Paradyn Tools Project

MRNet API Programmer's Guide

Release 5.0.0
July 2015

MRNet Project
www.paradyn.org/mrnet
mrnet@cs.wisc.edu

Paradyn Tools Project
www.paradyn.org
paradyn@cs.wisc.edu

Computer Sciences Department
University of Wisconsin
Madison, WI 53706-1685

Table of Contents

1. INTRODUCTION

MRNet is a customizable, high-throughput communication software system for parallel tools and applications with a master/slave architecture. MRNet reduces the cost of these tools' activities by incorporating a tree-based overlay network (TBON) of processes between the tool's front-end and back-ends. MRNet uses the TBON to distribute many important tool communication and computation activities, reducing analysis time and keeping tool front-end loads manageable.

MRNet-based tools send data between front-end and back-ends on logical flows of data called streams. MRNet internal processes use filters to synchronize and aggregate data sent to the tool's front-end. Using filters to manipulate data in parallel as it passes through the network, MRNet can efficiently compute averages, sums, and other more complex aggregations on back-end data.

Several features make MRNet especially well-suited as a general facility for building scalable parallel tools:

- *Flexible organization*. MRNet does not dictate the organization of the TBON. MRNet process organization is specified in a configuration file that can specify common network overlays like k-ary and k-nomial trees, or custom layouts tailored to the system(s) running the tool. For example, MRNet internal processes can be allocated to dedicated system nodes or co-located with tool back-end and application processes.
- *Scalable, flexible data aggregation*. MRNet's built-in filters provide efficient computation of averages, sums, concatenation, and other common data reductions. Custom filters can be loaded dynamically into the network to perform tool-specific aggregation operations.
- *High-bandwidth communication.* MRNet transfers data within the tool system using an efficient, packed binary representation. Zero-copy data paths are used whenever possible to reduce the cost of transferring data through internal processes.
- *Scalable multicast.* As the number of back-ends increases, serialization when sending control requests limits the scalability of existing tools. MRNet supports efficient message multicast to reduce the cost of issuing control requests from the tool front-end to its back-ends.
- *Multiple concurrent data channels*. MRNet supports multiple logical streams of data between tool components. Data aggregation and message multicast takes place within the context of a data stream, and multiple operations (both upward and downward) can be active simultaneously.

2. ABSTRACTIONS

The MRNet distribution has two main components: `libmrnet`, a library that is linked into a tool's front-end and back-end components, and `mrnet_commnode`, a program that runs on intermediate nodes interposed between the application front-end and back-ends. `libmrnet` exports an API (see **"C++ API Reference" on page 8**) that enables I/O interaction between the front-end and groups of back-ends via MRNet. The primary purpose of `mrnet_commnode` is to distribute data processing functionality across multiple computer hosts and to implement efficient and scalable group communications. In addition, there is another component, `libmrnet_lightweight`, which exports an API (see **"C API Reference" on page 26**) that enables I/O interaction between the front-end and groups of "lightweight" back-ends via MRNet. Lightweight back-ends provide a pure C implementation of the MRNet API. They also do not support loading custom filters, and by default the API cannot be used by multiple threads concurrently. There is a separately built component, `libmrnet_lightweight_r`, which is thread-safe. The following sub-sections describe the lower-level components of the MRNet API in more detail.

2.1 End-Points

An MRNet end-point represents a tool or application process. Specifically, they represent the back-end processes (i.e., leaf processes) in the overlay tree. The front-end can communicate in a unicast or multicast fashion with these end-points as described below.

2.2 Communicators

MRNet uses communicators to represent groups of end-points. Like communicators in MPI, MRNet communicators provide a handle that identifies a set of end-points for point-to-point, multicast or broadcast communications. MPI applications typically have a non-hierarchical layout of potentially identical processes. In contrast, MRNet enforces a tree-like layout of all processes, rooted at the front-end. Accordingly, MRNet communicators are created and managed by the front-end, and communication is only allowed between a front-end and its back-ends. Thus, back-ends cannot interact with each other directly using the MRNet API.

2.3 Streams

A stream is a logical channel that connects the front-end to the end-points of a communicator. All MRNet communication uses the stream abstraction. Streams carry data packets downstream, from the front-end toward the back-ends, and upstream, from the back-ends toward the front-end. Streams are expected to carry data of a specific type, allowing data aggregation operations to be associated with a stream. The type is specified using a format string (see **Appendix E: "Format Strings" on page 43**) similar to those used in C formatted I/O primitives (e.g., a packet whose data is described by the format string "`%d %d %f %s`" contains two integers followed by a float then a character string). MRNet expands the standard format string specification to allow for description of arrays.

2.4 Filters

Data aggregation is the process of merging multiple input data packets and transforming them into one or more output packets. Though it is not necessary for the aggregation to result in less or even different data, aggregations that reduce or modify data values are most common. MRNet uses data filters to aggregate data packets. Filters specify an operation to perform and the type of the data expected on the bound stream. Filter instances are bound to a stream at stream creation. MRNet uses two types of filters: synchronization filters and transformation filters. Synchronization filters organize data packets from downstream nodes into synchronized waves of data packets, while transformation filters operate on the synchronized data packets yielding one or more output packets. A distinction between synchronization and transformation filters is that synchronization filters are independent of the packet data type, but transformation filters operate on packets of a specific type.

Synchronization filters operate on data flowing upstream in the network, receiving packets one at a time and outputting packets only when the specified synchronization criteria has been met. Synchronization filters provide a mechanism to deal with the asynchronous arrival of packets from children nodes. The synchronizer collects packets and typically aligns them into waves, passing an entire wave onward at the same time. Therefore, synchronization filters do no data transformation and can operate on packets in a type-independent fashion. MRNet currently supports three synchronization modes:

- *Wait For All*: wait for a complete wave (i.e., a packet from every child node) before producing output packets (SFILTER_WAITFORALL)
- *Do Not Wait:* output packets immediately (SFILTER_DONTWAIT)
- *Timeout* : output packets after ‘timeout’ milliseconds (SFILTER_TIMEOUT), or when a complete wave has been accumulated. The timeout period begins upon receipt of the first packet since the filter last produced output. The timeout value in milliseconds can be set using `Stream::set_FilterParameters`. Note that this timeout value is used at each level of the tree - a timeout value of 100ms combined with a tree of depth three should produce outputs at the front-end approximately 300ms after a packet is sent from a back-end. The default timeout value is 0ms. If you use SFILTER_TIMEOUT without setting a non-zero timeout value, it will behave similar to SFILTER_DONTWAIT.

Transformation filters can be used on both upstream and downstream data flows. Transformation filters input a group of synchronized packets, and combine data from multiple packets by performing an aggregation that yields one or more new data packets. Data packets produced by a transformation filter can be forwarded in either direction on a Stream by placing them in the appropriate output set. Since transformation filters are expected to perform computational operations on data packets, there is a type requirement for the data packets to be passed to this type of filter: the data format string of the stream’s packets and the filter must be the same. Transformation operations must be synchronous, but are able to maintain state from one execution to the next. MRNet provides several transformation filters that should be of general use:

- *Basic scalar operations on characters/integers/floats*: minimum (TFILTER_MIN), maximum (TFILTER_MAX), summation (TFILTER_SUM), average (TFILTER_AVG)

- *Concatenation*: operation that inputs n scalars and outputs a vector of length n of the same base type (TFILTER_ARRAY_CONCAT)

Appendix D: "Adding New Filters" on page 41 describes facilities for adding new user-defined transformation and synchronization filters.

3. A SIMPLE EXAMPLE

3.1 The MRNet Interface

A complete description of the MRNet API is in **“C++ API Reference” on page 8** and **“C API Reference” on page 26**. This section offers a brief overview only. Using `libmrnet`, a tool can leverage a system of internal processes, instances of the `mrnet_commnode` program, as a communication substrate. After instantiation of the MRNet network (discussed in “MRNet Instantiation” on page 6), the front-end and back-end processes are connected by the internal processes. The connection topology and host assignment of these processes is determined by a configuration file, thus the geometry of MRNet’s process tree can be customized to suit the physical topology of the underlying hardware resources. While MRNet can generate a variety of standard topologies, users can easily specify their own topologies; see **Appendix C: “Process-Tree Topologies” on page 39** for further discussion.

The MRNet API contains Network, EndPoint, Communicator, and Stream objects that a tool’s front-end and back-end use for communication. The Network object is used to instantiate the MRNet network and access EndPoint objects that represent available tool back-ends. The Communicator object is a container for groups of end-points, and Stream objects are used to send data to the EndPoints in a Communicator.

```
1    front_end_main(...) {
2          Network * net;
3          Communicator * comm;
4          Stream * stream;
5          PacketPtr packet;
6          int tag = FirstApplicationTag;
7          float result;
8
9          net = Network::CreateNetworkFE(topology_file, backend_exe, argv);
10         comm = net->get_BroadcastCommunicator( );
11         stream = net->new_Stream(comm, TFILTER_SUM, SFILTER_WAITFORALL);
12         stream->send(tag, "%d", SUM_INIT);
13         stream->recv(&tag, packet)
14         packet->unpack("%f", &result);
15   }
```

Figure 1: MRNet Front-End Sample Code

A simplified version of code from an example tool front-end is shown in **Figure 1**. In the front-end code, after some variable definitions in lines 2-6, an instance of the MRNet network is created on line 9 using the topology specified in topology_file. In line 10, the newly created Network object is queried for an auto-generated broadcast communicator that contains all available end-points. In line 11, this Communicator is used to establish a Stream that will use a built-in filter that finds the summation of the data sent upstream. The front-end then sends one or more initialization messages to the backends; in our example code on line 12, we broadcast an integer initializer on the new stream. The tag parameter is an application-specific value denoting the nature of the mes-

```
1    back_end_main(int argc, char** argv) {
2          Stream * stream;
3          PacketPtr packet;
4          int val, tag;
5          float random_float = (float) random( );
6
7          Network * net = Network::CreateNetworkBE(argc,argv);
8          net->recv(&tag, packet, &stream);
9          packet->unpack("%d", &val );
10         if( val == SUM_INIT )
11               stream->send(tag, "%f", random_float);
12   }
```

Figure 2: MRNet Back-End Sample Code

sage being transmitted. After the send operation, the front-end performs a blocking stream receive at line 13. This call returns a tag and a packet. Finally, line 14 calls unpack to deserialize the floating point value contained in packet.

Figure 2 shows the code for the back-end that reciprocates the actions of the front-end. Each tool back-end first connects to the MRNet network in line 5, using the back-end version of the Network constructor that receives its arguments via the program argument vector (`argc/argv`). While the front-end makes a stream-specific receive call, the back-ends use a stream-anonymous network receive that returns the tag sent by the front-end, the packet containing the actual data sent, and a stream object representing the stream that the front-end has established. Finally, each back-end sends a scalar floating point value upstream toward the front-end.

A complete example of MRNet code can be found below in **Appendix B: “A Complete Example: Integer Addition” on page 33**.

3.2 MRNet Instantiation

While conceptually simple, creating and connecting the internal processes is complicated by interactions with the various job scheduling systems. In the simplest environments, MRNet can launch processes directly using facilities like `rsh` or `ssh`. In more complex environments, it is necessary to submit all requests to a job management system. In this case, MRNet is constrained by the operations provided by the job manager (and these vary from system to system). Currently, two modes of instantiating MRNet-based tools are supported.

In the first mode of process instantiation, MRNet creates the internal and back-end processes, using the specified MRNet topology configuration to determine the hosts on which the components should be located. First, the front-end consults the configuration and uses a remote shell program to create internal processes for the first level of the communication tree on the appropriate hosts. Upon instantiation, the newly created processes establish a network connection to the process that created it. The first activity on this connection is a message from parent to child containing the portion of the configuration relevant to that child. The child then uses this information to begin instantiation of the sub-tree rooted at that child. When a sub-tree has been established, the root of that sub-tree sends a report to its parent containing the end-points accessible via that

sub-tree. Each internal node establishes its children processes and their respective connections sequentially. However, since the various processes are expected to run on different compute nodes, sub-trees in different branches of the network are created concurrently, maximizing the efficiency of network instantiation.

In the second mode of process instantiation, MRNet relies on a process management system to create some of the MRNet processes. This mode accommodates tools that require their back-ends to create, monitor, and control other processes. For example, IBM's POE uses environment variables to pass information, such as the process' rank within the application's global MPI communicator, to the MPI run-time library in each application process. In cases like this, MRNet cannot provide back-end processes with the environment necessary to start MPI application processes. As a result, MRNet creates its internal processes recursively as in the first instantiation mode, but does not instantiate any back-end processes. MRNet then waits for the tool back-ends to be started by the process management system to ensure they have the environment needed to create application processes successfully. To allow back-ends to connect to the MRNet network, information such as process host names and connection port numbers must be provided to the back-ends. This information can be provided via the environment, using shared filesystems or other information services as available on the target system. To collect the necessary information, the front-end can use the MRNet API methods for discovering the network topology details. This mode of process instantiation is referred to as "back-end attach mode". We show how to construct a tool that requires back-end attach in `$MRNET_ROOT/Examples/NoBackEndInstantiation`.

4. THE MRNET API

Standard MRNet relies on the back-end nodes supporting C++ libraries. However, we have also created a lightweight backend library with a pure C interface. The instantiation process is the same and both methods of process instantation are supported, although the API interface is slightly different.

4.1 C++ API Reference

All classes are included in the `MRN` namespace. For this discussion, we do not explicitly include reference to the namespace; for example, when we reference the class `Network`, we are implying the class `MRN::Network`.

In MRNet, there are five top-level classes: `Network`, `NetworkTopology`, `Communicator`, `Stream`, and `Packet`. The `Network` class primarily contains methods for instantiating and destroying MRNet process trees. The `NetworkTopology` class represents the interface for discovering details about the topology of an instantiated network. Application back-ends are referred to as end-points, and the `Communicator` class is used to reference a group of end-points. A communicator is used to establish a `Stream` for unicast, multicast, or broadcast communications via the MRNet infrastructure. The `Packet` class encapsulates the data packets that are sent on a stream. The public members of these classes are detailed below.

4.1.1 Class Network

The corresponding lightweight backend API class is **“Class Network” on page 26**.

```
Network * Network::CreateNetworkFE(
      const char * topology,
      const char * backend_exe,
      const char ** backend_argv,
      const std::map< std::string, std::string>* attrs=NULL,
      bool rank_backends = true,
      bool using_memory_buffer = false );
```

The front-end constructor method that is used to instantiate the MRNet process tree. `topology` is the path to a configuration file that describes the desired process tree topology.

`backend_exe` is the path to the executable to be used for the application’s back-end processes. `backend_argv` is a null terminated list of arguments to pass to the back-end application upon creation (NOTE: `backend_argv` shoud not contain the executable). If `backend_exe` is NULL, no back-end processes will be started, and the leaves of the topology specified by topology will be instances of `mrnet_commnode`.

`attrs` is a pointer to a map of attribute-value string pairs. `attrs` allows front-ends to programatically set the values to use for the MRNet and XPlat environment variables (see Table 3 on page 46) -- MRNet will only query the environment for settings not given via `attrs`. On Cray

XT, when communication or back-end processes of the MRNet tree are to be co-located with application processes, `attrs` must contain a string pair that provides either the "CRAY_ALPS_APID" or "CRAY_ALPS_APRUN_PID" attribute value (see Table 3 on page 46 for a description of these attributes).

`rank_backends` indicates whether the back-end process ranks should begin at 0, similar to MPI rank numbering, and defaults to `true`.

If `using_memory_buffer` is set to `true` (default is `false`), the topology parameter is actually a pointer to a memory buffer containing the specification, rather than the name of a file.

When this function completes without error, all MRNet processes specified in the topology will have been instantiated. You may use `Network::has_error` to check for successful completion. The explicit use of the `Network` constructor is now deprecated.

`Network * Network::CreateNetworkBE( int argc, char ** argv );`

The back-end constructor method that is used when the process is started due to a front-end network instantiation. MRNet automatically passes the necessary information to the back-end process using the program argument vector (`argc/argv`) by inserting it after the user-specified arguments. The explicit use of the `Network` constructor is now deprecated.

In the "back-end attach" mode of network instantiation, where the back-end is not launched directly by MRNet, the back-end program must construct a suitable argument vector. Typically, the front-end program will obtain information about the leaf `mrnet_commnode` processes using the `NetworkTopology` class, and pass this information to back-ends using external communication channels (e.g., a shared file system). The back-ends choose a leaf process as a parent, and use that parent's host, port, and rank information to attach. Each back-end must choose a unique value for its local rank; this value must be larger than any of the ranks of the processes in the existing network. The following code shows how to construct a valid argument vector:

```
char parHostname[64], myHostname[64], parPort[6], parRank[6], myRank[6];
// fill parent data here using info from front-end
gethostname( myHostname, 64 );
sprintf( myRank, "%d", <unique rank> );
be_argc = 6;
char* be_argv[be_argc];
be_argv[0] = argv[0];
be_argv[1] = parHostname;
be_argv[2] = parPort;
be_argv[3] = parRank;
be_argv[4] = myHostname;
be_argv[5] = myRank;
```

`void Network::~Network();`

`Network::~Network` tears down the MRNet process tree when the `Network` object is deleted. Note that `Network::shutdown_Network` is deprecated.

`void Network::waitfor_ShutDown();`

`Network::waitfor_ShutDown` can be used by back-ends to block until the network has been shut down by the front-end.

`bool Network::is_ShutDown();`

Back-ends use this method to query if the network has been shut down; returns `true` if it has been shut down, `false` otherwise.

`bool Network::set_FailureRecovery( bool enable );`

`Network::set_FailureRecovery` is used by a front-end to control whether internal communication processes and back-ends will automatically re-connect to a new parent when their parent terminates unexpectedly. By default, failure recovery is enabled and processes will reconnect. Call this method with `enable` set to `false` to turn off automatic failure recovery. This method returns `true` if the setting has been applied successfully, `false` otherwise.

`bool Network::has_Error( );`

`Network::has_error` returns `true` if an error has occured during the last call to a `Network` method. `Network::print_error` can be used to print a message describing the exact error.

`ErrorCode Network::get_Error( );`

`Network::get_Error` returns an `ErrorCode` for an error that occured during the last call to a `Network` method. `Network::get_ErrorStr` can be used to retrieve a message string describing the error.

`const char * Network::get_ErrorStr( ErrorCode code );`

`Network::get_ErrorStr` returns a character string describing the error indicated by `code`.

`void Network::print_error( const char * error_msg );`

`Network::print_error` prints a message to `stderr` describing the last error encountered during a `Network` method. It first prints the null-terminated string `error_msg` followed by a colon, then the actual error message followed by a newline.

`std::string Network::get_LocalHostName();`

`Network::get_LocalHostName` returns the name of the host on which the local MRNet process is running.

`Port Network::get_LocalPort();`

`Network::get_LocalPort` returns the listening port of the local MRNet process.

`Rank Network::get_LocalRank();`

`Network::get_LocalRank` returns the rank of the local MRNet process.

```
int Network::load_FilterFunc( const char * so_file, const char* func );
```

This method loads a new filter operation for use in the Network, and is conveniently similar to the conventional `dlopen` facilities for opening a shared object and dynamically loading symbols defined within.

`so_file` is the path to a shared object file that contains the filter function to be loaded and `func_name` is the name of the function to be loaded.

On success, `Network::load_FilterFunc` returns the id of the newly loaded filter which may be used in subsequent calls to `Network::new_Stream`. A value of -1 is returned on failure.

```
int Network::load_FilterFuncs(
      const char * so_file,
      const std::vector< const char* > & functions,
      std::vector< int > & filter_ids );
```

This method loads several new filter operations residing in the same shared library into the Network.

`so_file` is the path to a shared object file that contains the filter function to be loaded and `functions` is a vector of function names to be loaded. `filter_ids` is an output vector of filter ids, where the id for the function at index i in `functions` will be stored at index i in `filter_ids`.

`Network::load_FilterFuncs` returns the number of filter functions that were successfully loaded, and populates the the `filter_ids` vector with the ids of the newly loaded filters (or -1 if a function could not be loaded).

```
int Network::recv(
      int * tag,
      PacketPtr & packet,
      Stream ** stream,
      bool blocking = true );
```

`Network::recv` is used to invoke a stream-anonymous receive operation. Any packet available (i.e., addressed to any stream) will be returned (in roughly FIFO order).

`otag` will be filled in with the integer tag value that was passed by the corresponding `Stream::send` operation. `packet` is the packet that was received. A pointer to the stream to which the packet was addressed will be returned in `stream`.

`blocking` is used to signal whether this call should block or return if data is not immediately available; it defaults to a blocking call.

A return value of -1 indicates that the Network has experienced a terminal failure, and further attempts to send or receive data on the Network will fail. A return value of 0 indicates no

packets were available for a non-blocking receive, or a stream has been closed for a blocking receive. The return value 1 indicates a packet has been received successfully.

```
int Network::send(
      Rank be,
      int tag,
      const char * format_string, ... );
```

`Network::send` is used to singlecast a packet from the front-end to a specific back-end. `be` is the rank of the back-end process. `tag` is an integer that identifies the data in the packet. `format_string` is a format string describing the data in the packet (See **Appendix E: "Format Strings" on page 43** for a full description.)

A return value of -1 indicates that the Network has experienced a terminal failure, and further attempts to send or receive data on the Network will fail. The return value 0 indicates a packet has been sent successfully.

NOTE: `tag` must have a value greather than or equal to the constant `FirstApplicationTag` defined by MRNet (`#include "mrnet/Types.h"`). Tag values less than `FirstApplicationTag` are reserved for internal MRNet use.

```
bool Network::enable_PerformanceData(
      perfdata_metric_t metric,
      perfdata_context_t context );
```

`Network::enable_PerformanceData` uses `Stream::enable_PerformanceData` to start the recording of performance data of the specified `metric` type for the given `context` on all streams. Returns `true` on success, `false` otherwise. **Appendix F: "MRNet Stream Performance Data" on page 44** describes the supported metric and context types. See `Stream::enable_PerformanceData` for additional details.

```
bool Network::disable_PerformanceData(
      perfdata_metric_t metric,
      perfdata_context_t context );
```

`Network::disable_PerformanceData` stops the recording of performance data of the specified `metric` type for the given `context` on all streams. Returns `true` on success, `false` otherwise. See `Stream::disable_PerformanceData` for additional details.

```
bool Network::collect_PerformanceData(
      std::map< int, rank_perfdata_map > & results,
      perfdata_metric_t metric,
      perfdata_context_t context,
      int aggr_filter_id = TFILTER_ARRAY_CONCAT );
```

`Network::collect_PerformanceData` collects the performance data of the specified `metric` type for the given `context` on all streams. The performance data of each stream is passed

through the transformation filter identified by `aggr_filter_id`. The data for all streams is stored within the map `results`, keyed by stream identifier. Returns `true` on success, `false` otherwise. See `Stream::collect_PerformanceData` for additional details.

```
void Network::print_PerformanceData(
        perfdata_metric_t metric,
        perfdata_context_t context );
```

`Network::enable_PerformanceData` uses `Stream::print_PerformanceData` to print recorded performance data of the specified `metric` type for the given `context` on all streams. Data is printed to the MRNet log files. See `Stream::print_PerformanceData` for additional details.

```
unsigned int Network::num_EventsPending( );
```

`Network::num_EventsPending` returns the number of pending events available for retrieval using `Network::next_Event`.

```
Event * Network::next_Event( );
```

This method returns a pointer the next pending `Event`, or `NULL` if no events are available. Each event has an associated `EventClass`, one of `Event::DATA_EVENT`, `Event::TOPOLOGY_EVENT`, or `Event::ERROR_EVENT`, that can be queried using `Event::get_Class`. Similarly, each event has an associated `EventType` that can be queried using `Event::get_Type`.

```
void Network::clear_Events( );
```

This method clears all pending events.

```
bool Network::register_EventCallback(
        EventClass eclass,
        EventType etyp,
        evt_cb_func cb_func,
        void * cb_func_data,
        bool onetime = false );
```

`Network::register_EventCallback` allows users to register a callback function to be called when events are generated.

`eclass` should be set to one of `Event::DATA_EVENT`, `Event::TOPOLOGY_EVENT`, or `Event::ERROR_EVENT`.

`etyp` should be set to either `Event::EVENT_TYPE_ALL`, to have the function called when any event within the specified `EventClass` occurs, or one of the valid class-specific `EventType` values (see the classes `DataEvent`, `TopologyEvent`, and `ErrorEvent` in `"mrnet/Event.h"` for the class-specific types).

The type `evt_cb_func` is defined as '`void (*evt_cb_fn)( Event* e, void* cb_data )`'. All user-defined callback functions must use the same function prototype. When an event

occurs, all callbacks registered for that type of event will be called. Each function is passed a pointer to the `Event`, and the value of the auxiliary data pointer `cb_func_data` given at registration, which may be `NULL`.

`onetime` should be set to `true` if the function should be removed after it is called for the first (and only) time. Note that onetime callbacks must be registered for a specific event type.

```
void Network::remove_EventCallback(
        evt_cb_func cb_func,
        EventClass eclass,
        EventType etyp );
```

This method removes `cb_func` from the list of functions to be called for the specified `EventClass` and `EventType`. If `eclass` is given as `Event::EVENT_CLASS_ALL`, the function will be removed for all events. `etyp` can be given as `Event::EVENT_TYPE_ALL` to remove the function for all types of events in the given `eclass`.

```
void Network::remove_EventCallbacks(
        EventClass eclass,
        EventType etyp );
```

This method removes all functions to be called for the specified `EventClass` and `EventType`. If `eclass` is given as `Event::EVENT_CLASS_ALL`, all callback functions will be removed for all events. `etyp` can be given as `Event::EVENT_TYPE_ALL` to remove all functions registered for all types of events in the given `eclass`.

```
int Network::get_EventNotificationFd( EventClass eclass );
```

`Network::get_EventNotificationFd` returns a file descriptor that can be used with `select` or `poll` to receive notification of interesting DATA, TOPOLOGY, or ERROR events.

`eclass` should be set to one of `Event::DATA_EVENT`, `Event::TOPOLOGY_EVENT`, or `Event::ERROR_EVENT`. `Event::DATA_EVENT` can be used by both front-end and back-end processes to provide notification that one or more data packets have been received. `Event::TOPOLOGY_EVENT` and `Event::ERROR_EVENT` can only be used by front-end processes, and provide notification when the front-end observes a change in network topology or an error, respectively.

When the file descriptor has data available (for reading), you should call `Network::clear_EventNotificationFd` before taking action on the notification. When notifications are no longer needed, use `Network::close_EventNotificationFd`.

NOTE: this functionality is not available on Windows platforms.

void Network::clear_EventNotificationFd(EventClass eclass);

This method resets the event notification file descriptor returned from `Network::get_EventNotificationFd`. `eclass` should be set to one of `Event::DATA_EVENT`, `Event::TOPOLOGY_EVENT`, or `Event::ERROR_EVENT`.

NOTE: this functionality is not available on Windows platforms.

void Network::close_EventNotificationFd(EventClass eclass);

This method closes the event notification file descriptor returned from `Network::get_EventNotificationFd`. `eclass` should be set to one of `Event::DATA_EVENT`, `Event::TOPOLOGY_EVENT`, or `Event::ERROR_EVENT`.

NOTE: this functionality is not available on Windows platforms.

bool is_LocalNodeChild( ) const;

bool is_LocalNodeParent( ) const;

bool is_LocalNodeInternal( ) const;

bool is_LocalNodeFrontEnd( ) const;

bool is_LocalNodeBackEnd( ) const;

These methods return `true` if the local process is of the specified type, `false` otherwise.

4.1.2 Class NetworkTopology

Instances of `NetworkTopology` are network specific, so they are created when a `Network` is instantiated. MRNet API users should not need to create their own `NetworkTopology` instances.

The corresponding lightweight backend API class is **“Class NetworkTopology” on page 28**.

NetworkTopology * Network::get_NetworkTopology( );

`Network::get_NetworkTopology` is used to retrieve a pointer to the underlying `NetworkTopology` instance of a `Network`.

unsigned int NetworkTopology::get_NumNodes( );

This method returns the total number of nodes in the tree topology, including front-end, internal, and back-end processes.

NetworkTopology::Node * NetworkTopology::find_Node(Rank node_rank);

This method returns a pointer to the tree node with rank equal to `node_rank`, or NULL if not found.

NetworkTopology::Node * NetworkTopology::get_Root( );

This method returns a pointer to the root node of the tree, or NULL if not found.

```
void NetworkTopology::get_Leaves(
      std::vector<NetworkTopology::Node * > & leaves );
```

This method fills the `leaves` vector with pointers to the leaf nodes in the topology. In the case where back-end processes are not started when the network is instantiated, a front-end process can use this function to retrieve information about the leaf internal processes to which the back-ends should attach.

```
void NetworkTopology::get_BackEndNodes(
      std::set< NetworkTopology::Node * > & nodes );
```

This method fills a set with pointers to all back-end process tree nodes. Note that this method is unsafe to use while the network topology is in flux, as is the case during the "back-end attach" mode of MRNet tree instantiation.

```
void NetworkTopology::get_ParentNodes(
      std::set<NetworkTopology::Node * > & nodes );
```

This method fills a set with pointers to all tree nodes that are parents (i.e., those nodes having at least one child).

```
void NetworkTopology::get_OrphanNodes(
      std::set< NetworkTopology::Node * > & nodes );
```

This method fills a set with pointers to all tree nodes that have no parent due to a failure.

```
void NetworkTopology::get_TreeStatistics(
      unsigned int & num_nodes,
      unsigned int & depth,
      unsigned int & min_fanout,
      unsigned int & max_fanout,
      double & avg_fanout,
      double & stddev_fanout );
```

This method provides users statistics about the tree topology by setting the passed parameters.

`num_nodes` is the total number of tree nodes (same as the value returned by `NetworkTopology::get_NumNodes`), `depth` is the depth of the tree (i.e., the maximum path length from root to any leaf), `min_fanout` is the minimum number of children of any parent node, `max_fanout` is the maximum number of children of any parent node, `avg_fanout` is the average number of children across all parent nodes, and `stddev_fanout` is the standard deviation in number of children across all parent nodes.

```
void NetworkTopology::print_TopologyFile( const char * filename );
```

This method will create (or overwrite) the specified topology file `filename` using the current state of this `NetworkTopology` object.

```
void NetworkTopology::print_DOTGraph( const char * filename );
```

This method will create (or overwrite) the specified dot graph file `filename` using the current state of this `NetworkTopology` object.

```
std::string NetworkTopology::Node::get_HostName();
```

This method returns a string identifying the hostname of the tree node.

```
Port NetworkTopology::Node::get_Port();
```

This method returns the listening port of the tree node.

```
Rank NetworkTopology::Node::get_Rank();
```

This method returns the unique rank of the tree node.

```
Rank NetworkTopology::Node::get_Parent();
```

This method returns the rank of the tree node's parent.

```
const std::set< NetworkTopology::Node * > &
NetworkTopology::Node::get_Children();
```

This method returns a set containing pointers to the children of the tree node, and is useful for navigating through the tree.

```
unsigned int NetworkTopology::Node::get_NumChildren();
```

This method returns the number of children of the tree node.

```
unsigned int NetworkTopology::Node::find_SubTreeHeight();
```

This method returns the height of the subtree rooted at this `NetworkTopology` node.

4.1.3 Class Communicator

Instances of `Communicator` are network specific, so their creation methods are functions of an instantiated `Network` object. There is no corresponding lightweight backend class.

```
Communicator * Network::new_Communicator();
```

This method returns a pointer to a new `Communicator` object. The object contains no end-points. Use `Communicator::add_EndPoint` to populate the communicator.

```
Communicator * Network::new_Communicator( Communicator & comm );
```

This method returns a pointer to a new `Communicator` object that contains the same set of end-points contained in `comm`.

```
Communicator * Network::new_Communicator(
      std::set< CommunicationNode * > & endpoints );
```

This method returns a pointer to a new `Communicator` object that contains the provided set of end-points.

```
Communicator * Network::new_Communicator( std::set< Rank > & endpoints );
```

This method returns a pointer to a new `Communicator` object that contains the set of end-points corresponding to processes whose ranks are provided in the passed set.

```
Communicator * Network::get_BroadcastCommunicator( );
```

This method returns a pointer to a broadcast `Communicator` containing all the end-points available in the system at the time the function is called.

Multiple calls to this method return the same pointer to the `Communicator` object created at network instantiation. If the network topology changes, as can occur when starting back-ends separately, the object will be updated to reflect the additions or deletions. This object should not be deleted.

```
bool Communicator::add_EndPoint( Rank ep_rank );
```

This method is used to add an existing end-point with rank `ep_rank` to the set contained by this `Communicator`.

If the set of end-points in the communicator already contains the new end-point, the function returns success. This method fails if there exists no end-point defined by `ep_rank`. This method returns `true` on success, `false` on failure.

```
bool Communicator::add_EndPoint( CommunicationNode * endpoint );
```

This method is similar to `add_EndPoint` above except that it takes a pointer to a `CommunicationNode` object instead of a rank. Success and failure conditions are exactly as stated above. This method returns `true` on success and `false` on failure.

```
const std::set< CommunicationNode * > & Communicator::get_EndPoints( );
```

Returns a reference to the set of `CommunicationNode` pointers comprising the end-points in the communicator.

```
std::string CommunicationNode::get_HostName( );
```

Returns a character string identifying the hostname of the end-point represented by this `CommunicationNode`.

```
Port CommunicationNode::get_Port( );
```

Returns the listening port of the end-point represented by this `CommunicationNode`.

```
Rank CommunicationNode::get_Rank( );
```

Returns the rank of the end-point represented by this `CommunicationNode`.

4.1.4 Class Stream

Instances of `Stream` are network specific, so their creation methods are functions of an instantiated `Network` object. The corresponding lightweight backend API class is **“Class Stream” on page 28**.

MRNet provides two types of streams, homegenous and heterogeneous. Homogenous streams use the same filters at every process participating in the stream, while heterogeneous streams allow for different filters to be used at different processes.

```
Stream * Network::new_Stream(
      Communicator * comm,
      int up_transfilter_id = TFILTER_NULL,
      int up_syncfilter_id = SFILTER_WAITFORALL,
      int down_transfilter_id = TFILTER_NULL );
```

This version of `Network::new_Stream` is used to create a homogenous `Stream` object attached to the end-points specified by a Communicator object `comm`.

`up_transfilter_id` specifies the transformation filter to apply to data flowing upstream from the application back-ends toward the front-end; the default value is `TFILTER_NULL`.

`up_syncfilter_id` specifies the synchronization filter to apply to upstream packets; the default value is `SFILTER_WAITFORALL`.

`down_transfilter_id` allows the user to specify a filter to apply to downstream data flows; the default value is `TFILTER_NULL`.

```
Stream * Network::new_Stream(
      Communicator * comm,
      std::string us_filters,
      std::string sync_filters,
      std::string ds_filters );
```

This version of `Network::new_Stream` is used to creae a heterogeneous `Stream` object. Users specify where packet filters are placed within the tree. Like the homogenous version of `Network::new_Stream`, the end-points are specified by the `comm` argument.

Strings are used to specify the filter placements, with the following syntax: "`filter_id => rank; [filter_id => rank; ...]`". If "`*`" is specified as the `rank` for an assignment, the filter will be assigned to all ranks that have not already been assigned. If a rank within `comm` is not assigned a filter, it will use the default filter. See `$MRNET_ROOT/Examples/Heteroge-`

neousFilters for an example of using Network::new_Stream to specify different filter types to be used within the same stream.

us_filters specifies the transformation filters to apply to data flowing upstream from the application back-ends toward the front-end.

sync_filters specifies the synchronization filters to apply to upstream packets.

ds_filters allows the user to specify filters to apply to downstream data flows.

Note that more than one filter should not be assigned to a single rank in any of these strings.

```
Stream * Network::get_Stream( unsigned int id );
```

Returns a pointer to the Stream identified by id, or NULL on failure. Back-ends may pass their local rank as the id to retrieve a singlecast stream that can be used for non-filtered communication directly with the front-end.

```
unsigned int Stream::get_Id( );
```

Returns the integer identifier for this Stream.

```
const std::set< Rank > & Stream::get_EndPoints( );
```

Returns the set of end-point ranks for this Stream.

```
unsigned int Stream::size( );
```

Returns an integer indicating the number of end-points for this Stream.

```
bool Stream::is_Closed( );
```

When used by back-ends, this method returns true if the front-end has closed this Stream by deleting the corresponding object, false otherwise. On the front-end, this method can be used to determine if the stream has been disabled due to a non-recoverable failure (e.g., a back-end process has died or a sub-tree containing stream end-points has become unreachable).

```
int Stream::send( int tag, const char * format_string, ... );
int Stream::send( const char * format_string, va_list list, int tag );
int Stream::send( int tag, const void** data, const char * format_string );
int Stream::send( PacketPtr & pkt );
```

Invokes a data send operation on the calling `Stream`. The first three interfaces construct a packet from the passed operands, while the fourth allows for sending an already constructed packet.

`tag` is an integer that identifies the data in the packet.

`format_string` is a format string describing the data in the packet (See **Appendix E: "Format Strings" on page 43** for a full description).

`data` is an array of pointers to individual data items; the format string indicates the type of data pointed to by each array index.

On success, `Stream::send` returns 0; on failure -1.

NOTE: `tag` must have a value greather than or equal to the constant `FirstApplicationTag` defined by MRNet (`#include "mrnet/Types.h"`). Tag values less than `FirstApplicationTag` are reserved for internal MRNet use.

```
int Stream::flush();
```

Commits a flush of all packets currently buffered by this `Stream`. A successful return value of 0 indicates that all buffered packets have been passed to the operating system for network transmission. A return value of -1 indicates that the stream has experienced a terminal failure, and further attempts to send or receive data on the stream will fail.

```
int Stream::recv( int * tag, PacketPtr & packet, bool blocking = true );
```

Invokes a stream receive operation. Packets received by the calling `Stream` will be returned by this method, one-at-a-time, in FIFO order.

`tag` will be filled in with the integer tag value that was passed by the corresponding `Stream::send` operation. `packet` is set to point to the received packet.

`blocking` determines whether the receive should block or return if data is not immediately available; it defaults to a blocking call.

A return value of -1 indicates that the stream has experienced a terminal failure, and further attempts to send or receive data on the stream will fail. A return value of 0 indicates no packets were available for a non-blocking receive, or the stream has been closed. The return value 1 indicates a packet has been received successfully.

int Stream::get_DataNotificationFd( );

`Stream::get_DataNotificationFd` returns a file descriptor that can be used with `select` or `poll` to receive notification that data has arrived for a stream.

When the file descriptor has data available (for reading), you should call `Stream::clear_DataNotificationFd` before taking action on the notification. When notifications are no longer needed, use `Stream::close_DataNotificationFd`.

NOTE: this functionality is not available on Windows platforms.

void Stream::clear_DataNotificationFd( );

This method resets the data notification file descriptor returned from `Stream::get_DataNotificationFd`.

NOTE: this functionality is not available on Windows platforms.

void Stream::close_DataNotificationFd( );

This method closes the data notification file descriptor returned from `Stream::get_DataNotificationFd`.

NOTE: this functionality is not available on Windows platforms.

```
int Stream::set_FilterParameters(
      FilterType ftype,
      const char *format_str, ... ) const;
```

`Stream::set_FilterParameters` allows users to dynamically configure the operation of a stream transformation filter by passing arbitrary data in a similar fashion to `Stream::send`. When the filter executes, the passed data is available as a `PacketPtr` parameter to the filter, and the filter can extract the configuration settings.

ftype should be given as `FILTER_UPSTREAM_SYNC` to configure the synchronization filter, `FILTER_UPSTREAM_TRANS` for upstream transformation filter and `FILTER_DOWNSTREAM_TRANS` for downstream transformation filter.

```
int Stream::set_FilterParameters(
      const char *format_str,
      va_list params,
      FilterType ftype ) const;
```

This method is the same as the previous method except for the filter configuration parameters are given in the `va_list` form.

```
bool Stream::enable_PerformanceData(
      perfdata_metric_t metric,
      perfdata_context_t context );
```

`Stream::enable_PerformanceData` starts recording performance data for the specified `metric` type for the given `context`. Returns `true` on success, `false` otherwise. **Appendix F: "MRNet Stream Performance Data" on page 44** describes the metric and context types.

```
bool Stream::disable_PerformanceData(
      perfdata_metric_t metric,
      perfdata_context_t context );
```

`Stream::disable_PerformanceData` stops recording performance data for the specified `metric` type for the given `context`. Previously recorded data is not discarded, so that it can be retrieved with `Stream::collect_PerformanceData`. Users can enable/disable recording for a particular `metric` and `context` any number of times before collecting the results. Returns `true` on success, `false` otherwise.

```
bool Stream::collect_PerformanceData(
      rank_perfdata_map & results,
      perfdata_metric_t metric,
      perfdata_context_t context,
      int aggr_filter_id = TFILTER_ARRAY_CONCAT );
```

`Stream::collect_PerformanceData` collects the recorded performance data for the specified `metric` type for the given `context`. The collected data is returned in a `rank_perfdata_map`, which associates individual node ranks to a `std::vector< perf_data_t >` containing the recorded data instances. After collection, the recorded data at each nodeis discarded. Returns `true` on success, `false` otherwise.

Users can aggregate the recorded data across nodes by specifying a transformation filter with `aggr_filter_id`. Note that only the built-in filter types of `TFILTER_SUM`, `TFILTER_MIN`, `TFILTER_MAX`, `TFILTER_AVG`, and `TFILTER_ARRAY_CONCAT` are supported. By default, performance data from each node is concatenated, and results contains every recorded data instance for each node. If the summary aggregation filters are used, results will contain a single associated pair. The rank for this pair is equal to `-1×(number of aggregated ranks)`, and the vector contains one or more aggregated instances.

```
void Stream::print_PerformanceData(
      perfdata_context_t metric,
      perfdata_context_t context );
```

`Stream::print_PerformanceData` prints recorded performance data of the specified `metric` type for the given `context`. At each rank, the data is printed to the MRNet log files and then discarded.

4.1.5 Class Packet

A `Packet` encapsulates a set of formatted data elements sent on a stream. Packets are created using a format string (e.g., "`%s %d`" describes a null-terminated string followed by a 32-bit integer, and the packet is said to contain two *data elements*). MRNet front-end and back-end processes typically do not create `Packet` instances, as they are automatically produced from the formatted data passed to `Stream::send` or `Network::send`. Each `Packet` is allocated using `malloc` of the C standard library, and therefore has the same byte alignment guarantees. **Appendix E: "Format Strings" on page 43** contains the full listing of data types that can be sent in a `Packet`.

When receiving a packet via `Stream::recv`, `Network::recv`, or in a filter's input vector, the `Packet` instance is stored within a `PacketPtr` object. `PacketPtr` is a class based on the Boost `shared_ptr` class, and helps with memory management of packets. A `PacketPtr` can be assumed to be equivalent to "`Packet *`", and all operations on packets require use of `PacketPtr`. Packets can be created explicitly using the constructors shown below, using the following method to instantiate a `PacketPtr`.

```
PacketPtr new_pkt( new Packet(...) );
```

The corresponding lightweight backend API class is **"Class Packet" on page 30**.

```
Packet::Packet(
    unsigned int stream_id,
    int tag,
    const char* format_str, ... );
```

Constructs a `Packet` that can be sent on the stream with the given `stream_id`. The packet is associated with a `tag` that can be used by receivers to identify the contents. `format_str` is a format string describing the data elements in the packet. The variable arguments following `format_str` should have the appropriate types for each data element. Note that for array data elements, an extra argument must be passed to hold each array's length. (See **Appendix E: "Format Strings" on page 43** for a full description.)

```
Packet::Packet(
    const char* format_str,
    va_list data_elems,
    unsigned int stream_id,
    int tag );
```

Works the same as the first `Packet` constructor, but allows for passing a `va_list` in place of the variable arguments. This constructor is useful for libraries built on top of MRNet that allow users to specify packet format strings.

```
Packet::Packet(
      unsigned int stream_id,
      int tag,
      const void** data_elems,
      const char* format_str );
```

Works the same as the first `Packet` constructor, but allows for passing an array of data element pointers instead of the variable arguments. The `data_elems` array must contain the same number of elements as indicated by `format_str`.

```
int Packet::get_Tag( );
```

Returns the integer tag associated with this `Packet`.

```
void Packet::set_Tag( int tag );
```

Sets the integer `tag` associated with this `Packet`.

```
unsigned int Packet::get_StreamId( );
```

Returns the stream id associated with this `Packet`.

```
void Packet::set_StreamId( unsigned int strm_id );
```

Sets the stream id associated with this `Packet` to `strm_id`.

```
const char * Packet::get_FormatString( );
```

Returns the character string specifying the data format of this `Packet`.

```
int Packet::unpack( const char * format_str, ... );
int Packet::unpack( va_list list, const char * format_str, bool );
```

Extracts data contained within this `Packet` according to the `format_str`, which must match that of the packet. `format_str` is a format string describing the data in the packet (See **Appendix E: "Format Strings" on page 43** for a full description).

For the first version, the function arguments following `format_str` should be pointers to the appropriate types of each data item. For string and array data types, new memory buffers to hold the data will be allocated using `malloc`, and it is the user's responsibility to `free` these strings and arrays. Note that for array data elements, an extra argument must be passed to hold each array's length.

For the second version, the va_list list should contain the arguments corresponding to the varargs from the first version. The third parameter is a dummy parameter required by some compilers to distinguish the second version from the first version, and its value is ignored.

The return value 0 indicates success; -1 indicates the format string supplied did not match the packet or a failure in unpacking.

```
void Packet::set_Tag( int tag );
```

This method can be used to set the packet's tag value after it has been created.

```
void Packet::set_Destinations( const Rank * bes, unsigned int num_bes );
```

This method can be used to tell MRNet to deliver the packet to a specific set of back-ends, rather than all the back-ends addressed by the stream on which the packet is sent. `bes` should point to an array of back-end ranks, and `num_bes` is the number of entries in the array.

```
void Packet::set_DestroyData( bool destroy );
```

This method can be used to tell MRNet whether or not to deallocate the string and array data members of a `Packet`. If `destroy` is `true`, string and array data members will be deallocated using `free` when the `Packet` destructor is executed - this assumes they were allocated using `malloc`. The default behavior for user-generated packets is not to deallocate (`false`). Turning on deallocation is useful in filter code that must allocate strings or arrays for output packets, which cannot be freed before the filter function returns.

4.2 C API Reference

In the MRNet lightweight back-end library, the MRNet C++ classes are mimicked for ease of use. With the exception of constructors/destructors, API calls in standard MRNet can be translated to their lightweight versions according to the following pattern:

```
return_type class::function_name( param1_type param1, ...);
```

translates to:

```
return_type class_function_name(
      class class_object,
      param1_type param1, ...);
```

4.2.1 Class Network

```
Network_t * Network_CreateNetworkBE( int argc, char ** argv );
```

The back-end constructor method. MRNet automatically passes the necessary information to the back-end process using the program argument vector (`argc/argv`) by inserting it after the user specified arguments. See **"Network * Network::CreateNetworkBE(int argc, char ** argv);" on page 9** for more information on the required arguments.

```
void delete_Network_t( Network_t * network );
```

`delete_Network_t` acts as a destructor for the `Network_t` object and cleans up internal structures before freeing the `Network_t` pointer.

```
void Network_waitfor_ShutDown( Network_t * network );
```

`Network_waitfor_ShutDown` blocks until the network has been shut down.

```
char Network_is_ShutDown( Network_t * network );
```

Returns `true` if the network has been shut down.

```
char* Network_get_LocalHostName( Network_t * network );
```

`Network_get_LocalHostName` returns the name of the host where the process is running.

```
Port Network_get_LocalPort( Network_t * network );
```

`Network_get_LocalPort` returns the listening port of the local process.

```
Rank Network_get_LocalRank( Network_t * network );
```

`Network_get_LocalRank` returns the rank of the local process.

```
int Network_recv(
      Network_t * network,
      int otag,
      Packet_t * packet,
      Stream_t * stream );
```

`Network_recv` is used to invoke a blocking stream-anonymous receive operation. Any packet available (i.e., addressed to any stream) will be returned in roughly FIFO order.

`otag` will be filled in with the integer tag value that was passed by the corresponding `Stream_send` operation. `packet` is the packet that was received. A pointer to the `Stream_t` to which the packet was addressed will be returned in `stream`.

A return value of -1 indicates an error and 1 indictes a success.

```
int Network_recv_nonblock(
      Network_t * network,
      int otag,
      Packet_t * packet,
      Stream_t * stream );
```

`Network_recv_nonblock` is used to invoke a non-blocking stream-anonymous receive operation. Any packet available (i.e., addressed to any stream) will be returned in roughly FIFO order.

`otag` will be filled in with the integer tag value that was passed by the corresponding `Stream_send` operation. `packet` is the packet that was received. A pointer to the `Stream_t` to which the packet was addressed will be returned in `stream`.

A return value of -1 indicates an error, 0 indicates no packets were available, and 1 indicates a success.

4.2.2 Class NetworkTopology

```
NetworkTopology_t * Network_get_NetworkTopology( Network_t * network );
```

Network_get_NetworkTopology is used to retrieve a pointer to the underlying NetworkTopology_t instance within network. Note that in the lightweight back-end library, the information available in the NetworkTopology_t may be a subset of the full topology.

```
Node_t * NetworkTopology_find_Node(
      NetworkTopology_t * net_top,
      Rank node_rank );
```

This method returns a pointer to the topology node with rank equal to node_rank, or NULL if no match is found.

```
Node_t * NetworkTopology_get_Root( NetworkTopology_t * net_top );
```

This method returns a pointer to the root node of the tree, or NULL if not found.

```
char * NetworkTopology_Node_get_HostName( Node_t * node );
```

This method returns a string identifying the hostname of the node.

```
Port NetworkTopology_Node_get_Port( Node_t * node );
```

This method returns the listening port of the node.

```
Rank NetworkTopology_Node_get_Rank( Node_t * node );
```

This method returns the rank of the node.

```
Rank NetworkTopology_Node_get_Parent( Node_t * node );
```

This method returns the rank of the node's parent.

```
unsigned int NetworkTopology_Node_find_SubTreeHeight( Node_t * node );
```

This method returns the height of the sub-tree rooted at the node.

4.2.3 Class Stream

```
Stream_t * Network_get_Stream( Network_t * network, unsigned int id );
```

Network_get_Stream returns a pointer to a Stream_t identified by id, or NULL on failure. Back-ends may pass their local rank as the id to retrieve a singlecast stream that can be used for non-filtered communication directly with the front-end.

```
void delete_Stream_t( Stream_t * stream );
```

delete_Stream_t acts as a destructor for the Stream_t object and cleans up internal structures before freeing the Stream_t pointer.

```
unsigned int Stream_get_Id( Stream_t * stream );
```

This method returns the integer identifier for this `Stream_t`.

```
int Stream_send(
      Stream_t * stream,
      int tag,
      const char * format_string, ... );
```

This method sends data on `stream`. `tag` is an integer that identifies the data to be sent by the stream. `format_string` is a format string describing the types of the data elements (see **Appendix E: "Format Strings" on page 43** for a full description.) On success, `Stream_send` returns 0; on failure, -1.

NOTE: `tag` must have a value greater than or equal to the constant `FirstApplicationTag` defined by MRNet (`#include "mrnet_lightweight/Types.h"`). Tag values less than `FirstApplicationTag` are reserved for internal MRNet use.

```
int Stream_send_packet(
      Stream_t * stream,
      Packet_t * packet );
```

This method sends `packet` on `stream`. On success, `Stream_send_packet` returns 0; on failure, -1.

```
int Stream_flush( Stream_t * stream );
```

This operation is currently not required in lightweight MRNet, as `Stream_send` will deliver the data for network transmission. This method will always return the value 0 for success.

```
int Stream_recv(
      Stream_t * stream,
      int * tag,
      Packet_t * packet,
      bool_t blocking );
```

`Stream_recv` invokes a stream-specific receive operation. Packets addressed to the passed `stream` will be returned, one-at-a-time, in FIFO order. If `blocking` is `true`, the operation will block until a packet is available for this stream; if `false`, the operation will return immediately.

`tag` will be filled in with the integer tag value that was passed by the corresponding `Stream::send` operation. `packet` is the received `Packet_t`.

A return value of -1 indicates an error, 0 indicates no packet available for a non-blocking receieve, and 1 indicates a packet was found.

```
char Stream_is_Closed( Stream_t * stream );
```

This method returns the value 1 if the stream has been closed by the front-end, 0 otherwise.

4.2.4 Class Packet

When receiving a packet, it is stored within a `Packet_t` object. Note that standard MRNet makes use of the `PacketPtr` object, which is based on the Boost library `shared_ptr` class. However, in the lightweight back-end library, pointers to `Packet_t` objects are used instead.

```
int Packet_get_Tag( Packet_t * packet );
```

This method returns the integer tag associated with `packet`.

```
void Packet_set_Tag( Packet_t * packet, int tag );
```

This method sets the integer tag associated with `packet`.

```
unsigned int Packet_get_StreamId( Packet_t * packet );
```

This method returns the stream id associated with `packet`.

```
void Packet_set_StreamId( Packet_t * packet, unsigned int strm_id );
```

This method sets the stream id associated with `packet`.

```
char* Packet_get_FormatString( Packet_t * packet );
```

This method returns the character string specifying the data format of `packet`.

```
void Packet_unpack(
      Packet_t * packet,
      const char * format_string, ... );
```

This method extracts data elements contained within `packet` according to the `format_string`, which must match that of `packet`. The function arguments following `format_string` should be pointers to the appropriate types of each data element. For string and array data types, new memory buffers to hold the data will be allocated using `malloc`, and it is the user's responsibility to `free` these strings and arrays. Note that for array data elements, an extra argument must be passed to hold each array's length.

The return value 0 indicates success; -1 indicates the format string supplied did not match the packet or a failure in unpacking.

```
void Packet_unpack_valist(
      Packet_t * packet,
      va_list arg_list,
      const char * format_string );
```

This method extracts data elements contained within `packet` according to the `format_string`, which must match that of `packet`. The function arguments contained in the

va_list `arg_list` should be pointers to the appropriate types of each data element. For string and array data types, new memory buffers to hold the data will be allocated using `malloc`, and it is the user's responsibility to `free` these strings and arrays. Note that for array data elements, an extra argument must be passed to hold each array's length. The fourth parameter

The return value 0 indicates success; -1 indicates the format string supplied did not match the packet or a failure in unpacking.

APPENDIX A: BUILDING AND TESTING MRNET

For this discussion, `$MRNET_ROOT` is the location of the top-level directory of the MRNet distribution and `$MRNET_ARCH` is a string describing the platform (OS and architecture) as discovered by the configure process.

A.1: Supported Platforms and Compilers

MRNet has been developed to be highly portable; we expect it to run properly on all common Unix-based as well as Windows platforms. Please refer to the README document included with the MRNet distribution for the list of currently supported platforms.

MRNet requires GNU make for building on Unix/Linux systems. Our build system attempts to use native system compilers where available. For building on Windows systems, Visual Studio 2010 solution/project files are available, as are pre-built libraries and binaries.

A.2: Configuration, Compilation, and Installation

Please refer to the INSTALL document included with the MRNet distribution for configuration, compilation, and installation instructions.

A.3: Testing the Code

The shell script, `mrnet_tests.sh` is placed in the build/installation directory for binaries along with the other executables. This script can be used to run the MRNet test programs and check their output for errors. The script is used as follows:

```
UNIX> mrnet_tests.sh { -l | -r hostfile | -a hostfile }

              [ -f ] [ -lightweight ]
```

One of the `-l`, `-r`, or `-a` flags is required. The `-l` flag is used to run all tests using only topologies that create processes on the local machine (note: running the tests locally can take quite a while, up to 15 minutes depending on machine capabilities). The `-r` flag runs tests using remote machines specified in the file whose name immediately follows this flag. To run tests both locally and remotely, use the `-a` flag and provide a hostfile.

To test MRNet's ability to dynamically load shared libraries containing filter functions, you must specify the `-f` flag. The `-lightweight` flag is used to run the tests using lightweight back-ends in addition to the standard back-ends.

A.4: Bugs, Questions, and Comments

MRNet is maintained by the Paradyn Tools Project at the University of Wisconsin-Madison. Comments and feedback whether positive or negative are encouraged; please send to mrnet@cs.wisc.edu. Bug fixes as patches are also welcome.