## 3.8 Sharing Across Domains

Most of the MRAPI primitives are shared across MRAPI domains (Section 3.2) by default. We expect that implementations may potentially suffer a performance impact for resources that are shared across domains.

The following MRAPI primitives are shared across domains by default: mutexes, semaphores, reader/writer locks, and remote memory. For any of these primitives you can disable sharing across domains by setting the `MRAPI_DOMAIN_SHARED` attribute to `MRAPI_FALSE` and passing it to the corresponding `*_create()` function.

For the remaining primitive, e.g., MRAPI shared memory, the determination of which nodes it can be shared with (regardless of their domains) is specified in the nodes list that is passed in when the shared memory is created.

## 3.9 Waiting for Non-blocking Operations

The API has blocking, non-blocking, and single-attempt blocking variants for many functions. The non-blocking variants have "_i" appended to the function name to indicate that the function call will return *immediately* but the requested transaction will complete in a non-blocking manner. The single-attempt blocking functions will have the word "try" in the function name (for example, `mrapi_mutex_trylock`). Remote memory is the only resource that supports non-blocking variants (for reads/writes).

The non-blocking versions fill in an `mrapi_request_t` object and return control to the user before the requested operation is completed. The user can then use the `mrapi_test()`, `mrapi_wait()`, and `mrapi_wait_any()` functions to query the status of the non-blocking operation. The `mrapi_test()` function is non-blocking whereas the `mrapi_wait()` and `mrapi_wait_any()` functions will block until the requested operation completes or a timeout occurs.

Some blocking functions may have to wait for system events, e.g. buffer allocation or for data to arrive, and the duration of the blocking will be arbitrarily long (may be infinite), whereas other blocking functions don't have to wait for system events and can always complete in a timely fashion, with a success or failure. Single-attempt blocking functions that complete in this timely fashion include `mrapi_mutex_trylock()`, `mrapi_sem_trylock()`, `mrapi_rwl_trylock()`.

If a buffer of data is passed to a non-blocking operation (for example, to `mrapi_rmem_write_i()`) that buffer may not be accessed by the user application for the duration of the non-blocking operation. That is, once a buffer has been passed to a non-blocking operation, the program may not read or write the buffer until `mrapi_test()`, `mrapi_wait()`, or `mrapi_wait_any()` have indicated completion, or until `mrapi_cancel()` has canceled the operation.

## 3.10 MRAPI Error Handling Philosophy

Error handling is a fundamental part of the specification, however some accommodations have been made to allow for trading off completeness for efficiency of implementation. For example, some API functions allow implementations to *optionally* handle errors. Consistency and efficient coding styles also govern the design of the error handling. In general, function calls include an error code parameter used by the API function to indicate detailed status. In addition, the return value of several API functions indicate success or failure which enables efficient coding practice. A parameter of type `mrapi_status_t` will encode success or failure states of API calls. `MRAPI_NULL` is a valid return value for `mrapi_status_t` (can be used for implementation optimization).

If a process/thread attached to a node were to fail it is generally up to the application to recover from this failure. MRAPI provides timeouts for the `mrapi_wait()` and `mrapi_wait_any()` functions and an `mrapi_cancel()` function to clear outstanding non-blocking requests (at the non-failing side). It is also possible to reinitialize a failed node, by first calling `mrapi_finalize()`.