Network Locality (netloc)

dev-52-g3f99e7e

Generated by Doxygen 1.6.1

Fri Aug 22 21:00:19 2014

Contents

10 Data Structure Documentation 83

Chapter 1

Network Locality

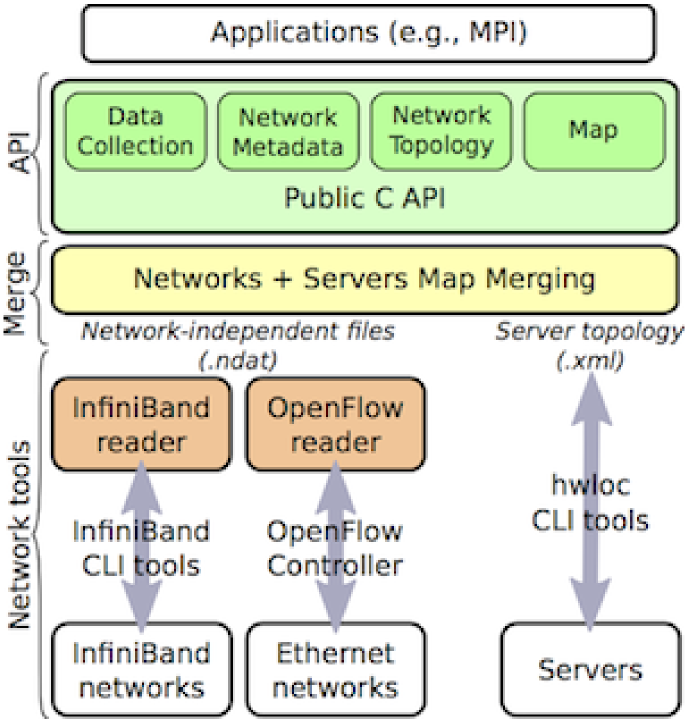
Portable abstraction of network topologies for high-performance computing

1.1 Introduction

The Portable Network Locality (netloc) software package provides network topology discovery tools, and an abstract representation of those networks topologies for a range of network types and configurations. It is provided as a companion to the Portable Hardware Locality (hwloc) package. These two software packages work together to provide a comprehensive view of the HPC system topology, spanning from the processor cores in one server to the cores in another - including the complex network(s) in between.

Towards this end, netloc is divided into three components:

- Network topology discovery tools for each network type and discovery technique (called readers)
- Merging hwloc server topology information with that network topology information to produce a unified map of an entire computing system (even if that system includes multiple networks of different types, and servers that are on at least one of those networks).
- A portable C API for higher-level software to query, traverse, and manipulate the abstract representation of the network topology and unified map (represented as a graph)



The network topology graph not only provides information about the physical nodes and edges in the network topology, but also information about the paths between nodes (both physical and logical, where available). Since the type of analysis (e.g., graph partitioning) required of this graph is often application-specific, netloc limits the amount of analysis it performs and leaves further analysis to higher level applications and libraries built upon this service. Additionally, the lsnettopo tool can display and export this network topology information in a variety of formats (e.g., GraphML and GEXF file formats) providing developers with an additional mechanism to access the data for further analysis.

Similar to hwloc, netloc primarily aims at helping applications with gathering information about modern computing and networking hardware so as to exploit it accordingly and efficiently.

1.1.1 Supported Networks

The following networks are currently supported:

- InfiniBand (See Reader: InfiniBand).
- OpenFlow-managed Ethernet newtworks (See Reader: OpenFlow-managed Ethernet). Below are the supported OpenFlow controllers:

  - `Cisco XNC`
  - `Floodlight`
  - `OpenDaylight`
- Static (User Defined) (See Reader: Static (User Defined)).

1.2 Installation

The typical installation follows the following pattern:

```
shell$ ./configure [options...]
shell$ make
shell$ make install
```

1.2.1 Configure Parameters

There are a few configuration options available. See ./configure --help for a complete list.

```
 --prefix=<directory>
   Install netloc into the base directory specified.

 --with-jansson=<directory>
   Installation directory of the Jansson JSON parsing library.
   http://www.digip.org/jansson/

 --with-hwloc=<directory>
   Installation directory of the hwloc library.
   http://www.open-mpi.org/projects/hwloc/
```

A small number of API unit tests and testing data have been made available as part of this distribution. To compile these tests use the following command:

```
shell$ make check
shell$ cd tests
# Run all of the programs compiled in this directory
```

1.3 Programming Interface

The netloc model separates network topology discovery mechanism from the mechanism for querying that network topology data via the netloc API. The reason for this separation is due to the need, for some networks, to run the discovery mechanism in a privileged mode.

Follow the link(s) below that best suit your intended use of netloc:

- Terms and Definitions (A good place to start)
- End-User API : For developers integrating netloc topology data into their application(s).
- Command Line Tools and Network Readers : For information on how to discover network topology data for your network.
- Reader (Data Collection) API : For developers interested in supporting a new type of network or extend support for existing networks in netloc.

1.4 Questions and Bugs

Questions should be sent to the netloc users and/or developers mailing list (`http://www.open-mpi.org/community/lists/netloc.php`).

Bug reports should be reported in the tracker (`https://git.open-mpi.org/trac/netloc/`).

Chapter 2

End-User API

There are a series of steps that a user must move through to gain access to the network topology information.

1. Run a netloc Reader tool to generate the .ndat file containing the network information (Command Line Tools and Network Readers). You will need to know the directory in which the .ndat files are contained.
2. Access Network Metadata

   This provides a lightweight discovery mechanism for choosing the network(s) about which to gather more detailed information.
3. Access the Network Topology Handle

   This opaque handle provides access to the detailed topology information.
4. Use the Network Topology Query Interfaces

   This interfaces allow you to access various components of the network topology including nodes, edges, and paths.

2.1 Network Metadata

The following interfaces allow the application to find available network information and choose the subset of those networks for further investigation. they

- netloc_find_network : Find a specific network
- netloc_foreach_network : Iterate through all available networks.

```
char **search_uris = NULL;
int num_uris = 1, ret;
netloc_network_t *tmp_network = NULL;

// Specify where to search for network data
search_uris = (char**)malloc(sizeof(char*) * num_uris );
search_uris[0] = strdup("file://data/");

// Find a specific InfiniBand network
tmp_network = netloc_dt_network_t_construct();
tmp_network->network_type = NETLOC_NETWORK_TYPE_INFINIBAND;
tmp_network->subnet_id    = strdup("fe80:0000:0000:0000");

// Search for the specific network
ret = netloc_find_network(search_uris[0], tmp_network);
if( NETLOC_SUCCESS != ret ) {
    fprintf(stderr, "Error: network not found!\n");
    netloc_dt_network_t_destruct(tmp_network);
    return NETLOC_ERROR;
}

printf("\tFound Network: %s\n", netloc_pretty_print_network_t(tmp_network));
```

```
// Cleanup (Do this only once finished querying the network)
netloc_dt_network_t_destruct(tmp_network);
tmp_network = NULL;
```

2.2 Network Topology Handle

The following interfaces attach a topology handle to a specific network discovered during the metadata discovery process (Network Metadata). they

- netloc_attach : Attach to a specific network.
- netloc_detach : Detach from the network.
- netloc_access_network_ref : Access the network handle associated with this topology.

(Note the code below is continued from the Network Metadata section.)

```
netloc_topology_t topology;

// Attach to the network
ret = netloc_attach(&topology, *tmp_network);
if( NETLOC_SUCCESS != ret ) {
    fprintf(stderr, "Error: netloc_attach returned an error (%d)\n", ret);
    return ret;
}

// Query the network topology (see next section, below)
// ...

// Detach from the network
ret = netloc_detach(topology);
if( NETLOC_SUCCESS != ret ) {
    fprintf(stderr, "Error: netloc_detach returned an error (%d)\n", ret);
    return ret;
}
```

2.3 Network Topology Query Interfaces

The following interfaces query the network topology using the netloc topology handle. they

- netloc_get_all_nodes : Access all of the nodes in the network topology.
- netloc_get_all_switch_nodes : Access only those nodes identified as switches.
- netloc_get_all_host_nodes : Access only those nodes identified as hosts.

- netloc_get_all_edges : Access all of the edges in the topology.
- netloc_get_node_by_physical_id : Find a node by their physical identifier.
- netloc_get_path : Access the physical or logical path between two nodes.

A few of these interfaces return a lookup table of information for collections of similar data types. The following functionality allows the user to tranverse this collection.

- netloc_dt_lookup_table_iterator_t_construct : Create an iterator for a lookup table.
- netloc_dt_lookup_table_iterator_t_destruct : Destroy a previously created iterator.
- netloc_lookup_table_destroy : Destroy a lookup table returned by the query API.
- netloc_lookup_table_size : Access the used size of the lookup table (number of entries).
- netloc_lookup_table_access : Access a specific entry in the table.
- netloc_lookup_table_iterator_next_key : Get the next key and advance the iterator.
- netloc_lookup_table_iterator_next_entry : Get the next entry and advance the iterator.
- netloc_lookup_table_iterator_at_end : Check if the iterator is at the end of the collection.
- netloc_lookup_table_iterator_reset : Reset the iterator to the beginning of the collection.

(Note the code below assumes a topology handle is attached, per Network Topology Handle section.)

```
netloc_topology_t topology;
// Assume that the ’topology’ handle is attached to a network.

netloc_dt_lookup_table_t nodes = NULL;
netloc_dt_lookup_table_iterator_t hti = NULL;
const char * key = NULL;
netloc_node_t *node = NULL;

// Access all of the nodes in the topology
ret = netloc_get_all_nodes(topology, &nodes);
if( NETLOC_SUCCESS != ret ) {
    fprintf(stderr, "Error: get_all_nodes returned %d\n", ret);
    return ret;
}
```

```
// Display all of the nodes found
hti = netloc_dt_lookup_table_iterator_t_construct( nodes );
while( !netloc_lookup_table_iterator_at_end(hti) ) {
    // Access the data by key (could also access by entry in the example)
    key = netloc_lookup_table_iterator_next_key(hti);
    if( NULL == key ) {
        break;
    }

    node = (netloc_node_t*)netloc_lookup_table_access(nodes, key);
    if( NETLOC_NODE_TYPE_INVALID == node->node_type ) {
        fprintf(stderr, "Error: Returned unexpected node: %s\n",
  netloc_pretty_print_node_t(node));
        return NETLOC_ERROR;
    }

    printf("Found: %s\n", netloc_pretty_print_node_t(node));
}

/* Cleanup */
netloc_dt_lookup_table_iterator_t_destruct(hti);
netloc_lookup_table_destroy(nodes);
free(nodes);
nodes = NULL;
```

2.4 Example Programs

The following small C example (named “netloc_hello.c”) accesses a specific network and searches for a specific node by its physical identifier (e.g., MAC address, GUID).

```
\textcolor{comment}{/*}
\textcolor{comment}{ * Copyright (c) 2013-2014 University of Wisconsin-La Crosse.}
\textcolor{comment}{ *                         All rights reserved.}
\textcolor{comment}{ *}
\textcolor{comment}{ * $COPYRIGHT$}
\textcolor{comment}{ *}
\textcolor{comment}{ * Additional copyrights may follow}
\textcolor{comment}{ * See COPYING in top-level directory.}
\textcolor{comment}{ *}
\textcolor{comment}{ * $HEADER$}
\textcolor{comment}{ *}
\textcolor{comment}{ * This program searches for a specific node in a specific network.}
\textcolor{comment}{ */}
\textcolor{preprocessor}{#include "netloc.h"}

\textcolor{keywordtype}{int} main(\textcolor{keywordtype}{void}) \{
    \textcolor{keywordtype}{char} **search\_uris = NULL;
    \textcolor{keywordtype}{int} num\_uris = 1, ret;
    \hyperlink{a00006}{netloc_network_t} *tmp\_network = NULL;

    \textcolor{comment}{// Specify where to search for network data}
    search\_uris = (\textcolor{keywordtype}{char}**)malloc(\textcolor{keyword}{sizeof}(\textcolor{keywordt
    search\_uris[0] = strdup(\textcolor{stringliteral}{"file://data/netloc"});
    \textcolor{comment}{// Find a specific InfiniBand network}
    tmp\_network = \hyperlink{a00013_ga495ee5817e6acb70ffb57b25c8b9acdb}{netloc_dt_network_t_c
    tmp\_network->\hyperlink{a00006_aa992ddb5f565d6e62f0a5dea6f3d03d3}{network_type} = \hyperli
    tmp\_network->\hyperlink{a00006_a248f35ff17f744331ff6351decc53083}{subnet_id}    = strdup(

    \textcolor{comment}{// Search for the specific network}
    ret = \hyperlink{a00013_ga2a09de16c27f7abc8301ae0ee8b9716e}{netloc_find_network}(search\_u
    \textcolor{keywordflow}{if}( \hyperlink{a00013_gga06fc87d81c62e9abb8790b6e5713c55ba4deb864
        fprintf(stderr, \textcolor{stringliteral}{"Error: network not found!\(\backslash\)n"})
        \hyperlink{a00013_ga3c9345d14e08d2fe0109590d49322895}{netloc_dt_network_t_destruct}(tm
        \textcolor{keywordflow}{return} \hyperlink{a00013_gga06fc87d81c62e9abb8790b6e5713c55ba
    \}

    printf(\textcolor{stringliteral}{"\(\backslash\)tFound Network: %s\(\backslash\)n"}, \hype

    \hyperlink{a00008}{netloc_topology_t} topology;

    \textcolor{comment}{// Attach to the network}
    ret = \hyperlink{a00013_gaf4046959469468de0422f1976a5c1480}{netloc_attach}(&topology, *tmp
    \textcolor{keywordflow}{if}( \hyperlink{a00013_gga06fc87d81c62e9abb8790b6e5713c55ba4deb864
        fprintf(stderr, \textcolor{stringliteral}{"Error: netloc\_attach returned an error (%d
        \textcolor{keywordflow}{return} ret;
    \}

    \textcolor{comment}{// Query the network topology}
    \textcolor{comment}{// Find a specific node by its physical ID (GUID in the case of Infini
    \hyperlink{a00007}{netloc_node_t} *node = NULL;
    \textcolor{keywordtype}{char} * phy\_id = strdup(\textcolor{stringliteral}{"000b:8cff:ff00
    node = \hyperlink{a00013_ga33c2f739f95d786e4a133454e713c79a}{netloc_get_node_by_physical_i
    \textcolor{keywordflow}{if}( \hyperlink{a00013_gga06fc87d81c62e9abb8790b6e5713c55ba4deb864
        fprintf(stderr, \textcolor{stringliteral}{"Error: netloc\_get\_node\_by\_physical\_id
      t);
        \textcolor{keywordflow}{return} ret;
    \}

    \textcolor{keywordflow}{if}( NULL == node ) \{
        printf(\textcolor{stringliteral}{"Did not find a node with the physical ID \(\backslas
    \} \textcolor{keywordflow}{else} \{
        printf(\textcolor{stringliteral}{"Found: %s\(\backslash\)n"}, \hyperlink{a00013_ga15be
    \}

    \textcolor{comment}{// Detach from the network}
    ret = \hyperlink{a00013_ga1bc063c4477a955290d15176268e9987}{netloc_detach}(topology);
    \textcolor{keywordflow}{if}( \hyperlink{a00013_gga06fc87d81c62e9abb8790b6e5713c55ba4deb864
        fprintf(stderr, \textcolor{stringliteral}{"Error: netloc\_detach returned an error (%d
        \textcolor{keywordflow}{return} ret;
    \}

    \textcolor{comment}{/*}
\textcolor{comment}{     * Cleanup}
\textcolor{comment}{     */}
    \textcolor{keywordflow}{if}( NULL != phy\_id ) \{
        free(phy\_id);
        phy\_id = NULL;
    \}

    \hyperlink{a00013_ga3c9345d14e08d2fe0109590d49322895}{netloc_dt_network_t_destruct}(tmp\_ne
    tmp\_network = NULL;

    \textcolor{keywordflow}{return} \hyperlink{a00013_gga06fc87d81c62e9abb8790b6e5713c55ba4deb864e4b46fe70
\}
```

The following C example (named “netloc_nodes.c”) is an accumulation of the inline examples above to display all of the nodes in a single network.

```
\textcolor{comment}{/*}
\textcolor{comment}{ * Copyright (c) 2013-2014 University of Wisconsin-La Crosse.}
\textcolor{comment}{ *                         All rights reserved.}
\textcolor{comment}{ *}
\textcolor{comment}{ * $COPYRIGHT$}
\textcolor{comment}{ *}
\textcolor{comment}{ * Additional copyrights may follow}
\textcolor{comment}{ * See COPYING in top-level directory.}
\textcolor{comment}{ *}
\textcolor{comment}{ * $HEADER$}
\textcolor{comment}{ *}
\textcolor{comment}{ * This program is meant to mirror the inline examples in netloc.doxy}
\textcolor{comment}{ */}
\textcolor{preprocessor}{#include "netloc.h"}

\textcolor{keywordtype}{int} main(\textcolor{keywordtype}{void}) \{
    \textcolor{keywordtype}{char} **search\_uris = NULL;
    \textcolor{keywordtype}{int} num\_uris = 1, ret;
    \hyperlink{a00006}{netloc_network_t} *tmp\_network = NULL;

    \hyperlink{a00008}{netloc_topology_t} topology;

    \hyperlink{a00003}{netloc_dt_lookup_table_t} nodes = NULL;
    \hyperlink{a00002}{netloc_dt_lookup_table_iterator_t} hti = NULL;
    \textcolor{keyword}{const} \textcolor{keywordtype}{char} * key = NULL;
    \hyperlink{a00007}{netloc_node_t} *node = NULL;


    \textcolor{comment}{// Specify where to search for network data}
    search\_uris = (\textcolor{keywordtype}{char}**)malloc(\textcolor{keyword}{sizeof}(\textcolor{keywordt
    search\_uris[0] = strdup(\textcolor{stringliteral}{"file://data/netloc"});

    \textcolor{comment}{// Find a specific InfiniBand network}
    tmp\_network = \hyperlink{a00013_ga495ee5817e6acb70ffb57b25c8b9acdb}{netloc_dt_network_t_construct}();
    tmp\_network->\hyperlink{a00006_aa992ddb5f565d6e62f0a5dea6f3d03d3}{network_type} = \hyperlink{a00013_g
    tmp\_network->\hyperlink{a00006_a248f35ff17f744331ff6351decc53083}{subnet_id}    = strdup(\textcolor{s

    \textcolor{comment}{// Search for the specific network}
    ret = \hyperlink{a00013_ga2a09de16c27f7abc8301ae0ee8b9716e}{netloc_find_network}(search\_uris[0], tmp\
    \textcolor{keywordflow}{if}( \hyperlink{a00013_gga06fc87d81c62e9abb8790b6e5713c55ba4deb864e4b46fe70b83
        fprintf(stderr, \textcolor{stringliteral}{"Error: network not found!\(\backslash\)n"});
        \hyperlink{a00013_ga3c9345d14e08d2fe0109590d49322895}{netloc_dt_network_t_destruct}(tmp\_network);
        \textcolor{keywordflow}{return} \hyperlink{a00013_gga06fc87d81c62e9abb8790b6e5713c55ba4b33f8da6ef
    \}

    printf(\textcolor{stringliteral}{"\(\backslash\)tFound Network: %s\(\backslash\)n"}, \hyperlink{a00013

    \textcolor{comment}{// Attach to the network}
    ret = \hyperlink{a00013_gaf4046959469468de0422f1976a5c1480}{netloc_attach}(&topology, *tmp
    \textcolor{keywordflow}{if}( \hyperlink{a00013_gga06fc87d81c62e9abb8790b6e5713c55ba4deb864e
        fprintf(stderr, \textcolor{stringliteral}{"Error: netloc\_attach returned an error (%d
        \textcolor{keywordflow}{return} ret;
    \}

    \textcolor{comment}{// Query the network topology}

    \textcolor{comment}{// Access all of the nodes in the topology}
    ret = \hyperlink{a00013_gaa1993053ddd68a59dd2bae49b0165815}{netloc_get_all_nodes}(topology,
    \textcolor{keywordflow}{if}( \hyperlink{a00013_gga06fc87d81c62e9abb8790b6e5713c55ba4deb864e
        fprintf(stderr, \textcolor{stringliteral}{"Error: get\_all\_nodes returned %d\(\backsla
        \textcolor{keywordflow}{return} ret;
    \}

    \textcolor{comment}{// Display all of the nodes found}
    hti = \hyperlink{a00013_ga475d024569e4e5d1734db3f496642097}{netloc_dt_lookup_table_iterator
    \textcolor{keywordflow}{while}( !\hyperlink{a00013_ga0b31195dc6ac77e33c2cb8a14aebc975}{netl
        \textcolor{comment}{// Access the data by key (could also access by entry in the exampl
        key = \hyperlink{a00013_gaa009bb37d5f6c61acf8c9ef2403e16d3}{netloc_lookup_table_iterato
        \textcolor{keywordflow}{if}( NULL == key ) \{
            \textcolor{keywordflow}{break};
        \}

        node = (\hyperlink{a00007}{netloc_node_t}*)\hyperlink{a00013_ga7fb2f9859e47e1706560890
        \textcolor{keywordflow}{if}( \hyperlink{a00013_gga2f3adc0994f3d3ed0d48e1f235bed020a49e
            fprintf(stderr, \textcolor{stringliteral}{"Error: Returned unexpected node: %s\(\ba
      \hyperlink{a00013_ga15becaec94159a6bab9ac19da06cb4d3}{netloc_pretty_print_node_t}(node))
            \textcolor{keywordflow}{return} \hyperlink{a00013_gga06fc87d81c62e9abb8790b6e5713c
        \}

        printf(\textcolor{stringliteral}{"Found: %s\(\backslash\)n"}, \hyperlink{a00013_ga15bec
    \}

    \textcolor{comment}{/* Cleanup the lookup table objects */}
    \textcolor{keywordflow}{if}( NULL != hti ) \{
        \hyperlink{a00013_ga62383c246b9ea1c372b04b15dd726fab}{netloc_dt_lookup_table_iterator_t
        hti = NULL;
    \}
    \textcolor{keywordflow}{if}( NULL != nodes ) \{
        \hyperlink{a00013_ga91eb820e034f959919189b35dbaae070}{netloc_lookup_table_destroy}(node
        free(nodes);
        nodes = NULL;
    \}

    \textcolor{comment}{// Detach from the network}
    ret = \hyperlink{a00013_ga1bc063c4477a955290d15176268e9987}{netloc_detach}(topology);
    \textcolor{keywordflow}{if}( \hyperlink{a00013_gga06fc87d81c62e9abb8790b6e5713c55ba4deb864e
        fprintf(stderr, \textcolor{stringliteral}{"Error: netloc\_detach returned an error (%d
        \textcolor{keywordflow}{return} ret;
    \}

    \textcolor{comment}{/*}
\textcolor{comment}{     * Cleanup}
\textcolor{comment}{     */}
    \hyperlink{a00013_ga3c9345d14e08d2fe0109590d49322895}{netloc_dt_network_t_destruct}(tmp\_ne
    tmp\_network = NULL;
    \textcolor{keywordflow}{return} \hyperlink{a00013_gga06fc87d81c62e9abb8790b6e5713c55ba4deb864e4b46fe70
\}
```

The following small C example (named “netloc_all.c”) prints all of the nodes in all of the network topologies discovered.

```
\textcolor{comment}{/*}
\textcolor{comment}{ * Copyright (c) 2013-2014 University of Wisconsin-La Crosse.}
\textcolor{comment}{ *                         All rights reserved.}
\textcolor{comment}{ *}
\textcolor{comment}{ * $COPYRIGHT$}
\textcolor{comment}{ *}
\textcolor{comment}{ * Additional copyrights may follow}
\textcolor{comment}{ * See COPYING in top-level directory.}
\textcolor{comment}{ *}
\textcolor{comment}{ * $HEADER$}
\textcolor{comment}{ *}
\textcolor{comment}{ * This program prints all of the nodes in all of the network topologies discover
      ed.}
\textcolor{comment}{ */}
\textcolor{preprocessor}{#include "netloc.h"}

\textcolor{keywordtype}{int} main(\textcolor{keywordtype}{void}) \{
    \textcolor{keywordtype}{int} i, num\_uris = 1;
    \textcolor{keywordtype}{char} **search\_uris = NULL;
    \textcolor{keywordtype}{int} ret, exit\_status = \hyperlink{a00013_gga06fc87d81c62e9abb8790b6e5713c55b
    \textcolor{keywordtype}{int} num\_all\_networks = 0;
    \hyperlink{a00006}{netloc_network_t} **all\_networks = NULL;

    \hyperlink{a00008}{netloc_topology_t} topology;

    \hyperlink{a00003}{netloc_dt_lookup_table_t} nodes = NULL;
    \hyperlink{a00002}{netloc_dt_lookup_table_iterator_t} hti = NULL;
    \hyperlink{a00007}{netloc_node_t} *node = NULL;

    \textcolor{comment}{/*}
\textcolor{comment}{     * Where to search for network topology information.}
\textcolor{comment}{     * Information generated from a netloc reader.}
\textcolor{comment}{     */}
    search\_uris = (\textcolor{keywordtype}{char}**)malloc(\textcolor{keyword}{sizeof}(\textcolor{keywordt
    \textcolor{keywordflow}{if}( NULL == search\_uris ) \{
        \textcolor{keywordflow}{return} \hyperlink{a00013_gga06fc87d81c62e9abb8790b6e5713c55ba4b33f8da6efc
    \}
    search\_uris[0] = strdup(\textcolor{stringliteral}{"file://data/netloc"});

    \textcolor{comment}{/*}
\textcolor{comment}{     * Find all of the networks in the specified serach URI locations}
\textcolor{comment}{     */}
    ret = \hyperlink{a00013_gae37494d22fada025bea1f568b4fb09c1}{netloc_foreach_network}((\textcolor{keywor
                            num\_uris,
                            NULL, \textcolor{comment}{// Callback function (NULL = include all n
      etworks)}
                            NULL, \textcolor{comment}{// Callback function data}
                            &num\_all\_networks,
                            &all\_networks);
    \textcolor{keywordflow}{if}( \hyperlink{a00013_gga06fc87d81c62e9abb8790b6e5713c55ba4deb864e
        fprintf(stderr, \textcolor{stringliteral}{"Error: netloc\_foreach\_network returned an
       ret);
        exit\_status = ret;
        \textcolor{keywordflow}{goto} cleanup;
    \}

    \textcolor{comment}{/*}
\textcolor{comment}{     * For each of those networks access the detailed topology}
\textcolor{comment}{     */}
    \textcolor{keywordflow}{for}(i = 0; i < num\_all\_networks; ++i ) \{
        \textcolor{comment}{// Pretty print the network for debugging purposes}
        printf(\textcolor{stringliteral}{"\(\backslash\)tIncluded Network: %s\(\backslash\)n"},
      orks[i]) );

        \textcolor{comment}{/*}
\textcolor{comment}{         * Attach to the network}
\textcolor{comment}{         */}
        ret = \hyperlink{a00013_gaf4046959469468de0422f1976a5c1480}{netloc_attach}(&topology,
        \textcolor{keywordflow}{if}( \hyperlink{a00013_gga06fc87d81c62e9abb8790b6e5713c55ba4deb
            fprintf(stderr, \textcolor{stringliteral}{"Error: netloc\_attach returned an error
      ;
            \textcolor{keywordflow}{return} ret;
        \}

        \textcolor{comment}{/*}
\textcolor{comment}{         * Access all of the nodes in the topology}
\textcolor{comment}{         */}
        ret = \hyperlink{a00013_gaa1993053ddd68a59dd2bae49b0165815}{netloc_get_all_nodes}(topol
        \textcolor{keywordflow}{if}( \hyperlink{a00013_gga06fc87d81c62e9abb8790b6e5713c55ba4deb
            fprintf(stderr, \textcolor{stringliteral}{"Error: get\_all\_nodes returned %d\(\bac
            \textcolor{keywordflow}{return} ret;
        \}

        \textcolor{comment}{// Display all of the nodes found}
        hti = \hyperlink{a00013_ga475d024569e4e5d1734db3f496642097}{netloc_dt_lookup_table_iter
        \textcolor{keywordflow}{while}( !\hyperlink{a00013_ga0b31195dc6ac77e33c2cb8a14aebc975}
            node = \hyperlink{a00013_ga738c5312136df22759a3df0ac2e6b403}{netloc_lookup_table_it
            \textcolor{keywordflow}{if}( NULL == node ) \{
                \textcolor{keywordflow}{break};
            \}
            \textcolor{keywordflow}{if}( \hyperlink{a00013_gga2f3adc0994f3d3ed0d48e1f235bed020a
                fprintf(stderr, \textcolor{stringliteral}{"Error: Returned unexpected node: %s
      \hyperlink{a00013_ga15becaec94159a6bab9ac19da06cb4d3}{netloc_pretty_print_node_t}(node));
                \textcolor{keywordflow}{return} \hyperlink{a00013_gga06fc87d81c62e9abb8790b6e5
            \}

            printf(\textcolor{stringliteral}{"Found: %s\(\backslash\)n"}, \hyperlink{a00013_ga1
        \}

        \textcolor{comment}{/* Cleanup the lookup table objects */}
        \textcolor{keywordflow}{if}( NULL != hti ) \{
            \hyperlink{a00013_ga62383c246b9ea1c372b04b15dd726fab}{netloc_dt_lookup_table_iterat
            hti = NULL;
        \}
        \textcolor{keywordflow}{if}( NULL != nodes ) \{
            \hyperlink{a00013_ga91eb820e034f959919189b35dbaae070}{netloc_lookup_table_destroy}
            free(nodes);
            nodes = NULL;
        \}

        \textcolor{comment}{/*}
\textcolor{comment}{         * Detach from the network}
\textcolor{comment}{         */}
        ret = \hyperlink{a00013_ga1bc063c4477a955290d15176268e9987}{netloc_detach}(topology);
        \textcolor{keywordflow}{if}( \hyperlink{a00013_gga06fc87d81c62e9abb8790b6e5713c55ba4deb864e4b46fe7
            fprintf(stderr, \textcolor{stringliteral}{"Error: netloc\_detach returned an error (%d)\(\back
      ;
            \textcolor{keywordflow}{return} ret;
        \}
    \}

    \textcolor{comment}{/*}
\textcolor{comment}{     * Cleanup}
\textcolor{comment}{     */}
 cleanup:
    \textcolor{keywordflow}{if}( NULL != hti ) \{
        \hyperlink{a00013_ga62383c246b9ea1c372b04b15dd726fab}{netloc_dt_lookup_table_iterator_t_destruct}(
        hti = NULL;
    \}
    \textcolor{keywordflow}{if}( NULL != nodes ) \{
        \hyperlink{a00013_ga91eb820e034f959919189b35dbaae070}{netloc_lookup_table_destroy}(nodes);
        free(nodes);
        nodes = NULL;
    \}

    \textcolor{keywordflow}{if}( NULL != all\_networks ) \{
        \textcolor{keywordflow}{for}(i = 0; i < num\_all\_networks; ++i ) \{
            \hyperlink{a00013_ga3c9345d14e08d2fe0109590d49322895}{netloc_dt_network_t_destruct}(all\_netwo
            all\_networks[i] = NULL;
        \}
        free(all\_networks);
        all\_networks = NULL;
    \}

    \textcolor{keywordflow}{if}( NULL != search\_uris ) \{
        \textcolor{keywordflow}{for}(i = 0; i < num\_uris; ++i) \{
            free(search\_uris[i]);
            search\_uris[i] = NULL;
        \}
        free(search\_uris);
        search\_uris = NULL;
    \}

    \textcolor{keywordflow}{return} \hyperlink{a00013_gga06fc87d81c62e9abb8790b6e5713c55ba4deb864e4b46fe70
\}
```

Chapter 3

Command Line Tools and Network Readers

3.1 lsnettopo

The `lsnettopo` command provides a description of the network information discovered. This command will list the network topology summary information for all networks in the specified directory. The network topology information is displayed to the console or can be exported in a variety of formats (e.g., GraphML and GEXF file formats) providing developers with an additional mechanism to access the data for further analysis.

```
shell$ lsnettopo data/
Network: ETH-unknown
  Type    : Ethernet
  Subnet  : unknown
  Hosts   :   8
  Switches:   7
---------------------------------------------------
shell$
shell$ lsnettopo data/ --export gexf
Network: ETH-unknown
         Filename: ETH-unknown.gexf
shell$
shell$ lsnettopo data/ -f
Network: ETH-unknown
  Type    : Ethernet
  Subnet  : unknown
  Hosts   :   8
  Switches:   7
---------------------------------------------------

Information by Host
---------------------
00:00:00:00:00:02 (  Host) on port  -1  [-> 1/1 <-]  00:00:00:00:00:00:00:03 (Switch) on port
00:00:00:00:00:07 (  Host) on port  -1  [-> 1/1 <-]  00:00:00:00:00:00:00:07 (Switch) on port
00:00:00:00:00:03 (  Host) on port  -1  [-> 1/1 <-]  00:00:00:00:00:00:00:04 (Switch) on port
00:00:00:00:00:06 (  Host) on port  -1  [-> 1/1 <-]  00:00:00:00:00:00:00:06 (Switch) on port
00:00:00:00:00:08 (  Host) on port  -1  [-> 1/1 <-]  00:00:00:00:00:00:00:07 (Switch) on port
00:00:00:00:00:05 (  Host) on port  -1  [-> 1/1 <-]  00:00:00:00:00:00:00:06 (Switch) on port
00:00:00:00:00:01 (  Host) on port  -1  [-> 1/1 <-]  00:00:00:00:00:00:00:03 (Switch) on port
00:00:00:00:00:04 (  Host) on port  -1  [-> 1/1 <-]  00:00:00:00:00:00:00:04 (Switch) on port

Information by Switch
---------------------
00:00:00:00:00:00:00:06 (Switch) on port   3  [-> 10000000000/1 <-]  00:00:00:00:00:00:00:05 (
00:00:00:00:00:00:00:06 (Switch) on port   2  [-> 1/1 <-]  00:00:00:00:00:06 (  Host) on port
00:00:00:00:00:00:00:06 (Switch) on port   1  [-> 1/1 <-]  00:00:00:00:00:05 (  Host) on port
00:00:00:00:00:00:00:03 (Switch) on port   2  [-> 1/1 <-]  00:00:00:00:00:02 (  Host) on port
00:00:00:00:00:00:00:03 (Switch) on port   3  [-> 10000000000/1 <-]  00:00:00:00:00:00:00:02 (
00:00:00:00:00:00:00:03 (Switch) on port   1  [-> 1/1 <-]  00:00:00:00:00:01 (  Host) on port
00:00:00:00:00:00:00:07 (Switch) on port   1  [-> 1/1 <-]  00:00:00:00:00:07 (  Host) on port
00:00:00:00:00:00:00:07 (Switch) on port   2  [-> 1/1 <-]  00:00:00:00:00:08 (  Host) on port
00:00:00:00:00:00:00:07 (Switch) on port   3  [-> 10000000000/1 <-]  00:00:00:00:00:00:00:05 (
00:00:00:00:00:00:00:02 (Switch) on port   2  [-> 10000000000/1 <-]  00:00:00:00:00:00:00:04 (
00:00:00:00:00:00:00:02 (Switch) on port   1  [-> 10000000000/1 <-]  00:00:00:00:00:00:00:03 (
00:00:00:00:00:00:00:02 (Switch) on port   3  [-> 10000000000/1 <-]  00:00:00:00:00:00:00:01 (
00:00:00:00:00:00:00:04 (Switch) on port   1  [-> 1/1 <-]  00:00:00:00:00:03 (  Host) on port
```

```
00:00:00:00:00:00:00:04 (Switch) on port   3  [-> 10000000000/1 <-]  00:00:00:00:00:00:00:02 (Switch) on p
00:00:00:00:00:00:00:04 (Switch) on port   2  [-> 1/1 <-]  00:00:00:00:00:04 (  Host) on port  -1
00:00:00:00:00:00:00:05 (Switch) on port   3  [-> 10000000000/1 <-]  00:00:00:00:00:00:00:01 (Switch) on p
00:00:00:00:00:00:00:05 (Switch) on port   1  [-> 10000000000/1 <-]  00:00:00:00:00:00:00:06 (Switch) on p
00:00:00:00:00:00:00:05 (Switch) on port   2  [-> 10000000000/1 <-]  00:00:00:00:00:00:00:07 (Switch) on p
00:00:00:00:00:00:00:01 (Switch) on port   1  [-> 10000000000/1 <-]  00:00:00:00:00:00:00:02 (Switch) on p
00:00:00:00:00:00:00:01 (Switch) on port   2  [-> 10000000000/1 <-]  00:00:00:00:00:00:00:05 (Switch) on p
------------------------------------------------------------------------------
```

3.1.1 Command Line Interface

There are a few command line options available. See `lsnettopo --help` for a complete list.

```
<input directory>             (Optional)
   Path to directory where the netloc .dat files are placed.
   Detected as the first unknown option on the command line
   Default: ./

--full | -f                   (Optional)
   Full output, in addition to the brief overview
   Default: disabled

--export | -e <export_type>   (Optional)
   Export the data in the specified format type.
   Supported Format Types
     screen  (Default)
        Display to the terminal
     GraphML
        File extension .graphml
        http://graphml.graphdrawing.org/
     GEXF
        File extension .gexf
        http://gexf.net/

--verbose | -v                (Optional)
   Verbose output.

--help | -h                   (Optional)
   Display a help message.
```

3.2 Reader: Static (User Defined)

The `netloc_reader_static` tool processes data from a user supplied input file. This reader is useful for users on networks that do not (yet) have a netloc reader or, due to restricted access, the user cannot run the necessary reader. The format of the input file is specified below.

- `netloc_reader_static` : Static specification of the network topology information.

```
shell$ netloc_reader_static -i example-2-nodes.json -o netloc/
  Input file       : example-2-nodes.json
  Output Directory : ./
Parsing the input file...
        Processing: Network Information...
        Processing: Node Information...
        Processing: Edge Information...
Status: Computing Physical Paths
Status: Validating the output...
        Number of hosts    :    2
        Number of switches:    1
        Number of edges    :    4
shell$
shell$  lsnettopo netloc/
Network: ETH-unknown (version 1)
  Type    : Ethernet
  Subnet  : unknown
  Hosts   :   2
  Switches:   1
---------------------------------------------------
```

3.2.1 Command Line Interfaces (netloc_reader_static)

There are a few command line options available. See `netloc_reader_static --help` for a complete list.

```
--input | -i <filename>
   The JSON input file describing the network nodes and edges.

--outdir | -o <output directory>       (Optional)
   Path to directory where output .dat filess are placed by the tool.
   Default: "./"

--progress | -p
   Show progress of processing node and edge information
   Default: disabled

--help | -h                            (Optional)
   Display a help message.
```

3.2.2 JSON Format

Below is the `JSON schema` for the input file.

Below is an example of the expected format of the JSON input file with two nodes and one swich.

3.3 Reader: InfiniBand

The following tools are available for discovering the network topology of an InfiniBand network.

- `netloc_ib_gather_raw` : Call the `ibnetdiscover` and `ibroutes` tools to generate the necessary raw data files.
- `netloc_ib_extract_dats` : This command simplifies the use of the `netloc_reader_ib` tool by processing all subnet data generated from the `netloc_ib_gather_raw` tool..
- `netloc_reader_ib` : Processes raw data from the `ibnetdiscover` and `ibroutes` tools. The resulting .ndat files are used as abstract representations of the network graph

```
Normal way to use this:
* Get some hwloc outputs from some nodes (at least enough nodes to make all subnets available)
  and store them as <hostname>.xml in a single directory
  shell$ ssh node001 lstopo ~/mycluster-data/hwloc/node001.xml

* Run netloc-ib-gather-raw.pl --hwloc-dir <hwloc XML directory> --raw-dir <raw IB output directory>
  - If you cannot run the entire script as root, add --sudo to run ib* programs as root.
  - If some subnets are not accessible from the local node, they will be skipped.
    Add --verbose to see where you could run the same command to discover other subnets.
  - If one subnet doesn't work for some reason, use --force-subnet instead of --hwloc-dir.

* Make sure netloc_ib_reader and friends are in PATH

* Run netloc-ib-extract-dats.pl --raw-dir <output directory> --out-dir <netloc output directory>
```

Example using `netloc_ib_gather_raw` and `netloc_ib_extract_dats`:

```
shell$ netloc_ib_gather_raw --hwloc-dir hwloc/ --raw-dir ib-raw/
shell$
shell$ netloc_ib_extract_dats --raw-dir ib-raw --out-dir netloc
---------------------------------------------------------------------
Processing Subnet: 3333:3333:3333:3333
---------------------------------------------------------------------
--------------- General Network Information
---------------------------------------------------------------------
Processing Subnet: 2222:2222:2222:2222
---------------------------------------------------------------------
--------------- General Network Information
shell$
shell$  lsnettopo netloc/
Network: IB-2222:2222:2222:2222
  Type    : InfiniBand
  Subnet  : 2222:2222:2222:2222
  Hosts   :   38
  Switches:   12
---------------------------------------------------
```

```
Network: IB-3333:3333:3333:3333
  Type    : InfiniBand
  Subnet  : 3333:3333:3333:3333
  Hosts   :   27
  Switches:   18
---------------------------------------------------
```

Example using `netloc_ib_gather_raw` and `netloc_reader_ib` to only process one of the subnets.

```
shell$ netloc-ib-gather-raw.pl --hwloc-dir hwloc/ --raw-dir ib-raw/
shell$
shell$ netloc_reader_ib --subnet 2222:2222:2222:2222 \
            --outdir dat_files/ \
            --file ib-raw/ib-subnet-2222\:2222\:2222\:2222.txt \
            --routedir ib-raw/ibroutes-2222\:2222\:2222\:2222/
  Output Directory   : dat_files/
  Subnet             : 2222:2222:2222:2222
  ibnetdiscover File : ib-raw/ib-subnet-2222:2222:2222:2222.txt
  ibroutes Directory : ib-raw/ibroutes-2222:2222:2222:2222/
Status: Querying the ibnetdiscover data for subnet 2222:2222:2222:2222...
Status: Processing Node Information
Status: Computing Physical Paths
Status: Querying the ibroutes data for subnet 2222:2222:2222:2222...
Status: Processing Logical Paths
Status: Validating the output...
        Number of hosts    :   38
        Number of switches:   12
        Number of edges    :  220
shell$
shell$  lsnettopo dat_files/
Network: IB-2222:2222:2222:2222
  Type    : InfiniBand
  Subnet  : 2222:2222:2222:2222
  Hosts   :   38
  Switches:   12
---------------------------------------------------
```

3.3.1 Command Line Interfaces (netloc_ib_gather_raw)

There are a few command line options available. See `netloc_ib_gather_raw` for a complete list.

```
Output directory for raw IB data must be specified with
 --out-dir <dir>

Input must be one of these
 --hwloc-dir <dir>
    Specifies that <dir> contains the hwloc XML exports of the some nodes,
    The list of IB subnets should be guessed from there.

 --force-subnet [<subnet>:]<board>:<port> to force the discovery
```

```
    Force discovery on local board <board> port <port>, and optionally force the
    subnet id <subnet> instead of reading it from the first GID.
    Examples: --force-subnet mlx4_0:1
              --force-subnet fe80:0000:0000:0000:mlx4_0:1

Other options
 --sudo
    Pass sudo to internal ibnetdiscover and ibroute invocations.
    Useful when the entire script cannot run as root.

 --ibnetdiscover --ibroute
    Specify exact location of programs. Default is /usr/bin/<program>

 --ignore-errors
    Ignore errors from ibnetdiscover and ibroute, assume their outputs are ok

 --verbose
    Add verbose messages

 --dry-run
    Do not actually run programs
```

3.3.2 Command Line Interfaces (netloc_ib_extract_dats)

There are a few command line options available. See `netloc_ib_extract_dats --help` for a complete list.

```
 --raw-dir <dir>                (Optional)
   Input directory with raw IB data must be specified with
   Default is ./ib-raw

 --out-dir <dir>                (Optional)
   Output directory for netloc data can be specified with
   Default is ./netloc

 --verbose | -v                 (Optional)
   Verbose and progress information

 --help | -h                    (Optional)
   Display a help message.
```

3.3.3 Command Line Interfaces (netloc_reader_ib)

There are a few command line options available. See `netloc_reader_ib --help` for a complete list.

```
--file <input file>
   The file containing the ibnetdiscover data

--routedir <path to routing files>  (Optional)
   Path to the file containing ibroutes data.
   Information for each host should be stored in a separate file.
   Default: Exclude logical routing information

--subnet <subnet id>
   The subset id of the network

--outdir <output directory>         (Optional)
   Path to directory where output .dat files are placed.
   Default: ./

--progress | -p                     (Optional)
   Display a progress percentage while processing the network files.

--help | -h                         (Optional)
   Display a help message.
```

3.4 Reader: OpenFlow-managed Ethernet

The `netloc_reader_of` tool processes data from a supported OpenFlow controller to discover information about an Ethernet network. The controller must be running and reachable from the machine running this tool.

- `netloc_reader_of` : Contact the OpenFlow controller and extract the network topology information.

```
shell$ netloc_reader_of --controller opendaylight -o netloc/
shell$
shell$  lsnettopo netloc/
Network: ETH-unknown
  Type    : Ethernet
  Subnet  : unknown
  Hosts   :   8
  Switches:   7
---------------------------------------------------
```

3.4.1 Command Line Interfaces (netloc_reader_of)

There are a few command line options available. See `netloc_reader_of --help` for a complete list.

```
--controller | -c <cname>
   Name of the controller to use to access the OpenFlow network
   information. See below for options.
   Supported Controllers
      opendaylight:
         Attach to the OpenDaylight controller for network information.

      floodlight:
         Attach to the Floodlight controller for network information.
      xnc:
         Attach to the Cisco XNC controller for network information.

--subnet | -s <subnet id>              (Optional)
   The subnet id of the network
   Default: "unknown"

--outdir | -o <output directory>       (Optional)
   Path to directory where output .dat filess are placed by the tool.
   Default: "./"

--addr | -a <IP Address:Port>          (Optional)
   IP address and port of the controller
   Default: 127.0.0.1:8080

--username | -u <username>             (Optional)
   Username for authorization to the controller
   Default: <none>

--password | -p <password>             (Optional)
   Password for authorization to the controller
   Default: <none>

--help | -h                            (Optional)
   Display a help message.
```

Chapter 4

Reader (Data Collection) API

There are a series of steps that a developer will need to go through to create a new netloc reader. The basic steps are below.

1. Access the node and edge information from your network. The remainder of this section assumes that you have this information and are trying to convert it into netloc .ndat files.
2. Setup Network Information :

   Setup the network information on the netloc_network_t handle.
3. Setup Data Collection Handle :

   Setup a data collection handle (netloc_data_collection_handle_t) associated with that network along with the output directory for the .ndat files.
4. Create netloc Nodes :

   Create a netloc_node_t object for each addressable network endpoint (e.g., MAC Address, GUID) in the system.
5. Add the Node to the collection :

   Add the netloc_node_t object to the collection.
6. Create edges between the nodes :

   Create a netloc_edge_t object for each unidirectional edge between netloc_node_t objects.
7. Add the edges to the collection :

   Add the netloc_edge_t object to the source netloc_node_t object.
8. (Optional) Physical Paths :

   Compute the shortest physical paths between all nodes.
9. (Optional) Logical Paths :

   Append logical paths between all nodes.
10. Close the data collection :

    Close the data collection handle to write the data to the .ndat files in the specified output directory.

4.1 Setup Network Information

The following interfaces are useful in setting up the network information. Note that the network information pertains to a single network type and subnet. they

- netloc_network_t : Network handle.

- netloc_dt_network_t_construct : Constructor for the network handle.
- netloc_dt_network_t_destruct : Destructor for the network handle.

```
netloc_network_t *network = NULL;

network = netloc_dt_network_t_construct();

network->network_type = NETLOC_NETWORK_TYPE_ETHERNET;
network->subnet_id    = strdup("unknown");
network->version      = strdup("1");
network->description  = strdup("This is an example");
network->data_uri     = strdup("file://.");
```

4.2 Setup Data Collection Handle

The following interfaces are useful in setting up the data collection handle and associating it with the network information. they

- netloc_data_collection_handle_t : Data Collection handle.
- netloc_dc_create : Creates the handle and associates it with the specified network.

```
netloc_data_collection_handle_t *dc_handle = NULL;

dc_handle = netloc_dc_create(network, outdir);

// After which the network can be destructed
netloc_dt_network_t_destruct(network);
network = NULL;
```

4.3 Create netloc Nodes

The following interfaces are useful in creating a netloc_node_t for each addressable network endpoing (e.g., MAC address, GUID). they

- netloc_node_t : Node type.
- netloc_dt_node_t_construct : Constructor for the node type.
- netloc_dt_node_t_destruct : Destructor for the node type.
- netloc_encode_node_type : Encode the node type (e.g., Switch, Host)

```
netloc_node_t *node = NULL;

node = netloc_dt_node_t_construct();

// fill in the necessary fields. For example,
node->network_type = dc_handle->network->network_type;
node->subnet_id    = strdup(dc_handle->network->subnet_id);
node->node_type    = netloc_encode_node_type("CA")
node->logical_id   = strdup("10.0.0.2");
node->physical_id  = strdup("00:00:00:00:00:02");
node->description  = strdup("eth0 on node02")
```

4.4 Add the Node to the collection

The following interfaces are useful in adding the netloc_node_t object to the data collection. they

- netloc_dc_append_node : Append the neloc node to the collection.

```
netloc_dc_append_node(dc_handle, node);

// You can destroy the node, since it is copied internally
netloc_dt_node_t_destruct(node);
```

4.5 Create edges between the nodes

The following interfaces are useful in creating netloc_edge_t objects representing the links between nodes. they

- netloc_edge_t : Netloc edge type.
- netloc_dt_edge_t_construct : Edge constructor.

```
netloc_edge_t *edge = NULL;

edge = netloc_dt_edge_t_construct();

// fill in the necessary fields. For example,
edge->src_node_id     = strdup(node->physical_id);
edge->src_node_type   = NETLOC_NODE_TYPE_HOST;
edge->src_port_id     = strdup("-1");

edge->dest_node_id    = strdup(dst_node->physical_id);
edge->dest_node_type  = NETLOC_NODE_TYPE_SWITCH;
edge->dest_port_id    = strdup("5");

edge->speed           = strdup("100000");
edge->width           = strdup("1");
edge->description     = strdup("node02 to switch");
```

4.6 Add the edges to the collection

The following interfaces are useful in adding the netloc_edge_t objects to the data collection. Note that the edge is always associated with the source node.

- netloc_dc_get_node_by_physical_id : Access a reference to the stored netloc_node_t object with the specified physical identifier.
- netloc_dc_append_edge_to_node : Add the edge to the node in the data collection.

```
// You need the node reference of the source node to attach the edge.
// We always attach the edge to the source node which can always be
// accessed by its physical ID
node = netloc_dc_get_node_by_physical_id(dc_handle, "00:00:00:00:00:02");

// Now add the edge to the node on the handle
netloc_dc_append_edge_to_node(dc_handle, node, edge);

// You can destroy the edge, since it is copied internally
netloc_dt_edge_t_destruct(edge);
```

4.7 (Optional) Physical Paths

The following interfaces are useful in constructing the physical path information between nodes. Note that all of the nodes and edges must be attached to the data collection handle before this will work.

- netloc_dc_compute_path_between_nodes : Compute the physical paths between the two nodes specified. Final parameter is "false" indicating a physical path computation.
- netloc_dc_append_path : Append a path between two nodes to the data collection. Final parameter is "false" indicating a physical path addition.

```
netloc_node_t *cur_src_node = NULL;
netloc_node_t *cur_dst_node = NULL;
int num_edges = 0;
netloc_edge_t **edges = NULL;

// Access the node objects
cur_src_node = netloc_dc_get_node_by_physical_id(dc_handle, "00:00:00:00:00:01
   ");
cur_dst_node = netloc_dc_get_node_by_physical_id(dc_handle, "00:00:00:00:00:02
   ");

// Use the netloc library to compute the physical paths between nodes
netloc_dc_compute_path_between_nodes(dc_handle,
                                     cur_src_node,
                                     cur_dst_node,
                                     &num_edges,
                                     &edges,
                                     false);

 // Store the path on the data collection handle
 netloc_dc_append_path(dc_handle,
                       cur_src_node->physical_id,
                       cur_dst_node->physical_id,
                       num_edges,
                       edges,
                       false);

 // Cleanup
 num_edges = 0;
 free(edges);
 edges = NULL;
```

4.8 (Optional) Logical Paths

The following interfaces are useful in constructing the logical path information between nodes.

Todo

JJH Logical paths have not been well tested.

- netloc_dc_compute_path_between_nodes : Compute the physical paths between the two nodes specified. Final parameter is "ture" indicating a logical path computation.
- netloc_dc_append_path : Append a path between two nodes to the data collection. Final parameter is "true" indicating a logical path addition.

```
// TODO
```

4.9 Close the data collection

The following interfaces are useful in closing the data collection. This will write the data to the .ndat files in the directory specified on the data collection handle.

- netloc_dc_close : Close the data collection and write out the netloc .ndat files.

```
// Close the data collection
netloc_dc_close(dc_handle);

// Cleanup the handle when we are finished with it.
netloc_dt_data_collection_handle_t_destruct(dc_handle);
dc_handle = NULL;
```

Chapter 5

Terms and Definitions

netloc network handle (netloc_network_t) Represents a lightweight handle to a single network subnet at a single point in time. It is from this handle that the user can access metadata about the network and create a netloc topology handle (netloc_topology_t).

This handle can be thought of as a tuple of information: network type, network subnet, and version/timestamp.

netloc topology handle (netloc_topology_t) An opaque data structure containing detailed network topology information. This handle is used by all of the network topology query APIs.

netloc node (netloc_node_t) Represents the concept of a node (a.k.a., vertex, endpoint) within a network graph. This could be a server NIC or a network switch.

If a server has more than one NIC then there are multiple netloc nodes for this server, one for each NIC. This is because some networks cannot distinguish node boundaries. In order to group multiple netloc nodes together into a logical server the netloc topology data will need to to be mapped with the hwloc data using the map API.

netloc edge (netloc_edge_t) Represents the concept of a directed edge withing a network graph. These are the physical connections between two netloc nodes (netloc_node_t).

Physical Path (netloc_node_t::physical_paths) Represents the shortest physical path from one netloc node to another. This path does not take into account higher level routing rules that might be in place in the network. The path is represented as a series of 'hops' through the network where each 'hop' is a netloc_edge_t object (from which you can access the source and destination netloc_node_t).

Path information is only calculated between servers, not between switches in the network.

Logical Path (netloc_node_t::logical_paths) Represents the logical path from one netloc node to another. This path takes into account the higher level routing rules that are in place in the network. Some network configurations do not provide this information, so it is possible that the logical path(s) for a given netloc_node_t is empty.

Currently only one logical path between any two netloc nodes is captured. Path information is only calculated between servers, not between switches in the network.

Chapter 6

Todo List

Global netloc_dc_append_edge_to_node JJH It would be easy to allow the node parameter to be NULL and infer to node from the edge.

JJH Add a check to make sure we only add edges to the source node.

Class netloc_edge_t JJH Is the note above still true?

Global netloc_map_find_neighbors Brice FIXME: get neighbor nodes at a given distance, within any or a single subnet

Brice FIXME: get neighbor nodes with enough cores, within any or a single subnet

Brice This interface is temporary, for debugging

Page Reader (Data Collection) API JJH Logical paths have not been well tested.

Chapter 7

Module Index

7.1 Modules

Here is a list of all modules:

Chapter 8

Data Structure Index

8.1 Data Structures

Here are the data structures with brief descriptions:

Chapter 9

Module Documentation

9.1 Netloc API

Data Structures

- struct netloc_network_t

  Netloc Network Type.

- struct netloc_edge_t

  Netloc Edge Type.

- struct netloc_node_t

  Netloc Node Type.

- struct netloc_topology_t

  Netloc Topology Context.

- struct netloc_dt_lookup_table_t

  Lookup Table Type.

- struct netloc_dt_lookup_table_iterator_t

  Lookup Table Iterator.

Typedefs

- typedef struct netloc_network_t netloc_network_t
- typedef struct netloc_node_t netloc_node_t

- typedef struct netloc_edge_t netloc_edge_t

Enumerations

- enum netloc_compare_type_t { NETLOC_CMP_SAME = 0, NETLOC_CMP_SIMILAR = -1, NETLOC_CMP_DIFF = -2 }
- enum netloc_network_type_t { NETLOC_NETWORK_TYPE_ETHERNET = 1, NETLOC_NETWORK_TYPE_INFINIBAND = 2, NETLOC_NETWORK_TYPE_INVALID = 3 }
- enum netloc_node_type_t { NETLOC_NODE_TYPE_SWITCH = 1, NETLOC_NODE_TYPE_HOST = 2, NETLOC_NODE_TYPE_INVALID = 3 }
- enum {

  NETLOC_SUCCESS = 0, NETLOC_ERROR = -1, NETLOC_ERROR_NOTDIR = -2, NETLOC_ERROR_NOENT = -3,

  NETLOC_ERROR_EMPTY = -4, NETLOC_ERROR_MULTIPLE = -5, NETLOC_ERROR_NOT_IMPL = -6, NETLOC_ERROR_EXISTS = -7,

  NETLOC_ERROR_NOT_FOUND = -8, NETLOC_ERROR_MAX = -9 }

Functions

- static netloc_network_type_t netloc_encode_network_type (const char ∗str_val)
- static const char ∗ netloc_decode_network_type (netloc_network_type_t net_type)
- static const char ∗ netloc_decode_network_type_readable (netloc_network_type_t net_type)
- static netloc_node_type_t netloc_encode_node_type (const char ∗str_val)
- static const char ∗ netloc_decode_node_type (netloc_node_type_t node_type)
- static char ∗ netloc_decode_node_type_readable (netloc_node_type_t node_type)
- netloc_network_t ∗ netloc_dt_network_t_construct (void)
- int netloc_dt_network_t_destruct (netloc_network_t ∗network)
- netloc_network_t ∗ netloc_dt_network_t_dup (netloc_network_t ∗network)
- int netloc_dt_network_t_copy (netloc_network_t ∗from, netloc_network_t ∗to)
- int netloc_dt_network_t_compare (netloc_network_t ∗a, netloc_network_t ∗b)
- int netloc_dt_edge_t_compare (netloc_edge_t ∗a, netloc_edge_t ∗b)
- int netloc_dt_node_t_compare (netloc_node_t ∗a, netloc_node_t ∗b)
- netloc_dt_lookup_table_iterator_t netloc_dt_lookup_table_iterator_t_construct (netloc_dt_lookup_table_t table)
- int netloc_dt_lookup_table_iterator_t_destruct (netloc_dt_lookup_table_iterator_t hti)
- int netloc_lookup_table_destroy (netloc_dt_lookup_table_t table)

- int netloc_lookup_table_size (netloc_dt_lookup_table_t table)
- void ∗ netloc_lookup_table_access (netloc_dt_lookup_table_t ht, const char ∗key)
- const char ∗ netloc_lookup_table_iterator_next_key (netloc_dt_lookup_table_iterator_t hti)
- void ∗ netloc_lookup_table_iterator_next_entry (netloc_dt_lookup_table_iterator_t hti)
- bool netloc_lookup_table_iterator_at_end (netloc_dt_lookup_table_iterator_t hti)
- void netloc_lookup_table_iterator_reset (netloc_dt_lookup_table_iterator_t hti)
- char ∗ netloc_pretty_print_network_t (netloc_network_t ∗network)
- char ∗ netloc_pretty_print_edge_t (netloc_edge_t ∗edge)
- char ∗ netloc_pretty_print_node_t (netloc_node_t ∗node)
- int netloc_find_network (const char ∗network_topo_uri, netloc_network_t ∗network)
- int netloc_foreach_network (const char ∗const ∗search_uris, int num_uris, int(∗func)(const netloc_network_t ∗network, void ∗funcdata), void ∗funcdata, int ∗num_networks, netloc_network_t ∗∗∗networks)
- int netloc_attach (netloc_topology_t ∗topology, netloc_network_t network)
- int netloc_detach (netloc_topology_t topology)
- int netloc_refresh (netloc_topology_t topology)
- netloc_network_t ∗ netloc_access_network_ref (netloc_topology_t topology)
- int netloc_get_all_nodes (netloc_topology_t topology, netloc_dt_lookup_table_t ∗nodes)
- int netloc_get_all_switch_nodes (netloc_topology_t topology, netloc_dt_lookup_table_t ∗nodes)
- int netloc_get_all_host_nodes (netloc_topology_t topology, netloc_dt_lookup_table_t ∗nodes)
- int netloc_get_all_edges (netloc_topology_t topology, netloc_node_t ∗node, int ∗num_edges, netloc_edge_t ∗∗∗edges)
- netloc_node_t ∗ netloc_get_node_by_physical_id (netloc_topology_t topology, const char ∗phy_id)
- int netloc_get_path (const netloc_topology_t topology, netloc_node_t ∗src_node, netloc_node_t ∗dst_node, int ∗num_edges, netloc_edge_t ∗∗∗path, bool is_logical)
- int netloc_topology_export_graphml (netloc_topology_t topology, const char ∗filename)
- int netloc_topology_export_gexf (netloc_topology_t topology, const char ∗filename)

9.1.1 Typedef Documentation

9.1.1.1 typedef struct netloc_edge_t netloc_edge_t

9.1.1.2 typedef struct netloc_network_t netloc_network_t

9.1.1.3 typedef struct netloc_node_t netloc_node_t

9.1.2 Enumeration Type Documentation

9.1.2.1 anonymous enum

Return codes

Enumerator:

NETLOC_SUCCESS Success

NETLOC_ERROR Error: General condition

NETLOC_ERROR_NOTDIR Error: URI is not a directory

NETLOC_ERROR_NOENT Error: URI is invalid, no such entry

NETLOC_ERROR_EMPTY Error: No networks found

NETLOC_ERROR_MULTIPLE Error: Multiple matching networks found

NETLOC_ERROR_NOT_IMPL Error: Interface not implemented

NETLOC_ERROR_EXISTS Error: If the entry already exists when trying to add to a lookup table

NETLOC_ERROR_NOT_FOUND Error: No path found

NETLOC_ERROR_MAX Error: Enum upper bound marker. No errors less than this number Will not be returned externally.

9.1.2.2 enum netloc_compare_type_t

Definitions for Comparators

See also:

These are the return values from the following functions: netloc_dt_network_t_compare, netloc_dt_edge_t_compare, netloc_dt_node_t_compare

Enumerator:

NETLOC_CMP_SAME Compared as the Same

NETLOC_CMP_SIMILAR Compared as Similar, but not the Same

NETLOC_CMP_DIFF Compared as Different

9.1.2.3 enum netloc_network_type_t

Enumerated type for the various types of supported networks

Enumerator:

NETLOC_NETWORK_TYPE_ETHERNET	Ethernet network
NETLOC_NETWORK_TYPE_INFINIBAND	InfiniBand network
NETLOC_NETWORK_TYPE_INVALID	Invalid network

9.1.2.4 enum netloc_node_type_t

Enumerated type for the various types of nodes

Enumerator:

NETLOC_NODE_TYPE_SWITCH	Switch node
NETLOC_NODE_TYPE_HOST	Host (a.k.a., network addressable endpoint - e.g., MAC Address) node
NETLOC_NODE_TYPE_INVALID	Invalid node

9.1.3 Function Documentation

9.1.3.1 netloc_network_t∗ netloc_access_network_ref (netloc_topology_t *topology*)

Access a reference to the netloc_network_t associated with the netloc_topology_t

The user should -not- call netloc_dt_network_t_destruct on the reference returned.

Parameters:

topology	A valid pointer to a topology handle

Returns:

A reference to the netloc_network_t assocaited with the topology
NULL on error.

9.1.3.2 int netloc_attach (netloc_topology_t ∗ *topology*, netloc_network_t *network*)

Attach to the specified network, and allocate a topology handle.

User is responsible for calling netloc_detach on the topology handle. The network parameter information is deep copied into the topology handle, so the user may destruct the network handle after calling this function and/or reuse the network handle.

Parameters:

topology A pointer to a netloc_topology_t handle.

network The netloc_network_t handle from a prior call to either:

- netloc_find_network()
- netloc_foreach_network()

Returns:

NETLOC_SUCCESS on success
NETLOC_ERROR upon an error.

9.1.3.3 static const char∗ netloc_decode_network_type (netloc_network_type_t *net_type*) `[inline, static]`

Decode the network type

Parameters:

net_type A valid member of the netloc_network_type_t type

Returns:

NULL if the type is invalid
A string for that netloc_network_type_t type

9.1.3.4 static const char∗ netloc_decode_network_type_readable (netloc_network_type_t *net_type*) `[inline, static]`

Decode the network type into a human readable string

Parameters:

net_type A valid member of the netloc_network_type_t type

Returns:

A string for that netloc_network_type_t type

9.1.3.5 static const char∗ netloc_decode_node_type (netloc_node_type_t *node_type*) `[inline, static]`

Decode the node type

Parameters:

node_type A valid member of the netloc_node_type_t type

Returns:

NULL if the type is invalid
A string for that netloc_node_type_t type

9.1.3.6 static char∗ netloc_decode_node_type_readable (netloc_node_type_t *node_type*) `[inline, static]`

Decode the node type into a human readable string

Parameters:

node_type A valid member of the netloc_node_type_t type

Returns:

NULL if the type is invalid
A string for that netloc_node_type_t type

9.1.3.7 int netloc_detach (netloc_topology_t *topology*)

Detach from a topology handle

Parameters:

topology A valid pointer to a netloc_topology_t handle created from a prior call to netloc_attach.

Returns:

NETLOC_SUCCESS on success
NETLOC_ERROR upon an error.

9.1.3.8 int netloc_dt_edge_t_compare (netloc_edge_t $*$ *a*, netloc_edge_t $*$ *b*)

Compare function for netloc_edge_t

Parameters:

a	A pointer to one edge object for comparison
b	A pointer to the other edge object for comparison

Returns:

NETLOC_CMP_SAME if the same
NETLOC_CMP_DIFF if different

9.1.3.9 netloc_dt_lookup_table_iterator_t netloc_dt_lookup_table_iterator_t_construct (netloc_dt_lookup_table_t *table*)

Constructor for a lookup table iterator

User is responsible for calling the netloc_dt_lookup_table_iterator_t_destruct on the handle.

Parameters:

table	The table to reference in this iterator

Returns:

A newly allocated pointer to the lookup table iterator.

9.1.3.10 int netloc_dt_lookup_table_iterator_t_destruct (netloc_dt_lookup_table_iterator_t *hti*)

Destructor for a lookup table iterator

Parameters:

hti	A valid lookup table iterator handle

Returns:

NETLOC_SUCCESS on success
NETLOC_ERROR on error

9.1.3.11 int netloc_dt_network_t_compare (netloc_network_t ∗ *a*, netloc_network_t ∗ *b*)

Compare function for netloc_network_t

Parameters:

a A pointer to one network object for comparison

b A pointer to the other network object for comparison

Returns:

NETLOC_CMP_SAME if the same
NETLOC_CMP_SIMILAR if only the metadata (e.g., version) is different
NETLOC_CMP_DIFF if different

9.1.3.12 netloc_network_t∗ netloc_dt_network_t_construct (void)

Constructor for netloc_network_t

User is responsible for calling the destructor on the handle.

Returns:

A newly allocated pointer to the network information.

9.1.3.13 int netloc_dt_network_t_copy (netloc_network_t ∗ *from*, netloc_network_t ∗ *to*)

Copy Function for netloc_network_t

Does not allocate memory for ’to’. Does a shallow copy of the pointers to data.

Parameters:

from A pointer to the network to duplicate

to A pointer to the network to duplicate into

Returns:

NETLOC_SUCCESS on success
NETLOC_ERROR on error

9.1.3.14 int netloc_dt_network_t_destruct (netloc_network_t * *network*)

Destructor for netloc_network_t

Parameters:

network A valid network handle

Returns:

NETLOC_SUCCESS on success
NETLOC_ERROR on error

9.1.3.15 netloc_network_t* netloc_dt_network_t_dup (netloc_network_t * *network*)

Copy Constructor for netloc_network_t

Allocates memory. User is responsible for calling netloc_dt_network_t_destruct on the returned pointer. Does a shallow copy of the pointers to data.

Parameters:

network A pointer to the network to duplicate

Returns:

A newly allocated copy of the network.

9.1.3.16 int netloc_dt_node_t_compare (netloc_node_t * *a*, netloc_node_t * *b*)

Compare function for netloc_node_t

Parameters:

a A pointer to one network object for comparison

b A pointer to the other network object for comparison

Returns:

NETLOC_CMP_SAME if the same
NETLOC_CMP_DIFF if different

9.1.3.17 static netloc_network_type_t netloc_encode_network_type (const char $*$ *str_val*) `[inline, static]`

Encode the network type

Note:

Only used by netloc readers to encode the network type

Parameters:

str_val String value to parse

Returns:

A valid member of the netloc_network_type_t type

9.1.3.18 static netloc_node_type_t netloc_encode_node_type (const char $*$ *str_val*) `[inline, static]`

Encode the node type

Note:

Only used by netloc readers to encode the network type

Parameters:

str_val String value to parse

Returns:

A valid member of the netloc_node_type_t type

9.1.3.19 int netloc_find_network (const char $*$ *network_topo_uri*, netloc_network_t $*$ *network*)

Find a specific network at the URI specified.

Parameters:

network_topo_uri URI to search for the specified network.

network Netloc network handle (IN/OUT) A network handle with the data structure fields set to specify the search. For example, the user can set 'IB' and nothing else, if they do not know the subnet or any of the other necessary information. If the method returns success then the network handle will be filled out with the rest of the information found. If the method returns some error then the network handle is not modified.

Returns:

NETLOC_SUCCESS if exactly one network matches the specification, and updates the network handle.
NETLOC_ERROR_MULTIPLE if more than one network matches the spec.
NETLOC_ERROR_EMPTY if no networks match the specification.
NETLOC_ERROR_NOENT if the directory does not exist.
NETLOC_ERROR_NOTDIR if the data_dir is not a directory.
NETLOC_ERROR if something else is wrong.

9.1.3.20 int netloc_foreach_network (const char ∗const ∗ *search_uris*, int *num_uris*, int(∗)(const netloc_network_t ∗network, void ∗funcdata) *func*, void ∗ *funcdata*, int ∗ *num_networks*, netloc_network_t ∗∗∗ *networks*)

Find all available networks in the specified URIs

User is responsible for calling the destructor for each element of the networks array paramater, then free() on the entire array.

Parameters:

search_uris Array of URIs. `file://` syntax is the only supported mechanism at the moment. Array is searched for .dat files. All uris will be searched. If NULL is supplied then the default search path will be used (currently the CWD).

num_uris Size of the search_uris array.

(∗func) A callback function triggered for each network found the user is provided an opportunity to decide if it should be included in the "networks" array or not. "net" is a handle to the network information (includes uri where it was found). If the callback returns non-zero then the entry is added to the networks array. If the callback returns 0 then the entry is not added to the networks array. If NULL is supplied as an argument for this function pointer then all networks are included in the array.

funcdata User specified data pointer to be passed to the callback function.

num_networks Size of the networks array.

networks An array of networks discovered.

Returns:

NETLOC_SUCCESS on success
NETLOC_ERROR otherwise

9.1.3.21 int netloc_get_all_edges (netloc_topology_t *topology*, netloc_node_t ∗ *node*, int ∗ *num_edges*, netloc_edge_t ∗∗∗ *edges*)

Get all of the edges from the specified node in the network topology. There should be one edge for every active port on this node.

The user should not free the array, neither its elements.

Parameters:

- ***topology*** A valid pointer to a topology handle
- ***node*** A valid pointer to a netloc_node_t from which to get the edges.
- ***num_edges*** The number of edges in the edges array.
- ***edges*** An array of netloc_edge_t objects

Returns:

NETLOC_SUCCESS on success
NETLOC_ERROR upon an error.

9.1.3.22 int netloc_get_all_host_nodes (netloc_topology_t *topology*, netloc_dt_lookup_table_t ∗ *nodes*)

Get only host nodes in the network topology

The user is responsible for calling the lookup table destructor on the nodes table (netloc_lookup_table_destroy). The user should -not- call the netloc_node_t's destructor on the elements in the lookup table. That interface (netloc_dt_node_t_destruct) is not publicly exposed.

Parameters:

- ***topology*** A valid pointer to a topology handle
- ***nodes*** A lookup table of the nodes requested Keys in the table are the netloc_node_t::physical_id's of the netloc_node_t objects The values are pointers to netloc_node_t objects

Returns:

NETLOC_SUCCESS on success
NETLOC_ERROR upon an error.

9.1.3.23 int netloc_get_all_nodes (netloc_topology_t *topology*, netloc_dt_lookup_table_t ∗ *nodes*)

Get all nodes in the network topology

The user is responsible for calling the lookup table destructor on the nodes table (netloc_lookup_table_destroy). The user should -not- call the netloc_node_t's destructor on the elements in the lookup table. That interface (netloc_dt_node_t_destruct) is not publicly exposed.

Parameters:

topology A valid pointer to a topology handle

nodes A lookup table of the nodes requested Keys in the table are the netloc_node_t::physical_id's of the netloc_node_t objects The values are pointers to netloc_node_t objects

Returns:

NETLOC_SUCCESS on success
NETLOC_ERROR upon an error.

9.1.3.24 int netloc_get_all_switch_nodes (netloc_topology_t *topology*, netloc_dt_lookup_table_t ∗ *nodes*)

Get only switch nodes in the network topology

The user is responsible for calling the lookup table destructor on the nodes table (netloc_lookup_table_destroy). The user should -not- call the netloc_node_t's destructor on the elements in the lookup table. That interface (netloc_dt_node_t_destruct) is not publicly exposed.

Parameters:

topology A valid pointer to a topology handle

nodes A lookup table of the nodes requested Keys in the table are the netloc_node_t::physical_id's of the netloc_node_t objects The values are pointers to netloc_node_t objects

Returns:

NETLOC_SUCCESS on success
NETLOC_ERROR upon an error.

9.1.3.25 netloc_node_t∗ netloc_get_node_by_physical_id (netloc_topology_t *topology*, const char ∗ *phy_id*)

Access the netloc_node_t pointer given a physical identifier (e.g., MAC address, GUID)

The user should -not- call the destructor on the returned value.

Parameters:

topology A valid pointer to a topology handle

phy_id The physical identifier to search for (e.g., MAC address, GUID)

Returns:

A pointer to the netloc_node_t with the specified physical identifier
NULL if the phy_id is not found.

9.1.3.26 int netloc_get_path (const netloc_topology_t *topology*, netloc_node_t ∗ *src_node*, netloc_node_t ∗ *dst_node*, int ∗ *num_edges*, netloc_edge_t ∗∗∗ *path*, bool *is_logical*)

Get the "path" from the source to the destination as an ordered array of netloc_edge_t objects

The user is responsible for calling free() on the allocated array, but -not- the elements in the array.

Warning:

A large API change is in the works for v1.0 that will change how we represent path data.

Parameters:

topology A valid pointer to a topology handle

src_node A valid pointer to the source node

dst_node A valid pointer to the destination node

num_edges The number of edges in the path array.

path An ordered array of netloc_edge_t objects from the source to the destination

is_logical If the path should represent the logical or the physical path information.

Returns:

NETLOC_SUCCESS on success
NETLOC_ERROR upon an error.

9.1.3.27 void∗ netloc_lookup_table_access (netloc_dt_lookup_table_t *ht*, const char ∗ *key*)

Access an entry in the lookup table

Parameters:

ht A valid pointer to a lookup table

key The key used to find the data

Returns:

NULL if nothing found
The pointer associated with this key

9.1.3.28 int netloc_lookup_table_destroy (netloc_dt_lookup_table_t *table*)

Destroy a lookup table.

Note:

The user is responsible for calling this function if they are ever returned a netloc_dt_lookup_table_t from a function such as netloc_get_all_nodes.

Parameters:

table The lookup table to destroy

Returns:

NETLOC_SUCCESS on success
NETLOC_ERROR on error

9.1.3.29 bool netloc_lookup_table_iterator_at_end (netloc_dt_lookup_table_iterator_t *hti*)

Check if we are at the end of the iterator

Parameters:

hti A valid pointer to a lookup table iterator

Returns:

true if at the end of the data, false otherwise

9.1.3.30 void∗ netloc_lookup_table_iterator_next_entry (netloc_dt_lookup_table_iterator_t *hti*)

Get the next entry and advance the iterator

Similar to netloc_lookup_table_iterator_next_key except the caller is given the next value directly. So they do not need to call the netloc_lookup_table_access function to access the value.

Parameters:

hti A valid pointer to a lookup table iterator

Returns:

NULL if error or at end
The pointer associated with this key

9.1.3.31 const char∗ netloc_lookup_table_iterator_next_key (netloc_dt_lookup_table_iterator_t *hti*)

Get the next key and advance the iterator

The user should -not- call free() on the string returned.

Parameters:

hti A valid pointer to a lookup table iterator

Returns:

NULL if error or at end
A newly allocated string copy of the key.

9.1.3.32 void netloc_lookup_table_iterator_reset (netloc_dt_lookup_table_iterator_t *hti*)

Reset the iterator back to the start

Parameters:

hti A valid pointer to a lookup table iterator

9.1.3.33 int netloc_lookup_table_size (netloc_dt_lookup_table_t *table*)

Access the -used- size of the lookup table

Parameters:

table A valid pointer to a lookup table

Returns:

The used size of the lookup table

9.1.3.34 char∗ netloc_pretty_print_edge_t (netloc_edge_t ∗ *edge*)

Pretty print the edge (Debugging Support)

The user is responsible for calling free() on the string returned.

Parameters:

edge A valid pointer to an edge

Returns:

A newly allocated string representation of the edge.

9.1.3.35 char∗ netloc_pretty_print_network_t (netloc_network_t ∗ *network*)

Pretty print the network (Debugging Support)

The user is responsible for calling free() on the string returned.

Parameters:

network A valid pointer to a network

Returns:

A newly allocated string representation of the network.

9.1.3.36 char∗ netloc_pretty_print_node_t (netloc_node_t ∗ *node*)

Pretty print the node (Debugging Support)

The user is responsible for calling free() on the string returned.

Parameters:

node A valid pointer to a node

Returns:

A newly allocated string representation of the node.

9.1.3.37 int netloc_refresh (netloc_topology_t *topology*)

Refresh the data associated with the topology.

Warning:

This interface is not currently implemented.

Parameters:

topology A valid pointer to a netloc_topology_t handle created from a prior call to netloc_attach.

Returns:

NETLOC_SUCCESS on success
NETLOC_ERROR upon an error.

9.1.3.38 int netloc_topology_export_gexf (netloc_topology_t *topology*, const char ∗ *filename*)

Exports the network topology to a GEXF formatted file.

Parameters:

topology A valid pointer to a topology handle

filename The filename to write the data to

Returns:

NETLOC_SUCCESS on success
NETLOC_ERROR upon an error.

9.1.3.39 int netloc_topology_export_graphml (netloc_topology_t *topology*, const char ∗ *filename*)

Exports the network topology to a GraphML formatted file.

Parameters:

topology	A valid pointer to a topology handle
filename	The filename to write the data to

Returns:

NETLOC_SUCCESS on success
NETLOC_ERROR upon an error.

9.2 Data Collection API

Data Structures

- struct netloc_data_collection_handle_t

  Data Collection Handle.

Typedefs

- typedef struct netloc_data_collection_handle_t netloc_data_collection_handle_t

Functions

- netloc_data_collection_handle_t ∗ netloc_dt_data_collection_handle_t_construct ()
- int netloc_dt_data_collection_handle_t_destruct (netloc_data_collection_handle_t ∗handle)
- netloc_data_collection_handle_t ∗ netloc_dc_create (netloc_network_t ∗network, char ∗dir)
- int netloc_dc_close (netloc_data_collection_handle_t ∗handle)
- netloc_network_t ∗ netloc_dc_handle_get_network (netloc_data_collection_handle_t ∗handle)
- char ∗ netloc_dc_handle_get_unique_id_str (netloc_data_collection_handle_t ∗handle)
- char ∗ netloc_dc_handle_get_unique_id_str_filename (char ∗filename)
- int netloc_dc_append_node (netloc_data_collection_handle_t ∗handle, netloc_node_t ∗node)
- int netloc_dc_append_edge_to_node (netloc_data_collection_handle_t ∗handle, netloc_node_t ∗node, netloc_edge_t ∗edge)
- int netloc_dc_append_edge_to_node_by_id (netloc_data_collection_handle_t ∗handle, char ∗phy_id, netloc_edge_t ∗edge)
- netloc_node_t ∗ netloc_dc_get_node_by_physical_id (netloc_data_collection_handle_t ∗handle, char ∗phy_id)
- int netloc_dc_append_path (netloc_data_collection_handle_t ∗handle, const char ∗src_node_id, const char ∗dest_node_id, int num_edges, netloc_edge_t ∗∗edges, bool is_logical)
- int netloc_dc_compute_path_between_nodes (netloc_data_collection_handle_t ∗handle, netloc_node_t ∗src_node, netloc_node_t ∗dest_node, int ∗num_edges, netloc_edge_t ∗∗∗edges, bool is_logical)
- void netloc_dc_pretty_print (netloc_data_collection_handle_t ∗handle)

9.2.1 Detailed Description

This interface extends the "north bound" (user facing) interface with functionlaity to support backed (or "south bound") readers.

Readers should use this API to store netloc structures. The intention of this interface is to abstract away the data storage mechanism from the readers.

9.2.2 Typedef Documentation

9.2.2.1 typedef struct netloc_data_collection_handle_t netloc_data_collection_handle_t

9.2.3 Function Documentation

9.2.3.1 int netloc_dc_append_edge_to_node (netloc_data_collection_handle_t ∗ *handle*, netloc_node_t ∗ *node*, netloc_edge_t ∗ *edge*)

Append netloc_edge_t information to the netloc_node_t structure

This function makes a copy of the edge information before storing it on the node. So the user may reuse the edge, and is responsible for calling the edge destructor when finished with it (netloc_dt_edge_t_destruct).

Todo

JJH It would be easy to allow the node parameter to be NULL and infer to node from the edge.
JJH Add a check to make sure we only add edges to the source node.

Parameters:

handle	A valid pointer to a data collection handle
node	A valid pointer to a netloc_node_t to append the edge to
edge	A valid pointer to the edge information to attach

Returns:

NETLOC_SUCCESS upon success
NETLOC_ERROR otherwise

9.2.3.2 int netloc_dc_append_edge_to_node_by_id (netloc_data_collection_handle_t ∗ *handle*, char ∗ *phy_id*, netloc_edge_t ∗ *edge*)

Append netloc_edge_t information to the internal netloc_node_t structure by using the physical ID of the node.

Logically, this is similar to doing the following.

```
netloc_dc_append_edge_to_node(handle, netloc_dc_get_node_by_physical_id(handle
   , phy_id), edge);
```

This function makes a copy of the edge information before storing it on the node. So the user may reuse the edge, and is responsible for calling the edge destructor when finished with it (netloc_dt_edge_t_destruct).

Parameters:

handle A valid pointer to a data collection handle

phy_id The physical_id to search for

edge A valid pointer to the edge information to attach

Returns:

NETLOC_SUCCESS upon success
NETLOC_ERROR otherwise

9.2.3.3 int netloc_dc_append_node (netloc_data_collection_handle_t ∗ *handle*, netloc_node_t ∗ *node*)

Append netloc_node_t information to the data collection

Parameters:

handle A valid pointer to a data collection handle

node A pointer to the netloc_node_t to append

Returns:

NETLOC_SUCCESS upon success
NETLOC_ERROR otherwise

9.2.3.4 int netloc_dc_append_path (netloc_data_collection_handle_t ∗ *handle*, const char ∗ *src_node_id*, const char ∗ *dest_node_id*, int *num_edges*, netloc_edge_t ∗∗ *edges*, bool *is_logical*)

Append a path between two netloc_node_t objects Each edge in this list will be appened to the data collection, if it is not already there.

Parameters:

handle A valid pointer to a data collection handle

src_node_id Physical node id of the source

dest_node_id Physical node id of the destination

num_edges Number of edges in the edges array

edges Ordered array of edges from the source to the destination

is_logical If the path is a logical or physical path

Returns:

NETLOC_SUCCESS upon success
NETLOC_ERROR otherwise

9.2.3.5 int netloc_dc_close (netloc_data_collection_handle_t ∗ *handle*)

Close a data collection handle This may write out data if the handle was created in netloc_dc_create.

The user is responsible for calling netloc_dt_data_collection_handle_t_destruct on the handle when finished with it. The close function does not destruct the handle.

Parameters:

handle A valid pointer to a data collection handle

Returns:

NETLOC_SUCCESS upon success
NETLOC_ERROR otherwise

9.2.3.6 int netloc_dc_compute_path_between_nodes (netloc_data_collection_handle_t ∗ *handle*, netloc_node_t ∗ *src_node*, netloc_node_t ∗ *dest_node*, int ∗ *num_edges*, netloc_edge_t ∗∗∗ *edges*, bool *is_logical*)

Compute the path between two nodes

Warning:

Logical paths is known not to be fully implemented/tested.

Parameters:

handle A valid point to a data collection handle

src_node A reference to the source node to compute the path from

dest_node A reference to the destination node to compute the path to

num_edges The number of edges in the edges array

edges An ordered list of edges from the source node to the destination node.

is_logical If the path is a logical or physical path

Returns:

NETLOC_SUCCESS upon success
NETLOC_ERROR_NOT_IMPL if is_logical is true
NETLOC_ERROR otherwise

9.2.3.7 netloc_data_collection_handle_t∗ netloc_dc_create (netloc_network_t ∗ *network*, char ∗ *dir*)

Create a new data collection for this network.

The user is responsible for calling the netloc_dt_data_collection_handle_t_destruct function on the pointer returned once finished with the handle.

This function duplicates the netloc_network_t pointer passed to it, so the user is free to call the the netloc_dt_network_t_destruct function on the pointer when finished with it.

Parameters:

network Network information (must be complete, from a prior call to netloc_find_network)

dir Directory to store the .ndat files (Allowed to be NULL if current working directory)

Returns:

NULL on error
A valid data collection handle on success

9.2.3.8 netloc_node_t∗ netloc_dc_get_node_by_physical_id (netloc_data_collection_handle_t ∗ *handle*, char ∗ *phy_id*)

Access a stored node by the physcial identifier (e.g., MAC address, GUID)

The user should -not- call the destructor on the returned value.

Parameters:

handle	A valid pointer to a data collection handle
phy_id	The physical_id to search for

Returns:

A pointer to the netloc_node_t with the specified physical_id
NULL if the phy_id is not found.

9.2.3.9 netloc_network_t∗ netloc_dc_handle_get_network (netloc_data_collection_handle_t ∗ *handle*)

Get the network information from the handle.

Parameters:

handle	A valid pointer to a data collection handle

Returns:

NULL if no network information found
Pointer to a netloc_network_t (caller is responsibe for deallocating this object)

9.2.3.10 char∗ netloc_dc_handle_get_unique_id_str (netloc_data_collection_handle_t ∗ *handle*)

Get the unique_id_str for the specified handle

Parameters:

handle	A valid pointer to a data collection handle

Returns:

NULL if handle is invalid, or has no unique_id_str
Unique ID string for this handle (caller is responsible for deallocating the string)

9.2.3.11 char∗ netloc_dc_handle_get_unique_id_str_filename (char ∗ *filename*)

Get the unique_id_str for the specified filename (so we might open it)

Parameters:

filename Filename with network information

Returns:

NULL if handle is invalid, or has no unique_id_str
Unique ID string for this handle (caller is responsible for deallocating the string)

9.2.3.12 void netloc_dc_pretty_print (netloc_data_collection_handle_t ∗ *handle*)

Pretty print the data collection to stdout (Debugging Support)

Parameters:

handle A valid pointer to a data collection handle

9.2.3.13 netloc_data_collection_handle_t∗ netloc_dt_data_collection_handle_t_construct ()

Constructor for netloc_data_collection_handle_t

User is responsible for calling the destructor on the handle.

Returns:

A newly constructed collection handle

9.2.3.14 int netloc_dt_data_collection_handle_t_destruct (netloc_data_collection_handle_t ∗ *handle*)

Destructor for netloc_data_collection_handle_t

Parameters:

handle A pointer to a netloc_data_collection_handle_t previously constructed by netloc_dt_data_collection_handle_t_construct.

9.3 Netloc Map API - Main objects.

Typedefs

- typedef void ∗ netloc_map_t
- typedef void ∗ netloc_map_server_t
- typedef void ∗ netloc_map_port_t

9.3.1 Typedef Documentation

9.3.1.1 typedef void∗ netloc_map_port_t

A netloc map port handle. Servers are interconnected by their ports.

9.3.1.2 typedef void∗ netloc_map_server_t

A netloc map server handle.

9.3.1.3 typedef void∗ netloc_map_t

A netloc map handle. A map contains servers interconnected by networks.

9.4 Netloc Map API - Building maps.

Enumerations

- enum netloc_map_build_flags_e { NETLOC_MAP_BUILD_FLAG_COMPRESS_HWLOC }

Functions

- int netloc_map_create (netloc_map_t ∗map)
- int netloc_map_load_hwloc_data (netloc_map_t map, const char ∗data_dir)
- int netloc_map_load_netloc_data (netloc_map_t map, const char ∗data_dir)
- int netloc_map_build (netloc_map_t map, unsigned long flags)
- int netloc_map_destroy (netloc_map_t map)

9.4.1 Enumeration Type Documentation

9.4.1.1 enum netloc_map_build_flags_e

Flags to be passed as a OR'ed set to the netloc_map_build() function

Enumerator:

NETLOC_MAP_BUILD_FLAG_COMPRESS_HWLOC Enable hwloc topology compression if supported.

9.4.2 Function Documentation

9.4.2.1 int netloc_map_build (netloc_map_t *map*, unsigned long *flags*)

Build a map that was previously created and where hwloc and netloc data were loaded.

Requires the netloc_map_load_hwloc_data() and netloc_map_load_netloc_data() functions have been called on the map object.

Parameters:

map A netloc map.

flags Any OR'ed set of netloc_map_build_flags_e.

Returns:

0 on success
-1 on error

9.4.2.2 int netloc_map_create (netloc_map_t ∗ *map*)

Create a map

Parameters:

map The map object to create.

Once created, the map object needs to be attached to hwloc and netloc data directories.

Returns:

0 on success
-1 on error

9.4.2.3 int netloc_map_destroy (netloc_map_t *map*)

Destroy a map.

Parameters:

map A netloc map.

This function must be called even if netloc_map_build() failed.

Returns:

0 on success

9.4.2.4 int netloc_map_load_hwloc_data (netloc_map_t *map*, const char ∗ *data_dir*)

Loading the hwloc data from a directory into a map.

Parameters:

map The map object to attach the data to.

data_dir the data directory to read the hwloc information from.

Actual failures to read the data will cause an error during netloc_map_build().

Returns:

0 on success

9.4.2.5 int netloc_map_load_netloc_data (netloc_map_t *map*, const char ∗ *data_dir*)

Loading the netloc data from a directory into a map.

Parameters:

map The map object to attach the data to.

data_dir the data directory to read the netloc information from.

Actual failures to read the data will cause an error during netloc_map_build().

Returns:

0 on success

9.5 Netloc Map API - Manipulating servers and ports

Functions

- int netloc_map_hwloc2port (netloc_map_t map, hwloc_topology_t htopo, hwloc_obj_t hobj, netloc_map_port_t ∗ports, unsigned ∗nrp)
- int netloc_map_netloc2port (netloc_map_t map, netloc_topology_t ntopo, netloc_node_t ∗nnode, netloc_edge_t ∗nedge, netloc_map_port_t ∗port)
- int netloc_map_port2netloc (netloc_map_port_t port, netloc_topology_t ∗ntopo, netloc_node_t ∗∗nnode, netloc_edge_t ∗∗nedge)
- int netloc_map_port2hwloc (netloc_map_port_t port, hwloc_topology_t ∗htopop, hwloc_obj_t ∗hobjp)
- int netloc_map_server2hwloc (netloc_map_server_t server, hwloc_topology_t ∗topology)
- int netloc_map_hwloc2server (netloc_map_t map, hwloc_topology_t topology, netloc_map_server_t ∗server)
- int netloc_map_put_hwloc (netloc_map_t map, hwloc_topology_t topology)
- int netloc_map_get_subnets (netloc_map_t map, unsigned ∗nr, netloc_topology_t ∗∗topos)
- int netloc_map_get_nbservers (netloc_map_t map)
- int netloc_map_get_servers (netloc_map_t map, unsigned first, unsigned nr, netloc_map_server_t servers[ ])
- int netloc_map_get_server_ports (netloc_map_server_t server, unsigned ∗nr, netloc_map_port_t ∗∗ports)
- int netloc_map_port2server (netloc_map_port_t port, netloc_map_server_t ∗server)
- int netloc_map_server2map (netloc_map_server_t server, netloc_map_t ∗map)
- int netloc_map_server2name (netloc_map_server_t server, const char ∗∗name)
- int netloc_map_name2server (netloc_map_t map, const char ∗name, netloc_map_server_t ∗server)

9.5.1 Function Documentation

9.5.1.1 int netloc_map_get_nbservers (netloc_map_t *map*)

Get the number of servers.

Parameters:

map A netloc map.

Returns:

The number of servers in the map.

9.5.1.2 int netloc_map_get_server_ports (netloc_map_server_t *server*, unsigned ∗ *nr*, netloc_map_port_t ∗∗ *ports*)

Return the map ports from the server.

Parameters:

server A map server.

nr The number of ports returned in `ports` on success.

ports The array of map server ports returned on success.

Note:

The caller should not free the array.

Returns:

0 on success

9.5.1.3 int netloc_map_get_servers (netloc_map_t *map*, unsigned *first*, unsigned *nr*, netloc_map_server_t *servers*[ ])

Fill the input array with a range of servers.

Parameters:

map A netloc map.

first The index of the first server to return.

nr The number of servers to return.

servers A preallocated array that is filled with servers on success.

Note:

Servers must be allocated (and freed) by the caller.
This function is not performance-optimized, it may be slow when first is high.

Returns:

0 on success
-1 on error, for instance if `first` and/or `nr` is too high.

9.5.1.4 int netloc_map_get_subnets (netloc_map_t *map*, unsigned $*$ *nr*, netloc_topology_t $**$ *topos*)

Get an array of subnets existing in a netloc map.

Parameters:

map A netloc map.

nr The number of network topologies returned in `*topos` on success.

topos The array of network topologies returned on success.

Note:

the caller should free the array, not its contents.

Returns:

0 on success
-1 on error

9.5.1.5 int netloc_map_hwloc2port (netloc_map_t *map*, hwloc_topology_t *htopo*, hwloc_obj_t *hobj*, netloc_map_port_t $*$ *ports*, unsigned $*$ *nrp*)

Returns map ports that are close to a hwloc topology and object.

Parameters:

map A netloc map.

htopo A hwloc topology that was previously returned by netloc.

hobj A optional hwloc object inside the hwloc topology. If `hobj` is `NULL`, all ports of that map server match. If `hobj` is a I/O device, the matching ports that are returned are connected to that device. Otherwise, the matching ports are connected to a I/O device close to `hobj`.

ports The array where the corresponding map ports will be stored. The caller must be preallocate a number of slots that is given in `*nr` on input.

nr A pointer to the number of ports. On input, specifies how many ports can be stored in the `ports` array. On output, specifies how many were actually stored.

Returns:

0 on success
-1 on error

9.5.1.6 int netloc_map_hwloc2server (netloc_map_t *map*, hwloc_topology_t *topology*, netloc_map_server_t $*$ *server*)

Return a map server from a hwloc topology.

Parameters:

map A netloc map.

topology A hwloc topology. It must have been previously obtained from this netloc map. It cannot be another topology loaded by another piece of software.

server The corresponding map server returned on success.

Note:

Server should not be freed by the caller.
Another way to find a server is to extract the hostname from a hwloc topology with hwloc_obj_get_info_by_name(hwloc_get_root_obj(topology), "HostName") and pass it to netloc_map_name2server().

Returns:

0 on success
-1 on error

9.5.1.7 int netloc_map_name2server (netloc_map_t *map*, const char $*$ *name*, netloc_map_server_t $*$ *server*)

Return the map server object from a name.

Parameters:

map A netloc map.

name The name of the server.

server The corresponding server object returned on success.

Returns:

0 on success
-1 on error

9.5.1.8 int netloc_map_netloc2port (netloc_map_t *map*, netloc_topology_t *ntopo*, netloc_node_t $*$ *nnode*, netloc_edge_t $*$ *nedge*, netloc_map_port_t $*$ *port*)

Return the map port corresponding to a network edge and/or node.

Parameters:

map A netloc map.

ntopo A network topology.

nnode A network node where to look for ports. Cannot be `NULL` if nedge is `NULL`.

nedge A network edge where to look for ports. Cannot be `NULL` if nnode is `NULL`.

port The corresponding port returned on success.

Note:

On input, one (and only one) of nedge and nnode may be NULL. If both are non-NULL, they should match.

Returns:

0 on success
-1 on error

9.5.1.9 int netloc_map_port2hwloc (netloc_map_port_t *port*, hwloc_topology_t ∗ *htopop*, hwloc_obj_t ∗ *hobjp*)

Return the hwloc topology and object from a port.

Parameters:

port A port to look for hwloc topology and object.

htopop The corresponding hwloc topology is returned on success.

hobjp The corresponding hwloc object is returned on success, if not `NULL`.

A reference will be taken on the returned hwloc topology, it should be released later with netloc_map_put_hwloc().

Returns:

0 on success
-1 on error

9.5.1.10 int netloc_map_port2netloc (netloc_map_port_t *port*, netloc_topology_t ∗ *ntopo*, netloc_node_t ∗∗ *nnode*, netloc_edge_t ∗∗ *nedge*)

Return the network node and edge from a port.

Parameters:

port A port to look for edge and node.

ntopo A netloc topology returned on success.

nnode The corresponding network node returned on success, if not `NULL`.

nedge The corresponding network edge returned on success, if not `NULL`.

Returns:

0 on success
-1 on error

9.5.1.11 int netloc_map_port2server (netloc_map_port_t *port*, netloc_map_server_t ∗ *server*)

Return the map server from a port.

Parameters:

port A map server port.

The corresponding map server returned on success.

Returns:

0 on success

9.5.1.12 int netloc_map_put_hwloc (netloc_map_t *map*, hwloc_topology_t *topology*)

Release a hwloc topology pointer that we got above.

Parameters:

map A netloc map.

topology A hwloc topology previously obtained with netloc_map_port2hwloc() or netloc_map_server2hwloc().

Returns:

0 on success
-1 on error

9.5.1.13 int netloc_map_server2hwloc (netloc_map_server_t *server*, hwloc_topology_t ∗ *topology*)

Return the hwloc topology from a map server.

Parameters:

port A map server.

htopop The corresponding hwloc topology is returned on success.

A reference will be taken on the returned hwloc topology, it should be released later with netloc_map_put_hwloc().

Returns:

0 on success
-1 on error

9.5.1.14 int netloc_map_server2map (netloc_map_server_t *server*, netloc_map_t ∗ *map*)

Return the map of a server

Parameters:

server A server object

map The corresponding map which the server is part of.

Returns:

0 on success

9.5.1.15 int netloc_map_server2name (netloc_map_server_t *server*, const char ∗∗ *name*)

Return the name of a server.

Parameters:

server A server object.

name The name associated with that server.

Returns:

0 on success
-1 on error

9.6 Netloc Map API - Finding paths within a map

Data Structures

- struct netloc_map_edge_s

Typedefs

- typedef void ∗ netloc_map_paths_t

Enumerations

- enum netloc_map_paths_flag_e { NETLOC_MAP_PATHS_FLAG_IO = (1UL << 0), NETLOC_MAP_PATHS_FLAG_VERTICAL = (1UL << 1) }

Functions

- int netloc_map_paths_build (netloc_map_t map, hwloc_topology_t srctopo, hwloc_obj_t srcobj, hwloc_topology_t dsttopo, hwloc_obj_t dstobj, unsigned long flags, netloc_map_paths_t ∗paths, unsigned ∗nr)
- int netloc_map_paths_get (netloc_map_paths_t paths, unsigned idx, struct netloc_map_edge_s ∗∗edges, unsigned ∗nr_edges)
- int netloc_map_paths_destroy (netloc_map_paths_t paths)

9.6.1 Typedef Documentation

9.6.1.1 typedef void∗ netloc_map_paths_t

A netloc map path handle.

9.6.2 Enumeration Type Documentation

9.6.2.1 enum netloc_map_paths_flag_e

Flags to be given as a OR'ed set to netloc_map_paths_build().

Note:

> By default only horizontal hwloc edges are reported, for instance cross-NUMA links.

Enumerator:

- ***NETLOC_MAP_PATHS_FLAG_IO*** Want edges between I/O objects such as PCI NICs and normal hwloc objects
- ***NETLOC_MAP_PATHS_FLAG_VERTICAL*** Want edges between normal hwloc object child and parent, for instance from a core to a NUMA node

9.6.3 Function Documentation

9.6.3.1 int netloc_map_paths_build (netloc_map_t *map*, hwloc_topology_t *srctopo*, hwloc_obj_t *srcobj*, hwloc_topology_t *dsttopo*, hwloc_obj_t *dstobj*, unsigned long *flags*, netloc_map_paths_t ∗ *paths*, unsigned ∗ *nr*)

Build the list of netloc map paths between two hwloc objects in two hwloc topologies.

Parameters:

- ***map*** A netloc map.
- ***srctopo*** The hwloc topology of the source server.
- ***srcobj*** The source hwloc object within the source topology.
- ***dsttopo*** The hwloc topology of the destination server.
- ***dstobj*** The destination hwloc object within the destination topology.
- ***flags*** A OR'ed set of netloc_map_paths_flag_e.
- ***paths*** The paths handle returned on success. It must be freed with netloc_map_paths_destroy() after use.
- ***nr*** The number of paths contained in the `paths` handle that is returned on success.

Returns:

0 on success
-1 on error

9.6.3.2 int netloc_map_paths_destroy (netloc_map_paths_t *paths*)

Destroy a previously built netloc map paths handle.

Parameters:

- ***paths*** A paths handle previously returned by netloc_map_paths_build().

Returns:

0 on success

9.6.3.3 int netloc_map_paths_get (netloc_map_paths_t *paths*, unsigned *idx*, struct netloc_map_edge_s $**$ *edges*, unsigned $*$ *nr_edges*)

Get a single path from a previously built netloc map paths handle.

Parameters:

paths A paths handle previously returned by netloc_map_paths_build().

idx The index of the path to return.

edges The array of map edges returned on success. It must not be modified or freed by the caller. It is only valid until netloc_map_paths_destroy() is called.

nr_edges The number of edges returned in the `edges` array on success.

Returns:

0 on success
-1 on error

9.7 Netloc Map API - Misc

Functions

- int netloc_map_find_neighbors (netloc_map_t map, const char ∗hostname, unsigned depth)
- int netloc_map_dump (netloc_map_t map)

9.7.1 Function Documentation

9.7.1.1 int netloc_map_dump (netloc_map_t *map*)

Display the map to stdout (for debugging purposes only).

Parameters:

map A map object.

Returns:

0 on success

9.7.1.2 int netloc_map_find_neighbors (netloc_map_t *map*, const char ∗ *hostname*, unsigned *depth*)

Find the neighbors of the specified node out to a given depth in the network.

Todo

Brice FIXME: get neighbor nodes at a given distance, within any or a single subnet
Brice FIXME: get neighbor nodes with enough cores, within any or a single subnet
Brice This interface is temporary, for debugging

Parameters:

map A map object.

hostname The hostname of the node to start from.

depth The depth into the network to search.

Returns:

0 on success
-1 on error

Chapter 10

Data Structure Documentation

10.1 netloc_data_collection_handle_t Struct Reference

Data Collection Handle.

`#include <netloc_dc.h>`

Data Fields

- netloc_network_t ∗ network
- bool is_open
- bool is_read_only
- char ∗ unique_id_str
- char ∗ data_uri
- char ∗ filename_nodes
- char ∗ filename_physical_paths
- char ∗ filename_logical_paths
- netloc_dt_lookup_table_t node_list
- netloc_dt_lookup_table_t edges
- json_t ∗ node_data
- json_t ∗ node_data_acc
- json_t ∗ path_data
- json_t ∗ path_data_acc
- json_t ∗ phy_path_data
- json_t ∗ phy_path_data_acc

10.1.1 Detailed Description

Data Collection Handle. THe data collection handle off of which the topology data is stored.

10.1.2 Field Documentation

10.1.2.1 char∗ netloc_data_collection_handle_t::data_uri

Data URI

10.1.2.2 netloc_dt_lookup_table_t netloc_data_collection_handle_t::edges

Lookup table for all edge information

10.1.2.3 char∗ netloc_data_collection_handle_t::filename_logical_paths

Filename: Logical Paths

10.1.2.4 char∗ netloc_data_collection_handle_t::filename_nodes

Filename: Nodes

10.1.2.5 char∗ netloc_data_collection_handle_t::filename_physical_paths

Filename: Physical Paths

10.1.2.6 bool netloc_data_collection_handle_t::is_open

Status of the handle : If it is open

10.1.2.7 bool netloc_data_collection_handle_t::is_read_only

Status of the handle : If it is read only

10.1.2.8 netloc_network_t∗ netloc_data_collection_handle_t::network

Point to the network

10.1.2.9 json_t∗ netloc_data_collection_handle_t::node_data

JSON Object for nodes

10.1.2.10 json_t∗ netloc_data_collection_handle_t::node_data_acc

(Internal Use only) Accumulation object used to store JSON data while the node lists are being built in netloc_dc_append_node

10.1.2.11 netloc_dt_lookup_table_t netloc_data_collection_handle_t::node_list

Lookup table for all node information

10.1.2.12 json_t∗ netloc_data_collection_handle_t::path_data

JSON Object for paths

10.1.2.13 json_t∗ netloc_data_collection_handle_t::path_data_acc

(Internal Use only) Accumulation object

10.1.2.14 json_t∗ netloc_data_collection_handle_t::phy_path_data

JSON Object for paths

10.1.2.15 json_t∗ netloc_data_collection_handle_t::phy_path_data_acc

(Internal Use only) Accumulation object

10.1.2.16 char∗ netloc_data_collection_handle_t::unique_id_str

Unique ID String

The documentation for this struct was generated from the following file:

- netloc_dc.h

10.2 netloc_dt_lookup_table_iterator_t Struct Reference

Lookup Table Iterator.

`#include <netloc.h>`

10.2.1 Detailed Description

Lookup Table Iterator. An opaque data structure representing the next location in the lookup table

The documentation for this struct was generated from the following file:

- netloc.h

10.3 netloc_dt_lookup_table_t Struct Reference

Lookup Table Type.

`#include <netloc.h>`

10.3.1 Detailed Description

Lookup Table Type. An opaque data structure to represent a collection of data items

The documentation for this struct was generated from the following file:

- netloc.h

10.4 netloc_edge_t Struct Reference

Netloc Edge Type.

`#include <netloc.h>`

Data Fields

- int edge_uid
- netloc_node_t ∗ src_node
- char ∗ src_node_id
- netloc_node_type_t src_node_type
- char ∗ src_port_id
- netloc_node_t ∗ dest_node
- char ∗ dest_node_id
- netloc_node_type_t dest_node_type
- char ∗ dest_port_id
- char ∗ speed
- char ∗ width
- char ∗ description
- void ∗ userdata

10.4.1 Detailed Description

Netloc Edge Type. Represents the concept of a directed edge within a network graph.

Note:

> We do not point to the netloc_node_t structure directly to simplify the representation, and allow the information to more easily be entered into the data store without circular references.

Todo

> JJH Is the note above still true?

10.4.2 Field Documentation

10.4.2.1 char∗ netloc_edge_t::description

Description information from discovery (if any)

10.4.2.2 netloc_node_t∗ netloc_edge_t::dest_node

Dest: Pointer to neloc_node_t

10.4.2.3 char∗ netloc_edge_t::dest_node_id

Dest: Physical ID from netloc_node_t

10.4.2.4 netloc_node_type_t netloc_edge_t::dest_node_type

Dest: Node type from netloc_node_t

10.4.2.5 char∗ netloc_edge_t::dest_port_id

Dest: Port number

10.4.2.6 int netloc_edge_t::edge_uid

Unique Edge ID

10.4.2.7 char∗ netloc_edge_t::speed

Metadata: Speed

10.4.2.8 netloc_node_t∗ netloc_edge_t::src_node

Source: Pointer to neloc_node_t

10.4.2.9 char∗ netloc_edge_t::src_node_id

Source: Physical ID from netloc_node_t

10.4.2.10 netloc_node_type_t netloc_edge_t::src_node_type

Source: Node type from netloc_node_t

10.4.2.11 char∗ netloc_edge_t::src_port_id

Source: Port number

10.4.2.12 void* netloc_edge_t::userdata

Application-given private data pointer. Initialized to NULL, and not used by the netloc library.

10.4.2.13 char* netloc_edge_t::width

Metadata: Width

The documentation for this struct was generated from the following file:

- netloc.h

10.5 netloc_map_edge_s Struct Reference

`#include <netloc_map.h>`

Public Types

- enum netloc_map_edge_type_e {
  NETLOC_MAP_EDGE_TYPE_NETLOC, NETLOC_MAP_EDGE_TYPE_HWLOC_PARENT, NETLOC_MAP_EDGE_TYPE_HWLOC_HORIZONTAL, NETLOC_MAP_EDGE_TYPE_HWLOC_CHILD,
  NETLOC_MAP_EDGE_TYPE_HWLOC_PCI }

Data Fields

- enum netloc_map_edge_s::netloc_map_edge_type_e type
- union {
    struct {
      netloc_edge_t ∗ edge
      netloc_topology_t topology
    } netloc
    struct {
      hwloc_obj_t src_obj
      hwloc_obj_t dest_obj
      unsigned weight
    } hwloc
  };

10.5.1 Detailed Description

A netloc map edge.

10.5.2 Member Enumeration Documentation

10.5.2.1 enum netloc_map_edge_s::netloc_map_edge_type_e

A netloc map edge type.

Enumerator:

NETLOC_MAP_EDGE_TYPE_NETLOC The edge is a regular network edge.

NETLOC_MAP_EDGE_TYPE_HWLOC_PARENT The edge is a hwloc edge from child to parent.

NETLOC_MAP_EDGE_TYPE_HWLOC_HORIZONTAL The edhe is a horizontal hwloc edge.

NETLOC_MAP_EDGE_TYPE_HWLOC_CHILD The edge is a hwloc edge from parent to child.

NETLOC_MAP_EDGE_TYPE_HWLOC_PCI The edge is a hwloc edge between a PCI and a regular object.

10.5.3 Field Documentation

10.5.3.1 union { ... }

10.5.3.2 hwloc_obj_t netloc_map_edge_s::dest_obj

The target object of a hwloc edge.

10.5.3.3 netloc_edge_t∗ netloc_map_edge_s::edge

A regular network edge.

10.5.3.4 struct { ... } netloc_map_edge_s::hwloc

10.5.3.5 struct { ... } netloc_map_edge_s::netloc

10.5.3.6 hwloc_obj_t netloc_map_edge_s::src_obj

The source object of a hwloc edge.

10.5.3.7 netloc_topology_t netloc_map_edge_s::topology

The netloc topology corresponding to the edge.

10.5.3.8 enum netloc_map_edge_s::netloc_map_edge_type_e netloc_map_edge_s::type

A netloc map edge type.

10.5.3.9 unsigned netloc_map_edge_s::weight

The documentation for this struct was generated from the following file:

- netloc_map.h

10.6 netloc_network_t Struct Reference

Netloc Network Type.

`#include <netloc.h>`

Data Fields

- netloc_network_type_t network_type
- char ∗ subnet_id
- char ∗ data_uri
- char ∗ node_uri
- char ∗ phy_path_uri
- char ∗ path_uri
- char ∗ description
- char ∗ version
- void ∗ userdata

10.6.1 Detailed Description

Netloc Network Type. Represents a single network type and subnet.

10.6.2 Field Documentation

10.6.2.1 char∗ netloc_network_t::data_uri

Data URI

10.6.2.2 char∗ netloc_network_t::description

Description information from discovery (if any)

10.6.2.3 netloc_network_type_t netloc_network_t::network_type

Type of network

10.6.2.4 char∗ netloc_network_t::node_uri

Node URI

10.6.2.5 char∗ netloc_network_t::path_uri

Path URI

10.6.2.6 char∗ netloc_network_t::phy_path_uri

Physical Path URI

10.6.2.7 char∗ netloc_network_t::subnet_id

Subnet ID

10.6.2.8 void∗ netloc_network_t::userdata

Application-given private data pointer. Initialized to NULL, and not used by the netloc library.

10.6.2.9 char∗ netloc_network_t::version

Metadata about network information

The documentation for this struct was generated from the following file:

- netloc.h

10.7 netloc_node_t Struct Reference

Netloc Node Type.

`#include <netloc.h>`

Data Fields

- netloc_network_type_t network_type
- netloc_node_type_t node_type
- char ∗ physical_id
- unsigned long physical_id_int
- char ∗ logical_id
- int __uid__
- char ∗ subnet_id
- char ∗ description
- void ∗ userdata
- int num_edges
- netloc_edge_t ∗∗ edges
- int num_edge_ids
- int ∗ edge_ids
- int num_phy_paths
- netloc_dt_lookup_table_t physical_paths
- int num_log_paths
- netloc_dt_lookup_table_t logical_paths

10.7.1 Detailed Description

Netloc Node Type. Represents the concept of a node (a.k.a., vertex, endpoint) within a network graph. This could be a server or a network switch. The node_type parameter will distinguish the exact type of node this represents in the graph.

10.7.2 Field Documentation

10.7.2.1 int netloc_node_t::__uid__

Internal unique ID: 0 - N

10.7.2.2 char∗ netloc_node_t::description

Description information from discovery (if any)

10.7.2.3 int∗ netloc_node_t::edge_ids

Edge IDs (Internal use only)

10.7.2.4 netloc_edge_t∗∗ netloc_node_t::edges

Outgoing edges from this node

10.7.2.5 char∗ netloc_node_t::logical_id

Logical ID of the node (if any)

10.7.2.6 netloc_dt_lookup_table_t netloc_node_t::logical_paths

Lookup table for logical paths from this node

10.7.2.7 netloc_network_type_t netloc_node_t::network_type

Type of the network connection

10.7.2.8 netloc_node_type_t netloc_node_t::node_type

Type of the node

10.7.2.9 int netloc_node_t::num_edge_ids

Number of edge IDs (Internal use only)

10.7.2.10 int netloc_node_t::num_edges

Number of Outgoing edges from this node

10.7.2.11 int netloc_node_t::num_log_paths

Number of logical paths computed from this node

10.7.2.12 int netloc_node_t::num_phy_paths

Number of physical paths computed from this node

10.7.2.13 char∗ netloc_node_t::physical_id

Physical ID of the node (must be unique)

10.7.2.14 unsigned long netloc_node_t::physical_id_int

10.7.2.15 netloc_dt_lookup_table_t netloc_node_t::physical_paths

Lookup table for physical paths from this node

10.7.2.16 char∗ netloc_node_t::subnet_id

Subnet ID

10.7.2.17 void∗ netloc_node_t::userdata

Application-given private data pointer. Initialized to NULL, and not used by the netloc library.

The documentation for this struct was generated from the following file:

- netloc.h

10.8 netloc_topology_t Struct Reference

Netloc Topology Context.

`#include <netloc.h>`

10.8.1 Detailed Description

Netloc Topology Context. An opaque data structure used to reference a network topology.

Note:

Must be initialized with netloc_attach()

The documentation for this struct was generated from the following file:

- netloc.h

Index