NAAICE Middleware API Documentation

Florian Mikolajczak, Dylan Everingham, Hannes Signer
NAAICE AP2
University of Potsdam, Zuse-Institute Berlin

September 4, 2025

Contents

1 Introduction & Motivations

This document describes the API for the NAAICE Middleware which integrates a network-attached accelerator (NAA) into HPC data centers using RoCEv2, allowing for IP-based remote direct memory accesses (RDMA). In order for this to work, a middleware/communication library was designed to achieve the following goals:

1. **Easy integration into HPC applications**
   The communication details should be transparent to the user, since an application developer should not deal with communication specifics, but rather concentrate on the specification of the computation which should be offloaded to the NAA. Communication details include for example a communication context containing `ibverbs`-specific structures such as Infiniband queue pairs or details of memory region handling.

2. **Fast adaption by the HPC community**
   The middleware library should be easily understood and used by the HPC community. As such, the popular and well known Message Passing Interface (MPI) standard will be taken as an inspiration to formulate an NAA middleware, which will reproduce or reuse functionalities and structures of the MPI standards. We assume that a middleware with analogies to MPI can be more easily adopted by the HPC community.

3. **Ability for communication-computation-overlap (CCO)**
   Instead of serialized communication and computation, the aim of the middleware is to allow both to happen concurrently. For this, non-blocking communication calls are necessary. Non-blocking communication functions return before the actual communication i.e. data-transfer is done. The host node can then continue with some other computation while the NIC continues with the data transfer.

2 Middleware API

The middleware consists of only five well-defined functions, which are presented in the following together with the corresponding data types.

2.1 Datatypes & Structs

Different data structures are used to manage connection establishment, connection status, and the memory regions to be exchanged.

Connection Management

```
typedef struct naa_handle {
    unsigned int function_code;
    struct naaice_communication_context *comm_ctx;
} naa_handle;
```

The `naa_handle` is the central data structure for managing the communication context of the connection. It also contains the function code informing the NAA which kernel to execute.

```
typedef struct naa_status {
        enum naaice_communication_state state;
        enum naa_error naa_error;
        uint32_t bytes_received;
} naa_status;
```

The `naa_status` structure contains all status information relating to the connection and RPC calls. The connection between the host and NAA is modelled as a state machine, with the current state stored in the `state` variable. Information about the status of an RPC call is stored in `naa_error`. For additional information, the `bytes_received` variable records the number of bytes sent back to the host by the NAA.

Memory Region Management

```
typedef struct naa_param_t {
        void *addr;
        size_t size;
        bool single_send;
} naa_param_t;

```

Data for offloading is stored in memory regions (MRs), which need to be contiguous memory for RDMA operations. These MRs are managed in the `naa_param_t` structure and are basically defined by the start address and size of the memory region. In addition, the `single_send` parameter can be set to true, which controls whether an MR is only transferred on the first call. This is useful for configuration data, for example. By default, this option is set to false, so that an MR is transferred with every RPC call of a connection.

Error Codes

```
enum naa_error {
  NAA_SUCCESS = 0x00,
  SOCKET_UNAVAIL = 0x01,
  KERNEL_TIMEOUT = 0x02
}
```

Currently, three values are defined in `naa_error`. In addition to `NAA_SUCCESS` (`0x00`) for an error-free RPC call, a distinction is made between two error cases. `SOCKET_UNAVAIL` (`0x01`) indicates an error on the side of the NAA's RoCe stack and specifies that the kernel requested in the function code is not available. The second error is indicated by `KERNEL_TIMEOUT` (`0x02`) and is returned if the NAA kernel does not terminate after a user-defined time. In addition to the current values, the values from `0x03` to `0x0f` are reserved for future errors, while the values from `0x10` to `0x7f` are reserved for application-specific errors.

2.2 Function Calls

All routines return an integer of type `naa_err`. Successful routines return 0 or a positive integer on failure, indicating the failure reason.

1. **Connection Establishment**

```
naa_create(function_code, *input_params, input_amount, *output_params,
               output_amount, *handle)

   IN   function_code   functionality to be executed on
                        the receiver side
   IN   *input_params   array of type naa_params_t for the
                        input memory regions
   IN   input_amount    number of input memory regions
   OUT  *output_params  array of type naa_params_t for the
                        output memory regions
   IN   output_amount   number of output memory regions
   IN   *handle         pointer to naa_handle for communication management
```

```
int naa_create(unsigned int function_code, naa_param_t *input_params,
               unsigned int input_amount, naa_param_t *output_params,
               unsigned int output_amount, naa_handle *handle);
```

- Finding IP address and socket ID for an NAA matching required function code. Prepare connection, register and exchange memory region information between HPC node and NAA.
- IP address and socket ID are already known to HPC node. Info is retrieved from resource management system (Slurm) at creation/deployment of slurm job. User knows function code for method/calculation to outsource to NAA. Connection to NAA is done by connection establishment protocol from the Infiniband standard. During connection preparation, the HPC nodes allocates buffers for memory regions and resolves route to NAA. After connection establishment, memory region

information is exchanged between HPC node and NAA. The protocol for this was designed in NAAICE AP1.

- This method will register the addresses of the parameters with ibverbs as memory regions, hiding the memory region semantic from the user. All memory regions, for both input and output parameters are announced to the NAA during `naa_create()`. Therefore, memory regions can not be changed from input to output between iterations. Currently, no example has been found where this is necessary. The handle object was previously returned by the library and includes information on how to connect to the right NAA. The resource management system will provide information on the IP of the NAA and socket ID of the NAA.

2. **Sending Data**

```
naa_invoke(*handle)
   IN   *handle   pointer to naa_handle for communication management
```

`naa_invoke(naa_handle *handle)`

- Write data (in 1 or more RDMA operations) to NAA and trigger RPC-like behavior.
- Application programmer and NAA programmer have to agree on the size and meaning of each memory region beforehand. For now, the syntax and semantics of the transferred data is not explicitly stated.
- Data transfer is done with `RDMA_WITH_IMM`. If the transfer requires $n > 1$ operations, $n-1$ `RDMA_WRITE` operations are done. The last writing operation is `RDMA_WITH_IMM`, where the immediate data value is the function code. `RDMA_WITH_IMM` signals the end of the data transfer to the NAA and initiates calculations on the NAA (RPC start).

3. **Receiving Data**

```
naa_wait(*handle, *status);
   IN   *handle   pointer to naa_handle for communication management
   OUT  *status   result of operation is stored in naa_status
```

`int naa_wait(naa_handle *handle, naa_status *status)`

- Check if result data has been written to HPC node.
- Much like `MPI_WAIT`, the `naa_wait` call is blocking and polls the completion queue of the queue pair associated with the data transfer. `naa_wait` returns, when data has been written back to the HPC node.
- The call returns with the information on the completed operation stored in the `status` variable.

```
naa_test(*handle, *flag, *status)
   IN   *handle   pointer to naa_handle for communication management
   OUT  *flag     indicated whether operation is finished
   OUT  *status   result of operation is stored in naa_status
```

`int naa_test(naa_handle *handle, bool *flag, naa_status *status)`

- Check if result data has been written to HPC node.
- Much like `MPI_TEST`, the `naa_test` call is non-blocking and polls the completion queue of the queue pair associated with the data transfer.
- A call to `naa_test` returns `flag=true` if the operation identified by handle is complete. In such a case, the status object is set to contain information on the completed operation. The call returns flag = false if the operation is not complete. In this case, the value of the status object is undefined.

4. **Connection Termination**

```
naa_finalize(naa_handle *handle)
   IN   *handle   pointer to naa_handle for communication management
```

`int naa_finalize(naa_handle *handle)`

- Disconnect in an orderly fashion from the NAA and clean up allocated resources.
- In `naa_finalize`, the connection between HPC node and NAA is cleaned up as standardized in the Infiniband specification. In addition, resources that were allocated for and during the data transfer are cleaned up. `naa_finalize` is not called after each iteration of a given simulation/job, but only once, when the program terminates and the NAA is no longer needed.

3 Example Program

This sample code snippet shows the offloading of vector addition from the client's perspective. The logic of the addition is implemented accordingly on the NAA side. The call demonstrates both non-blocking waiting (1st RPC call) and blocked waiting (2nd RPC call).

```c
//All data is gathered and sent to the NAA in one go.
#include <naaice_ap2.h>
#define FNCODE_VEC_ADD 0t

void *a, *b, *c;
a = calloc(64, sizeof(double));
b = calloc(64, sizeof(double));
c = calloc(64, sizeof(double));

// define input and output memory regions
naa_param_t input_param[2] = {{a, 64 * sizeof(double)},
                              {b, 64 * sizeof(double)}};
naa_param_t output_param[1] = {{c, 64 * sizeof(double)}};

naa_handle handle;

// establish connection
naa_create(FNCODE_VEC_ADD, &input_params, 2, &output_params, 1, &handle) ;

int flag = 0;
naa_status status;

// transfer data to NAA
naa_invoke(&handle);

// non-blocking check if results are received
naa_test(&handle,&flag,&status)
while (!flag) {
  do_other_work();
  naa_test(&handle,&flag,&status)
}
process_results(c);

// set inputs with new data
set_inputs(a, 64, b, 64) ;

naa_invoke(&handle);

// blocked waiting for RPC to finish
naa_wait (&handle,&status)
process_results(c);

// finalize connection
naa_finalize(&handle);
```