsp
*   Does all logic for the MRSP in a blocking fashion.
*
* params:
*   naaice_communication_context *comm_ctx:
*     Pointer to struct describing the connection.
*
* returns:
*   0 if successful, -1 if not.
*/
int naaice_swnaa_do_mrsp(struct naaice_communication_context
   *comm_ctx);
/**
* naaice_swnaa_do_mrsp
*   Does all logic for the MRSP in a blocking fashion.
*
* params:
*   naaice_communication_context *comm_ctx:
*     Pointer to struct describing the connection.
*
* returns:
*   0 if successful, -1 if not.
*/
int naaice_swnaa_do_mrsp(struct naaice_communication_context
   *comm_ctx);
```

Data Transfer

```
/**
* naaice_swnaa_post_recv_data
*   Posts a recieve for a memory region write.
*   It is only necessary to post a recieve for the final memory
    region to be
*   written, that is, the write with an immediate value. RDMA
    writes without
*   an immediate simply occur without consuming a recieve
    request in the queue.
*
```

```
*  The memory region specified in the recieve request is the
   MRSP region;
*  this is just a dummy value, as the region written to is
   specified by the
*  sender.
*
* params:
*  naaice_communication_context *comm_ctx:
*    Pointer to struct describing the connection.
*
* returns:
*  0 if successful, -1 if not.
*/
int naaice_swnaa_post_recv_data(struct
   naaice_communication_context *comm_ctx);

/**
* naaice_swnaa_handle_metadata
*  Updates information in the communication context based on
   received
*  metadata, which before this call should be available in the
   local metadata
*  memory region.
*
* params:
*  naaice_communication_context *comm_ctx:
*    Pointer to struct describing the connection.
*
* returns:
*  0 if successful, -1 if not.
*/
int naaice_swnaa_handle_metadata(
  struct naaice_communication_context *comm_ctx);

/**
* naaice_swnaa_write_data:
*  Writes the return memory region, specified by
   comm_ctx->mr_return_idx, to
*  the remote peer. Done with a ibv_post_send using opcode
*  IBV_WR_RDMA_WRITE_WITH_IMM. The immediate value indicates if
   an error has
*  occured during calculation (nonzero = error).
*
* params:
*  naaice_communication_context *comm_ctx:
*    Pointer to struct describing the connection.
*  uint8_t fncode:
*    Function Code for NAA routine. Positive, 0 on error.
*
```

```
* returns:
*  0 if sucessful, -1 if not.
*/
int naaice_swnaa_write_data(
  struct naaice_communication_context *comm_ctx, uint8_t
     errorcode);
/**
* naaice_swnaa_receive_data_transfer
*  Handles recieving data from remote peer, blocking until this
   is finished.
*  Information about the data is updated in the communication
   context.
*
* params:
*  naaice_communication_context *comm_ctx:
*    Pointer to struct describing the connection.
*
* returns:
*  0 if successful, -1 if not.
*/
int naaice_swnaa_receive_data_transfer(
  struct naaice_communication_context *comm_ctx);
*
* Starts the data transfer to the client. That is, posts the
   write for return memory region.
*
* params:
*  naaice_communication_context *comm_ctx:
*    Pointer to struct describing the connection and memory
   regions.
*  uint8_t fncode:
*    error code returned by NAA routine. 0, positive on error.
*
* returns:
*  0 if sucessful, -1 if not.
*/
int naaice_swnaa_write_data_transfer(
    struct naaice_communication_context *comm_ctx, uint8_t
       errorcode);
```

References

[1] Steffen Christgau, Dylan Everingham, Florian Mikolajczak, Niklas Schelten, Bettina Schnor, Max Schroetter, Benno Stabernack, and Fritjof Steinert. Enabling communication with fpga-based network-attached accelerators for hpc workloads. In *Proceedings of the SC '23 Workshops of The International Conference on High Performance*

Computing, Network, Storage, and Analysis, SC-W '23, page 530–538, New York, NY, USA, 2023. Association for Computing Machinery.

[2] Infiniband Trade Association. *InfiniBand Architecture Specification Volume 1 Release 1.2.1*, 2007.

[3] InfiniBand Trade Association. *InfiniBand Architecture Specification Volume 1, Release 1.6*. InfiniBand Trade Association, July 2022.

[4] Stuart Sechrest. An introductory 4.4 bsd interprocess communication tutorial. *Computer Science Research Group, Department of Electrical Engineering and Computer Science, University of California, Berkeley*, 1986.