

Contents

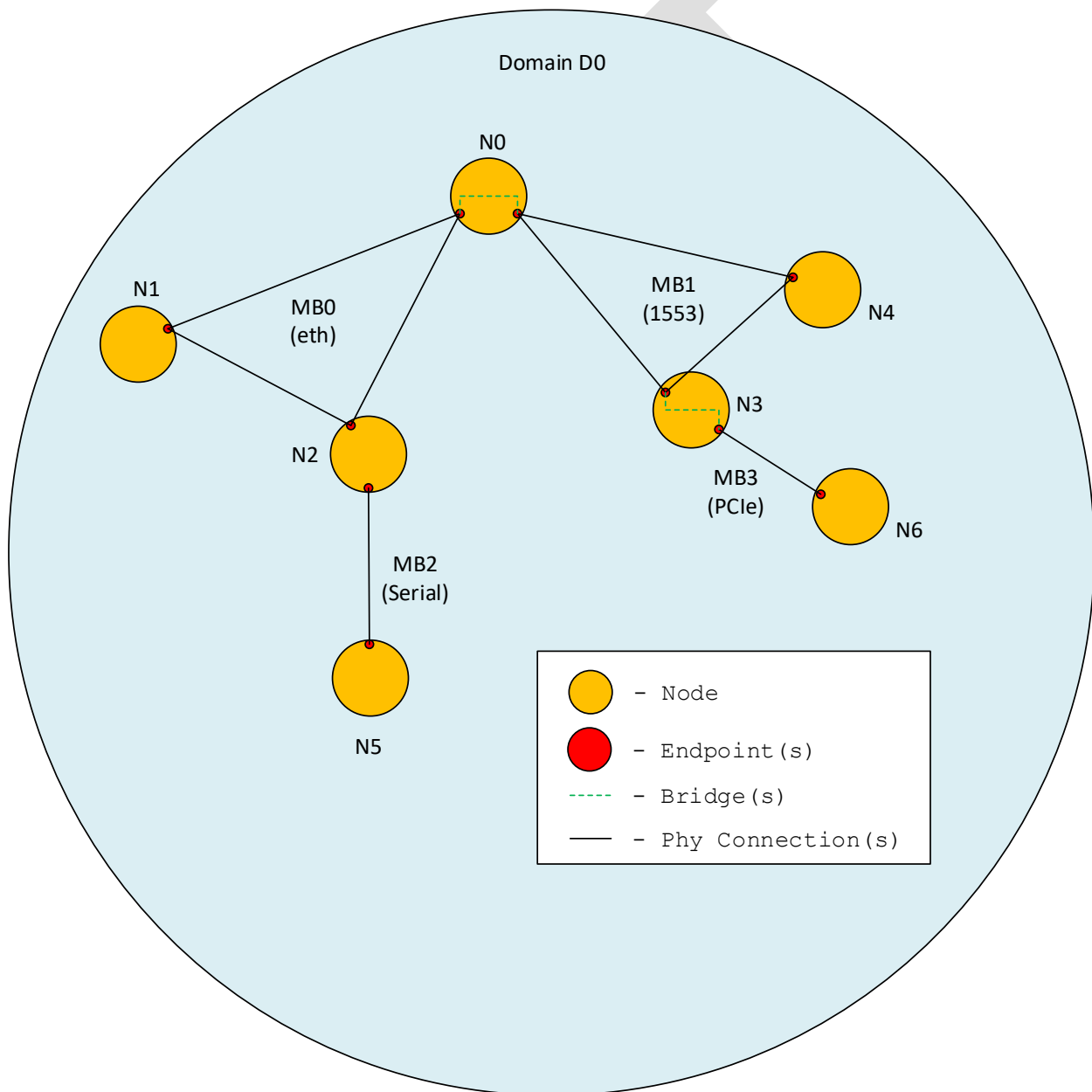
OE Router	3
Object Interface	4
Allocation Methods.....	4
Registration Methods	4
General Use Methods	4
Special Use Methods.....	5
OE Router Buffer.....	7
Object Interface	7
Example.....	8
OE Router Protocols (Stacks)	10
Object Interface	12
OE Router Protocols (Stacks) – New	16
Examples:	17
OE Router Database	23
Database Routing Table	23
Object Interface	24
COE 4.0 Router Node Discovery.....	27
Overview	27
Node Discovery Flow Diagram	29
Node Discovery Message Data Structures.....	30
COE Initialization.....	31
OE Message.....	31
OE Endpoint	31
Object Interface	31
Allocation Methods.....	31
Registration Methods	32
General Use Methods	33
Special Use Methods.....	35
OE Thread.....	35
OE Semaphore	36
OE Mutex	36

OE Event.....	36
OE EventFlag	36
OE Clock Handler.....	36
OE Encryptor	36
OE Synchronization.....	36
Nameserver.....	36
Data Interchange	36
Abstract Data Types	36
C++ Shell.....	36

DRAFT

OE Router

The OE Router facilitates the sending and receiving of labeled messages. It is an abstract object that keeps track of message labels and the COE Endpoints registered to these labels and keeps track of router stack protocols through the use of a database storage object. Routers are what tie Endpoints to a physical Media Binding (MB) so when an Endpoint sends a labeled message, it will be passed to the Router object which will eventually be sent out of the bounded Media Binding. The same goes for when a labeled message comes into a Media Binding, it will make its way up the Router stack and be placed into the receiving Endpoint's queue and/or be bridged (forward) to another Media Binding. The OE Router also incorporates an OE Router Protocol object acting as the top level protocol in the protocol stack.



Object Interface

Allocation Methods

- **OE_Router_Create**
This routine creates a Router object. This routine will normally be called at system initialization immediately after creating the Router Database, as the Database is passed into this routine. The Router is also (optionally) given two Encryptor objects, one for message encryption, and one for message signing. These Encryptors will be used for messages that have encryption and/or signing enabled.
These Encryptor objects should be created with Mutex objects, as the Router is designed to be reentrant.
- **OE_Router_Delete**
This routine deletes a Router object.
- **OE_Router_Delete_Endpoint**
This routine deregisters an endpoint from receiving all labels.

Registration Methods

- **OE_Router_Register_Protocol**
This routine registers a Protocol with the Router. This routine will normally be called at system initialization when setting up the system topology. Media Bindings, which are the bottom of the protocol stack, are handled as Protocols.
Each Router implementation will provide an initialization structure defining its routines and capabilities to the common interface routines for the Router. It also provides a definition for a structure containing a set of initialization parameters to customize its operation.
Both these structures are passed into the Create routine.
- **OE_Router_Deregister_Protocol**
This routine deregisters a Protocol with the Router.
- **OE_Router_Register_Label**
This routine registers an endpoint for a label. Following registration, a copy of any message received with the matching label and domain will be delivered to the specified endpoint. If the message data was sourced in a differing endian than that of the hosting processor, and a data interchange format packet is specified, the endian of the data will be corrected.
- **OE_Router_Deregister_Label**
This routine deregisters an endpoint from receiving a label. Following deregistration, messages received with the matching label and domain will no longer be delivered to the specified endpoint.

General Use Methods

- **OE_Router_Send**

This routine sends a message. The specified message will be delivered to all local Endpoints that have registered for Messages matching the label and domain of the specified Message and to all Bindings that have indicated a need for that label and domain.

Special Use Methods

- **OE_Router_Set_Default**

This routine sets the default Router object. Endpoints interact with only a single Router. The default Router object is used when the Endpoint is created unless another Router is specified when the Endpoint is created. Also, the first created Router is automatically designated as the default Router, so the application does not need to specify a default object, nor assign a Router to any Endpoints unless a topology is used that goes beyond one with just a single Router for all Endpoints.

- **OE_Router_Get_Default**

This routine returns the default Router object.

- **OE_Router_Query_Protocol (Currently Unused, why is this needed???)**

This routine sends a registration query down the Protocol specified to request that one (or all) remote nodes return their registration information.

- **OE_Router_Bridge_Protocol**

This routine bridges one Protocol to another within the Router. This routine will normally be called at system initialization when setting up the system topology. This routine should be called after the two protocol stacks are bound (which defines the protocol stack itself), but before the protocol stacks are registered with the Router.

Each bridge is a one-way bridge. Registrations and registered messages received on this router from the Source_Protocol protocol stack will be repeated to the Destination_Protocol protocol stack; this is the definition of a "bridge". If a full-duplex bridge is desired (bridging on both directions from two protocol stacks), two bridges can be defined, one for each direction.

As many bridges as desired can be created. This allows the capability to constrain the message traffic to be bridged to contain only what is absolutely needed. Once a bridge is created, it cannot be deleted, unless the router itself is deleted. The bridge is intended to only be created at router creation time, based on the specific system or application network topology.

A range of label and domain values can be specified to restrict the values of labels and domains that will be bridged. This allows an application to restrict the bridging to a subset of all message traffic.

- **OE_Router_Set_Label_Options**

This routine sets the current options for a specific message label. These options will be applied to all messages with the indicated label. This capability is used to enable encryption and/or signing of messages on an individual label basis across the system. One of the bit flags is provided to specify whether encryption and/or signing will be applied to all messages with the indicated label, or only those messages with that label that are being transmitted from the

executing host system via a media communication binding. The default value (bit value of 0) will be only transmitted messages. The value of `OE_Message_Flag_All_Messages` will force all messages to be encrypted and/or signed when the message data is transferred from the Message object.

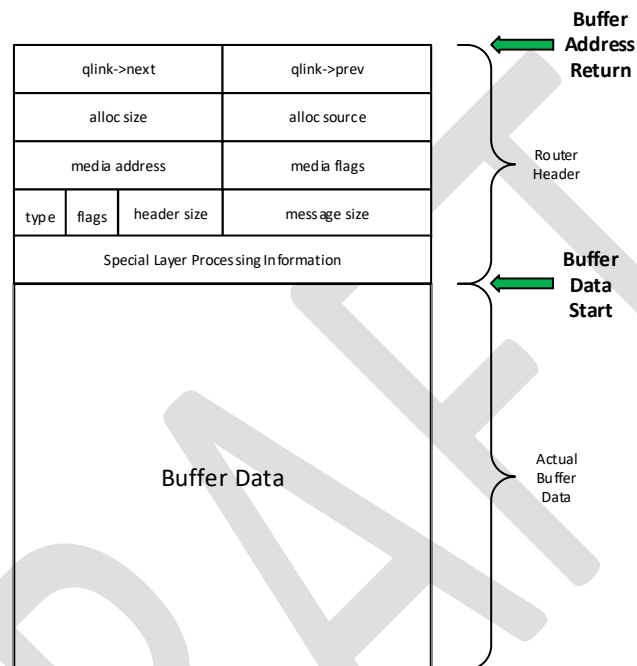
Note that the option flag definitions are defined in the `OE_message.h` header file, as these are the same message options that are defined by the OE message object.

- **OE_Router_Get_Router**

This routine returns the router given a protocol module. This allows a protocol module that has been registered with a router to retrieve the router object. This routine should only be used by a routine after it has been registered.

OE Router Buffer

The OE Router Buffer is the buffer structure that is passed between protocol layers. The given structure below describes the layout of the Buffer, however this information will be hidden from the protocol layers and requires OE Router Buffer functions to get specific information from the buffer. (NOTE: Keep in mind this definition can change, this is a draft). By keeping the Special Processing Information located in the Router Header, this frees the Protocol layer from having to dissect the Buffer data looking for how to determine if the buffer needs special attention.



Object Interface

Allocate Methods

- **OE_Router_Buffer_Create**
This routine creates a Router Buffer object. This routine will normally be called at system initialization during setup of the routing subsystem, as Protocol Buffers must be supplied to Routers, Media Bindings, and other Protocol Modules.
- **OE_Router_Buffer_Delete**
This routine deletes a Router Buffer object. It is important to note that when Protocol Modules are created, they are given a Protocol Buffer for its use in managing buffers. The Router Buffer should not be deleted until all dependent modules are deleted.

Registration Methods

General Use Methods

- **OE_Router_Buffer_Obtain**

This routine obtains a router protocol buffer from the router's buffer pool. The calling routine (thread) will be suspended if a buffer is not available, until one becomes available.

- **OE_Router_Buffer_Release**

This routine releases a router protocol buffer from the router's buffer pool.

- **OE_Router_Buffer_Get_Allocations**

This routine returns the number of buffers currently allocated from the buffer pool.

- **OE_Router_Buffer_Get_Free_Buffers**

This routine returns the number of buffers currently available to be allocated from the buffer pool.

The operation of this routine may depend on the memory allocation being used by the buffer module.

Special Use Methods

- **OE_Router_Buffer_Get_Statistics**

This routine returns the statistics structure from the underlying allocation module. The underlying buffer allocation/management module may provide statistics on its memory allocation activity. This routine will return the address of the structure. The contents of the structure is understood by the specific buffer module configured within the protocol buffer object.

- **OE_Router_Buffer_Add_Header_Space**

This routine is an internally used interface for protocol modules to declare that they use additional header area. The additional area is a number of bytes required above the base allocation.

Example

Given a buffer pool definition, when creating a Router Buffer there is some extra memory that is instantiated. The buffer pool definition list each pool as number of buffers followed by a buffer size.

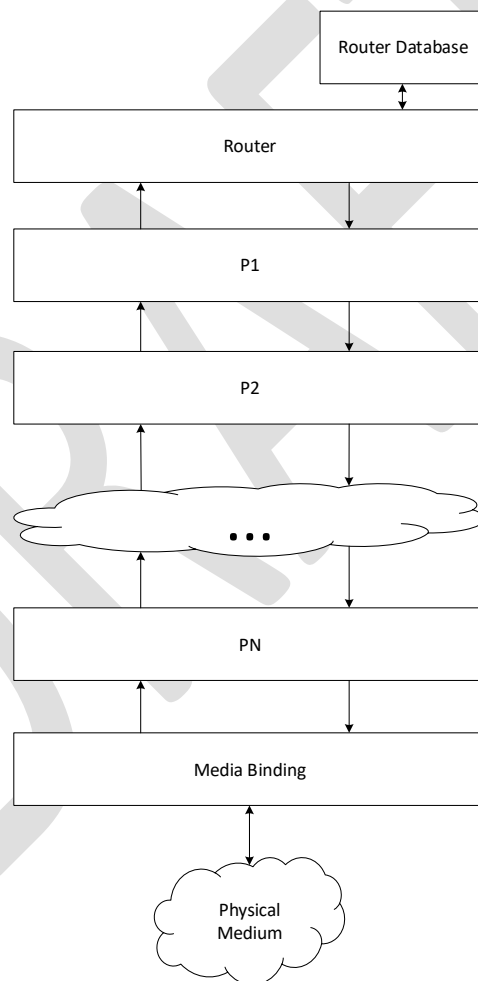
```
266 static Pool_Definition pools[] =
267 /*
268 // The pools is a structure containing the number of buffers, followed by the size of the
269 // buffer. There can be as many sizes as needed, but this structure *MUST* be ordered in
270 // ascending size.
271 //
272 // Each entry is of the form { number of buffers, size of buffers }
273 // This structure is only used for linked list buffer management.
274 */
275 {
276     { 100, 128 },
277     { 50, 256 },
278     { 10, 8192 }
279 };
280
```


In the definition there are three pools with the first pool having 100 buffers of size 128, the second pool having 50 buffers or size 256, and the last pool having 10 buffers of size 8192. Given this information, it may seem that only 107520 bytes are allocated, however for each size allocated the size of a ROUTER_HEADER and OE_Message_Header is added to the size, which in this example would come out to 119040 bytes allocated:

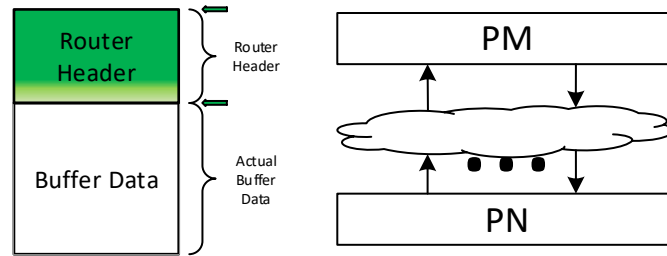
100	X	128 + ROUTER_HEADER+OE_Message_Header	=	20,000
50	X	256 + ROUTER_HEADER+OE_Message_Header	=	16400
10	X	8192 + ROUTER_HEADER+OE_Message_Header	=	82640
				119,040

OE Router Protocols (Stacks)

A router stack has at a minimum 2 protocol layers, a router protocol layer and a media binding layer, and can (conceptually) have any number of protocol layers in between. The layers pass OE Router Buffers up and down the protocol stack. When a layer receives a buffer, it must determine what type of processing must be done on the buffer, this allows for designs that want to couple, or decouple layers. In an uncoupled design, the processing information in the buffer would be specified to “normal” processing. In a coupled design, where one layer needs to talk to another layer for specific capability, it would create a special processing request, in an OE Buffer, and pass it up/down to the recipient protocol layer. While passing through other layers, each protocol layer would be required to check the processing instruction, and if it is a special processing request and not meant for them, they would not apply any special processing to that Buffer and continue to pass that Buffer up/down the stack.

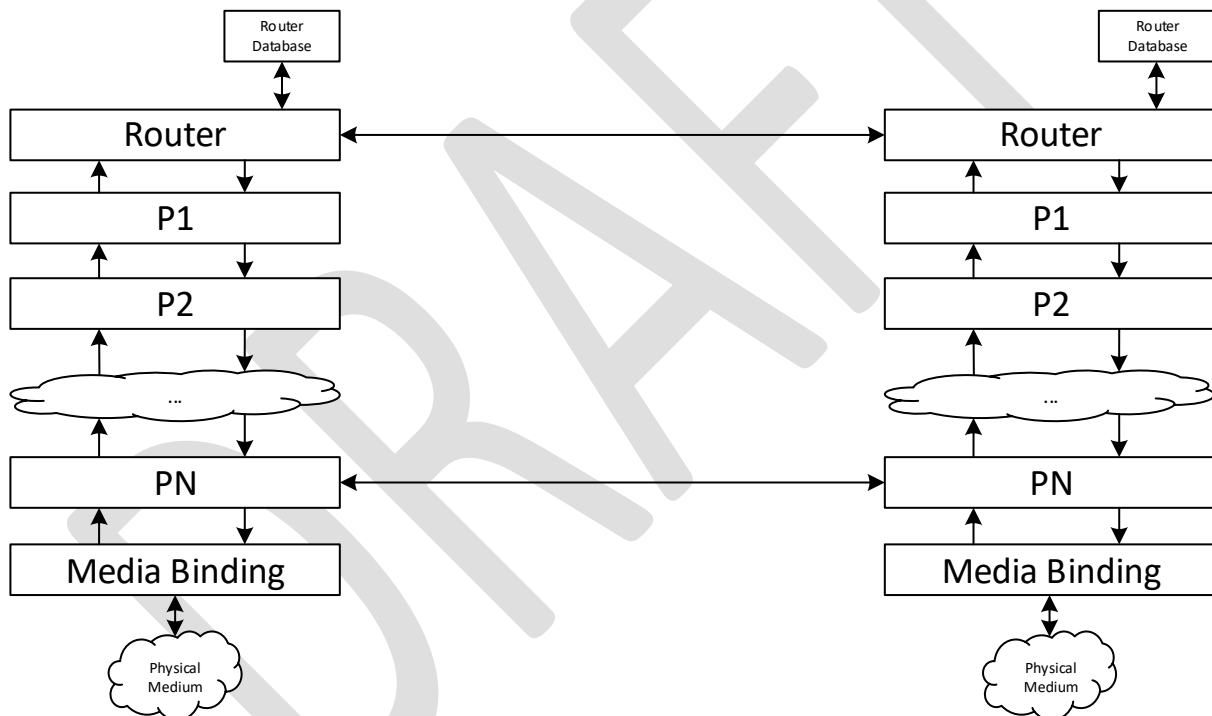


Communications up and down the stack is done through the Router Header, there are protocol commands that specify whether normal processing takes place, or if there is a special processing required, which allows one protocol to talk to another protocol in the same stack.

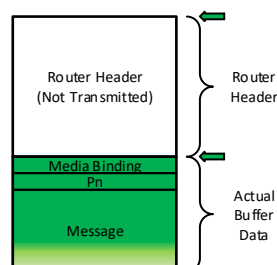


If it is desired to couple two protocols together, then the use of the Router Header is where this communication would take place, and then the Buffer Data is not meant to be processed by any protocol in between the two protocols.

If a protocol wants to talk to the “same” protocol in a different stack, then it must perform normal processing down the stack and utilize the Buffer data to pass it’s information.

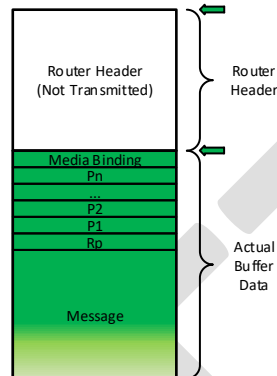


This diagram shows two cases, 1) the PN protocol wants to talk to another PN Protocol, then it would send it’s communications information as follows (the OE Buffer manipulation would look like the following):



Since protocol PN knows how to talk to PN (itself), then it in theory can generate a message without needing the upper layer to invoke it, so that when the PN layer on the other side receives the Buffer, it will know that it doesn't need to continue passing up the protocol stack (since it know how to talk to itself).

2) If the Router Protocol want to talk to another Router Protocol, then it would send it's communications information as follows (the OE Buffer manipulation would look like the following):



By implementing a layer processing commands in the Router Header, this allows the bypassing of layers without having to dig into the actual buffer to determine if there are special processing commands. Also, if there is buffer manipulation in a specific protocol, determining special processing will be impossible without explicitly coupling each layer to the unique layer that is implementing this capability.

Object Interface

Allocation Methods

- **OE_Router_Protocol_Create**

This routine creates a Router Protocol object. This routine will normally be called at system initialization during setup of the routing subsystem, as Protocols are used to connect Routers, Media Bindings, and Protocols.

- **OE_Router_Protocol_Delete**

This routine deletes a Router Protocol object. It is important to note that when Protocol Modules are created, they are created in a stack, where each Protocol Module is linked with other Protocol Modules. Deleting a single Module will destroy this chain, and cause very undesirable consequences if messages are sent up/down the stack. The entire stack should be deleted, starting with deleting the Router Database (which will shut off the transmit capability), then deleting the Media Bindings (shutting off any receive). Then all other Protocol Modules can be deleted.

Registration Methods

- **OE_Router_Protocol_Bind**

This routine creates binds a Protocol Module in a Protocol Stack. Binding is the specification of a Protocol Module's position in a Protocol Stack.

Protocol Binding is done by specifying a Protocol Module immediately above the given Module, and another Protocol Module immediately below the given Module.

Stringing Protocol Modules together in this way forms the Protocol Stack.

Each Protocol Module is also given a Protocol Buffer, from which it can allocate data buffers.

- **OE_Router_Protocol_Unbind**

This routine unbinds all Protocol Modules in a Protocol Stack. This breaks the connection between all the Protocol Modules. This is usually done after the Protocol Stack is disabled, in anticipation of deleting each of the Protocol Modules.

- **OE_Router_Protocol_Enable_Stack**

This routine enables a Protocol Stack. This routine will normally be called at system initialization during setup of the routing subsystem, after a series of Protocol Modules are created and bound. This allows the entire stack to be set up before the actual communication media are enabled to begin transmissions.

Each protocol module will have been created and initialized but should refrain from enabling operation (especially the Media Bindings) until they are explicitly enabled. Enabling the Protocol Modules is an indication that they can interface with their upper and lower Modules. This call will enable the entire stack (all the Protocol Modules in the stack) starting with the bottom, going to the top. This call gives Protocol modules the go-ahead to communicate with their peers and/or other modules in the stack.

- **OE_Router_Protocol_Disable_Stack**

This routine disables a Protocol Stack. This routine will normally be called during system shutdown before Protocol Modules are deleted. This call will disable the entire stack (all the Protocol Modules in the stack) starting with the top, going to the bottom.

General Use Methods

- **OE_Router_Protocol_Transmit**

This routine sends a data buffer down the protocol stack. The protocol module is handed a message buffer, containing metadata and a message stream to process and/or send down to the next protocol module, ending with a media binding protocol module which places the data on the medium.

The protocol module may do some processing on the data, or it may just pass the data to the next protocol module in the protocol stack. It is required to implement a specific protocol whether it passes the data, modifies the data being passed, or heavily processes the data.

There are two possible returns from this routine, ignoring error conditions. A status return of OE_Success indicates that the data buffer has been sent down the protocol stack to its point of consumption, has been released back to the system, and the calling module is free to send another buffer down the stack. If this module is the consuming module for the message, it must

release the message buffer and reply with the OE_Success. If this module merely passes the buffer on, it will call the downstream module (which may call further downstream modules), and return the status return.

If the module needs to hold further calls to its transmit, It will return a status of OE_Protocol_Operation_Pending. This indicates that no more calls to transmit should be made until the upstream module's OE_Router_Protocol_Transmit_Acknowledge routine is called. In this case, the buffer is held by the protocol module until it is processed. Note that sending a transmit down the protocol stack may result in a OE_Protocol_Operation_Pending status at some point which will be propagated up the stack. The protocol module should keep track of whether it is expecting an acknowledgment, or it is merely passing one up the stack.

- **OE_Router_Protocol_Receive**

This routine sends a data buffer up the protocol stack. It works the same as the OE_Router_Protocol_Transmit routine, but up the stack instead of down.

- **OE_Router_Protocol_Transmit_Acknowledge**

This routine acknowledges data being sent down the stack to the upstream protocol. This routine is called by the downstream module when it completes the processing of transmit data and is able to accept additional data.

The downstream module has a choice of merely acknowledging the transmit indicating that it is ready for another call to its transmit routine, or it can acknowledge the transmit while requesting the next transmit buffer. It does this by providing a pointer value to return a router buffer (router header) for the next packet. This allows the upstream and downstream modules the opportunity to avoid thread context switches. If the downstream module isn't implemented in such a way to provide this processing, or there are no more buffers for processing, a NULL value will be returned. If a next buffer is available, it will be returned in this pointer. If the downstream module doesn't want to have the next buffer returned, it should provide a NULL value for this pointer.

The module can also just decline to return a message (per its implementation) by using the same technique or returning a NULL value for the next router buffer.

If a value for the message parameter is provided, this is a "pull" of the data. The message buffer fields and buffers will be filled out just as a message would be before a transmit call via the OE_Router_Protocol_Transmit call, which is a "push" of the data. This form of an acknowledge that pulls the data can avoid potential thread context switches, since the Protocol Module will return the next data packet directly, rather than waiting for a context switch to enable another thread to push that packet. It remains the Protocol Module's responsibility to ensure that whether data is pushed or pulled, data is provided only once.

Note, that any one protocol module can be implemented to use data pushes, data pulls, or a combination of pushes and pulls. However, all Protocol Modules must cope with receiving a OE_Protocol_Operation_Pending status followed by an acknowledgement.

Once data is pulled by a Protocol Module, the Module providing the data must consider that the receiving Module will not be able to accept another packet until the next acknowledge, whether that next one results in a push or a pull. In other words, once a data pull is done, the Protocol

Module providing the data must wait for a successive acknowledgment just as if it called the receive function and received a `OE_Protocol_Operation_Pending` status;

- **OE_Router_Protocol_Receive_Acknowledge**

This routine acknowledges data being sent up the stack to the downstream protocol. This routine provides the same functionality as the `OE_Router_Protocol_Transmit_Acknowledge` routine does for the downstream data.

Special Use Methods

- **OE_Router_Protocol_Extended_Routine**

Specific implementations of the underlying protocol module may provide capabilities that go beyond the functional specification of the common API. This routine provides an interface for accessing those capabilities. It is the responsibility of the application to ensure that the capabilities are properly configured and called; this routine merely provides a conduit through which the calls can be made.

OE Router Protocols (Stacks) – New

In the Router Protocol implementation, the handle method allows the user to register specific functionality/methods (drvOps – driver operations) that are to be used on a specific protocol type. It has been identified that there are three (3) major protocol types that are used in a router protocol stack, Router Protocol, Media Binding Protocol, and a Layer Protocol. Not all protocols are necessary to implement router stack functionality, and in most cases, only the Router protocol and Media Binding protocols are required.

Functionality (Router Protocol Binding) has been separated out to promote reuse, so that one can register a common set of functionality which can be used on multiple instances; example, is a system has two instances of the Routing Protocol, but they are tied to different Media Binding instances. Thus the two Routing protocols use the same functionality but they have different private data, and when creating the separate instances of the Routing Protocol object, they can both point to the same instance of the Router Protocol Binding Handle.

Router Protocol Binding (Definition) is created by specifying what Protocol type the functionality is attached to (Router Protocol, Layer Protocol, or Media Binding Protocol), a globally unique id, and a structure of object functionality (methods). By supporting a globally unique id, this allows for dynamic stack creation, which can be very useful in a CMIT environment (debugging), as this will allow CMIT to either construct a stack locally that mimics a remote stack device (all the layers), or it can command a remote device to create a stack dynamically with the specified stack layers to support communications.

To create an instance of a Protocol Binding, one has to specify the globally unique id of the Protocol Binding Definition, or the Handle ID returned when creating the Protocol Binding Definition, the corresponding data that is specific to that definition (termed configuration data), and possibly control flags.

Protocol Stacks are created by specifying a list of Protocol Binding instances. The list specifies the number of handles in the list, followed by Router Protocol Instance Handle, followed by the Media Binding Protocol Instance Handle, and finally followed by any layer protocol Handles (to be bound in the specified order – top down). The list is not in the exact binding order due to all stacks requiring the routing protocol and media binding protocol, but not requiring a layer protocol. The flags will also help control parsing the list in the event of creating a partially completed stack without binding the layers, by default, when creating a Router Protocol Stack, it will also bind all the layers provided in the list.

Once a Protocol Stack is created, the handle can then be passed to an endpoint to support labeled message communications. Due to this process still being somewhat involved, a simple implementation of the old COE 3.0 Socket Media Binding functionality is provided for those who wish to use the basic provided Router Protocol and Media Binding Protocol (for Ethernet). This function also allows for the user to add layers in between the Router Protocol and Media Binding, however this requires a bit more complexity, but that is only for the layers they are providing.

[Currently spinning the idea of using a single global Router Database, not seeing the benefit of having multiple different definitions of a database for different Routers ... Need to discuss]

Examples:

Creating Router Protocol Binding (Protocol Methods mapped to a Unique ID and Type)

```
HANDLE_ID  
Create(PROTOCOL_TYPE,  
       UNIQUE_ID,  
       PROTO_DEV_OPS);
```

```
HANDLE_ID  
Find(UNIQUE_ID, // IN  
     FLAGS);    // IN
```

```
H1 =  
Create(PROTO_MEDIA_TYPE,  
       0x10,  
       mb1DevOps);
```

```
H2 =  
Create(PROTO_ROUTER_TYPE,  
       0x2C,  
       rp1DevOps);
```

```
H3 =  
Create(PROTO_LAYER_TYPE,  
       0x3A,  
       lp1DevOps);
```

Router Protocol
Binding
(Definition)

HANDLE

MB ₁ 10 mb1DevOps
MB ₂ 16 mb2DevOps
RP ₁ 2C rp1DevOps
RP ₂ 1F rp2DevOps
...
P ₁ 3A lp1DevOps

0

1

2

3

N-1

Creating Router Protocol Instances (Protocol Config Data mapped to Protocol Binding)

```
HANDLE_ID  
Create(UNIQUE_ID,    // IN  
      CONFIG_DATA,  // IN  
      FLAGS);        // IN
```

```
HANDLE_ID  
Find(UNIQUE_ID,    // IN  
     CONFIG_DATA,  // IN  
     FLAGS);        // IN
```

```
Handle1 = Create(0x2C,    // RP ID  
                rplConfig, // RP CFG  
                UNIQ_ID);
```

```
Handle2 = Create(H1,      // MB Handle ID  
                mblConfig, // MB CFG  
                0);
```

```
Handle3 = Create(0x3A,    // LP ID  
                lplConfig, // LP CFG  
                UNIQ_ID);
```

Router Protocol Instance HANDLE

protoBindingPtr protoPrivateData	0
protoBindingPtr protoPrivateData	1
protoBindingPtr protoPrivateData	2
protoBindingPtr protoPrivateData	3
...	
protoBindingPtr protoPrivateData	N-1

Media Bindings (Address data mapped to Protocol Binding)

```
HANDLE_ID  
Create(UNIQUE_ID, // IN  
      ADDR_DATA, // IN  
      FLAGS);    // IN
```

```
HANDLE_ID  
Find(UNIQUE_ID, // IN  
     ADDR_DATA, // IN  
     FLAGS);    // IN
```

```
Handle0 = Create(0x10, // MB ID  
                node0Addr, // MB ADDR  
                (UNIQ_ID | NO_CHECK));
```

```
Handle1 = Create(0x10, // MB ID  
                node1Addr, // MB ADDR  
                (UNIQ_ID | CHECK_ADDR_FMT));
```

```
Handle5 = Create(H1, // MB Handle ID  
                node3Addr, // MB ADDR  
                CHECK_ADDR_FMT);
```

Media Binding
Address
HANDLE

addrData	0
addrData	1
addrData	2
addrData	3
...	
addrData	N-1

Router Stack Creation (Binding a list of Protocol Handles)

```
Handle1 = Create(stk1553Cfg,
                 (RP_PROTO | MB_PROTO));
```

```
Handle2 = Create(ethSvrCfg,
                 (MB_PROTO | NO_BIND));
```

```
HANDLE_ID
Create(CONFIG_DATA, // IN
       FLAGS);      // IN
```

```
BOOL
Add(HANDLE_ID, // IN
    CONFIG_DATA, // IN
    FLAGS);     // IN
```

```
BOOL
Bind(HANDLE_ID, // IN
     FLAGS);     // IN
```

```
HANDLE_ID
Find(PROTO_INST_ID, // IN
     FLAGS);         // IN
```

Router Stack
Instance
HANDLE

protoInstHandle[]	0
protoInstHandle[]	1
protoInstHandle[]	2
protoInstHandle[]	3
...	
protoInstHandle[]	N-1

Router Stack
CONFIG_DATA

Num Handles
RP Handle
MB Handle
LP1 Handle
...
LPN Handle

stk1553Cfg
CONFIG_DATA

2
7 (RP Handle)
5 (MB Handle)

ethSvrCfg
CONFIG_DATA

1
1 (MB Handle)

```
Handle1 =
Create(stk1553Cfg,
      (RP_PROTO | MB_PROTO));
```

```
Handle2 =
Create(ethSvrCfg,
      (MB_PROTO | NO_BIND));
```

Socket Media Binding Logic (Using Handle Method)

```
// RP_Config_Data - Configuration data for Router Protocol
// MB_Config_Data - Configuration data for Media Binding
// List<LP_Config_Data> - A list of Configuration data for
//                        Layered Protocols in between Router
//                        and Media Binding Protocol
SMB_CONFIG(RP_Config_Data, MB_Config_Data, List<LP_Config_Data> = NULL)
{
    // Create Protocol Definitions
    // This allows subsystems to search for defined
    // protocols by their unique ID, and get the
    // device operations (driver functions) from this
    // interface
    SMB_RP_H = Create(PROTO_MEDIA_TYPE, // Media Binding RP
                     SMB_UNIQ_ID,
                     getSmbDevOps()); // getSmbDevOps is a global function that returns
                                     // the object methods for the Socket Media Binding

    SRP_RP_H = Create(PROTO_ROUTER_TYPE, // Router Protocol RP
                     SRP_UNIQ_ID,
                     getSrpDevOps()); // getSrpDevOps is a global function that returns
                                     // the object methods for the Socket Router Protocol

    for i in List<LP_Config_Data>.Count() :
        LRP_RP_H[i] = Create(PROTO_LAYER_TYPE, // Layer Protocol RP
                             List<i>.UNIQ_ID,
                             List<i>.devOps);
    end for

    // Create Protocol Instances
    // This will create a single entry for each set
    // of private data, this allows the re-use of protocol
    // functionality while maintaining different state information
    // through the use of the Private Data
    SMB_RP_I_H = Create(SMB_UNIQ_ID,
                       MB_Config_Data,
                       UNIQ_ID);

    SRP_RP_I_H = Create(SRP_UNIQ_ID,
                       RP_Config_Data,
                       UNIQ_ID);

    for i in List<LP_Config_Data>.Count() :
        LRP_RP_I_H[i] = Create(List<i>.UNIQ_ID,
                               List<i>.Config_Data,
                               UNIQ_ID);
    end for

    // The stack object is just a list of protocols instances
    // to be bound that makes up a Router Stack Instance
    ROUTER_STACK_CONFIG ethSrvCfg; // This is just a list of integers

    ethSrvCfg.insert_at_end(2 + List<LP_Config_Data>.Count());
    ethSrvCfg.insert_at_end(SRP_RP_I_H);
    ethSrvCfg.insert_at_end(SMB_RP_I_H);
}
```

```
for i in List<LP_Config_Data>.Count() :
    ethSrvCfg.insert_at_end(LRP_RP_I_H[i]);
end for

// Creating the Stack Handle will Bind all
// the layer protocols in the order defined
// With the Router Protocol as the 1st element,
// the Media Binding as the 2nd element, and all
// the Layer Protocols (in order) 3rd to last.
StackH = Create(ethSrvCfg,
                (RP_PROTO | MB_PROTO | LP_PROTO));

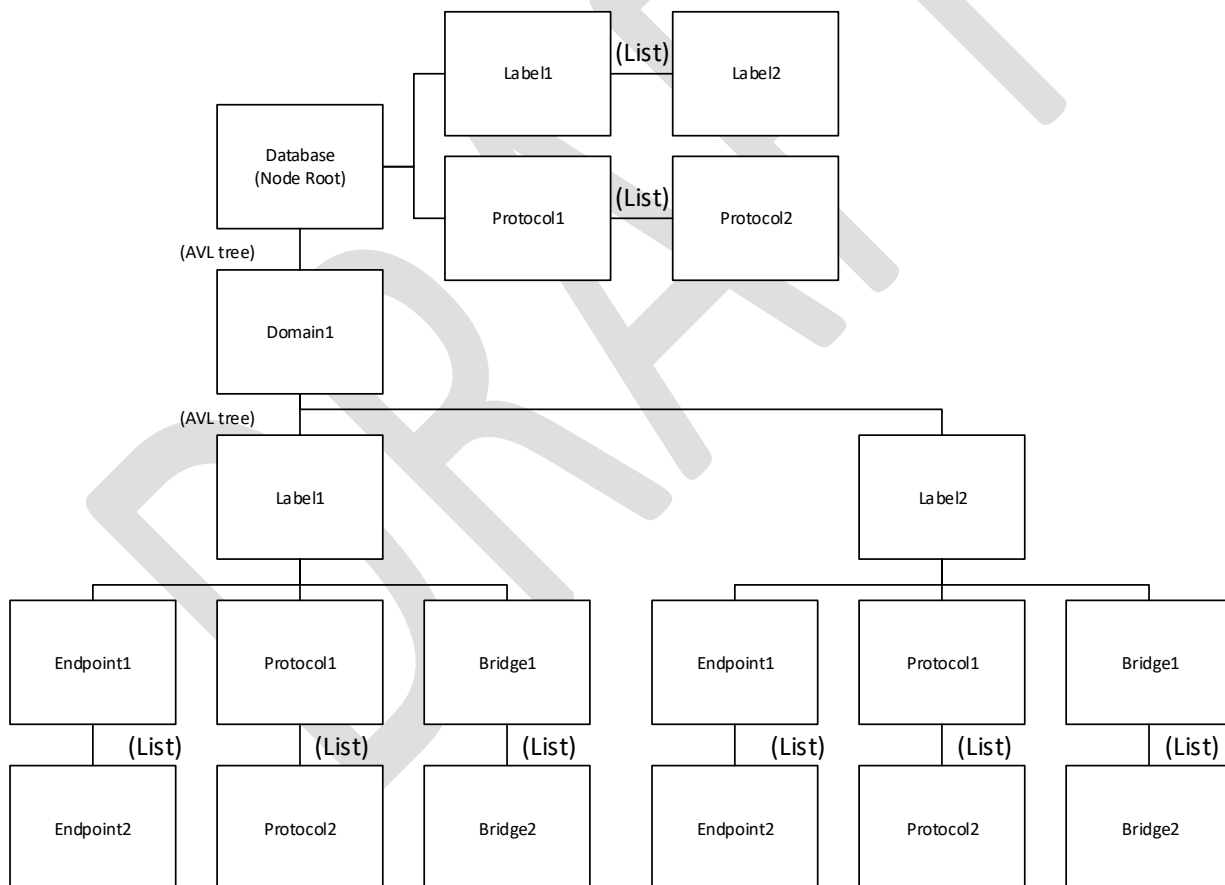
return StackH;
}
```

OE Router Database

An OE Router Database is an object that contains the necessary information that a router can use to maintain routing information for sending messages to all registered destinations. Routing information is stored in a routing table for all endpoints, routing protocols, and bridges for a given domain and label.

Database Routing Table

The database routing table is divided into 2 main sections the lists and the AVL tree structure. The lists are a complete set of all the labels and protocols that have been created inside the database regardless of if they have been registered or which domain they are registered on. The AVL tree structure is a definition of which items have been registered and where. This tree structure begins with a tree of domains. For each domain a tree of labels manages the information registered on that domain. Each label consists of 3 linked lists of items registered: Endpoints, Protocols, and bridges.



When the routing protocol receives a message (from the lower protocol), it queries the database based upon the domain and label, and iterates through the list of endpoints copying the message (with the given domain:label) into each endpoints receive queue. It then iterates through the list of bridges for the given domain:label and forwards the message to each bridge router protocol.

When the routing protocol sends a message (via the Endpoint.Send method), it queries the database based upon the domain and label, and iterates through the list of Protocols (Router Protocols) to send the given labeled message to each registered Node.

Object Interface

The following methods are user defined and registered when a specific Router Protocol is instantiated.

- **OE_Router_Database_Create**
This routine creates a Router Database object. This routine will normally be called at system initialization immediately prior to creating the Router, as the Database is passed into the Router when the Router is created.
Each Database implementation will provide an initialization structure defining its routines and capabilities to the common interface routines for the Router. It also provides a definition for a structure containing a set of initialization parameters to customize its operation.
Both these structures are passed into the Create routine.
- **OE_Router_Database_Delete**
This function deletes the Database object and all resources associated with it.
- **OE_Router_Database_Label_Configure**
This routine configures a label entry in the database with data used by the routers.
The data is meaningless to the database; it is merely managed for the use of the router.
- **OE_Router_Database_Register_Endpoint**
This routine registers a local Endpoint for a specified label. This routine is normally called by a Router in response to a request by an Endpoint to register for a label. Registered Endpoints will be returned from the Database when the routing for that specified label is requested.
- **OE_Router_Database_Deregister_Endpoint**
This routine deregisters a local Endpoint for a specified label. This routine is normally called by a Router in response to a request by an Endpoint to deregister for a label. Deregistration will remove the Endpoint from the list of destinations when the routing for that specified label is requested.
- **OE_Router_Database_Delete_Endpoint**
This routine deregisters a local Endpoint from all labels for which it has been registered. This routine is normally called by an endpoint before it is deleted as part of its deletion sequence.
- **OE_Router_Database_Add_Protocol**
This routine adds a Protocol for registrations. This routine is normally called by a Router in response to a request by the application to add a Protocol stack to the messaging system.
- **OE_Router_Database_Remove_Protocol**
This routine removes a Protocol from registrations. This routine is normally called by a Router in response to a request by the application to remove a Protocol stack from the messaging system.

- **OE_Router_Database_Register_Protocol**
This routine registers a Protocol for a specified label. This routine is normally called by a Router in response to a request by a remote Protocol to register for a label. Registered Protocols will be returned from the Database when the routing for that specified label is requested.
- **OE_Router_Database_Deregister_Protocol**
This routine deregisters a Protocol for a specified label. This routine is normally called by a Router in response to a request by a remote Protocol to deregister a label. Deregistration will remove the Protocol from the list of destinations for the label's routing.
- **OE_Router_Database_Delete_Protocol**
This routine deregisters an OE_Router_Protocol from all labels for which it has been registered. This routine is normally called by a Protocol when it disconnects.
- **OE_Router_Database_Register_Bridge**
This routine registers a bridge to a Protocol. Any message received by a Router can be echoed to a bridge. The creation of a bridge would commonly be done at system initialization based on the system topology, but can be done dynamically during operation in response to changing conditions.
- **OE_Router_Database_Deregister_Bridge**
This routine deregisters a bridge from a Protocol.
- **OE_Router_Database_Query_Open**
This routine starts a label query by setting up the Database to return routing information for a label. This call is made before receiving individual routing items.
Opening a Query allows the Database to locate the information for the specified label, verify that destinations are available, as well as to take whatever actions are needed to prepare for maintaining data integrity as a Router distributes messages over the set of destinations.
- **OE_Router_Database_Query_Next**
This routine returns the next destination for the previously setup label query. Following a call to OE_Router_Database_Query_Open, repeated calls to this routine will return the set of destinations to which to send the message. Each call returns a single destination as a pointer to an OE_Router_Label_Destination structure. When this pointer is NULL, the end of the list has been reached. Each destination is either an Endpoint or a Binding.
- **OE_Router_Database_Query_Close**
This routine terminates the label query. This is made following all calls to the OE_Router_Database_Query_Open routine to release any resources that may have been allocated for the query.

- **OE_Router_Database_Query_Registration**

This routine requests registration information. This query returns information on registrations and deregistrations that have been made for endpoints and bridges in the database. This information is meant for the router to be able to send the registration/deregistration messages throughout the system. The database ensures that information for registrations is properly ordered and consistent to agree with current information within the database. Registration requests are queued by the database; it is the responsibility of the router to watch for availability of registration information that must be handled. An optional Event Flag is passed in (along with a mask and a time interval) to allow the routine to block to allow a router thread to pend for registration information to process. This routine is meant to be used by a single client, unlike the destination queries.

- **OE_Router_Database_Query_Registration_Close**

This routine terminates a pending registration query. This call is made to terminate any possible open call to the OE_Router_Database_Query_Registration routine. If a thread is currently pending on registration information, that call will be terminated with an error. This allows a graceful shutdown of a pending thread, usually from a router.

- **OE_Router_Database_Extended_Routine**

Specific implementations of the underlying database may provide capabilities that go beyond the functional specification of the common API. This routine provides an interface for accessing those capabilities. It is the responsibility of the application to ensure that the capabilities are properly configured and called; this routine merely provides a conduit through which the calls can be made.

COE 4.0 Router Node Discovery

Overview

Node discovery is meant to be configurable, to either support a static or dynamic discovery. What this means is in a static discovery the Node has a configuration of all the nodes it will communicate with, any traffic from nodes outside its list will be ignored. In dynamic discovery, the node will allow nodes that are not in its' list to connect and will respond to traffic with these nodes.

Node discovery only required 4 messages, the Label Registration Message, Request Label Registration Message, Node Registration Message, and Request Node Registration Message.

Label Registration Message is meant to send a list of all registered Labeled Messages from one Node to another. Request Label Registration Message is meant to get the list of all registered Labeled Messages from a remote Node, it is meant to trigger the Remote Node to send a Label Registration Message to the requesting Node.

Node Registration Message is meant to notify a remote Node of all the Nodes the Local Node knows about. This is the message that builds the network of nodes. The Request Node Registration Message is meant to get the list of all known Nodes from a remote Node, which is meant to trigger the Remote Node to send a Node Registration Message to the requesting Node.

Each node has a list of registered labels and the size of the message for each of these labels. This information must be propagated to all other nodes participating in the domain. To reduce message processing, a SHA1 hash is computed for all the labels and sizes (in increasing order of the label id) and is sent with the Label Registration Message as well as the Node Registration Message. This is sent both in the Label Registration Message and in the Node Registration Message.

A Node when it is initialized will have a list of Nodes it needs to connect to. The Node will iterate over the list and send out Label Registration message to each of these Nodes and will start a timer in which to receive a corresponding Label Registration message from the specific Node. If the timer expires, the Node will then send a Request Label Registration message to the Node and restart the timer. The Node will eventually stop the timer when it receives the Label Registration message from the specific node. If the Node receives a Label Registration message from a Node in its configuration list before it send out its own Label Registration message, then the Node will not enable a timer for that specific Node.

A Node will send out a Node Registration message after it has sent out a Label Registration message, and will start a timer in which to receive a corresponding Node Registration message with the particular Node id in it. If the timer expires, the Node will send a Request Node Registration message to the Node and restart the timer.

When a Node receives a Node Registration message it will check to see if there is a Node id in the message that it currently does not know about, and it will check all the message hashes (of the Label Ids and message sizes) of existing Node ids. The Node will also stop any timer associated with the remote Node id for Node Registration messages. If there was at least one new Node or one updated Message Hash, the Node will forward the Node Registration message to all Nodes except the remote Node that sent the Node Registration message. The Node will iterate through each new Node added or each Node's updated Message Hash and send out a Request Label Registration message to get that Node's set (updated) of Labeled Messages. For each of these requests, it will start a timer to resend the request

if it does not receive a Label Registration message before the timer expires [need to discuss if this should be done for all new/updated Nodes or only for Nodes in the original configuration list].

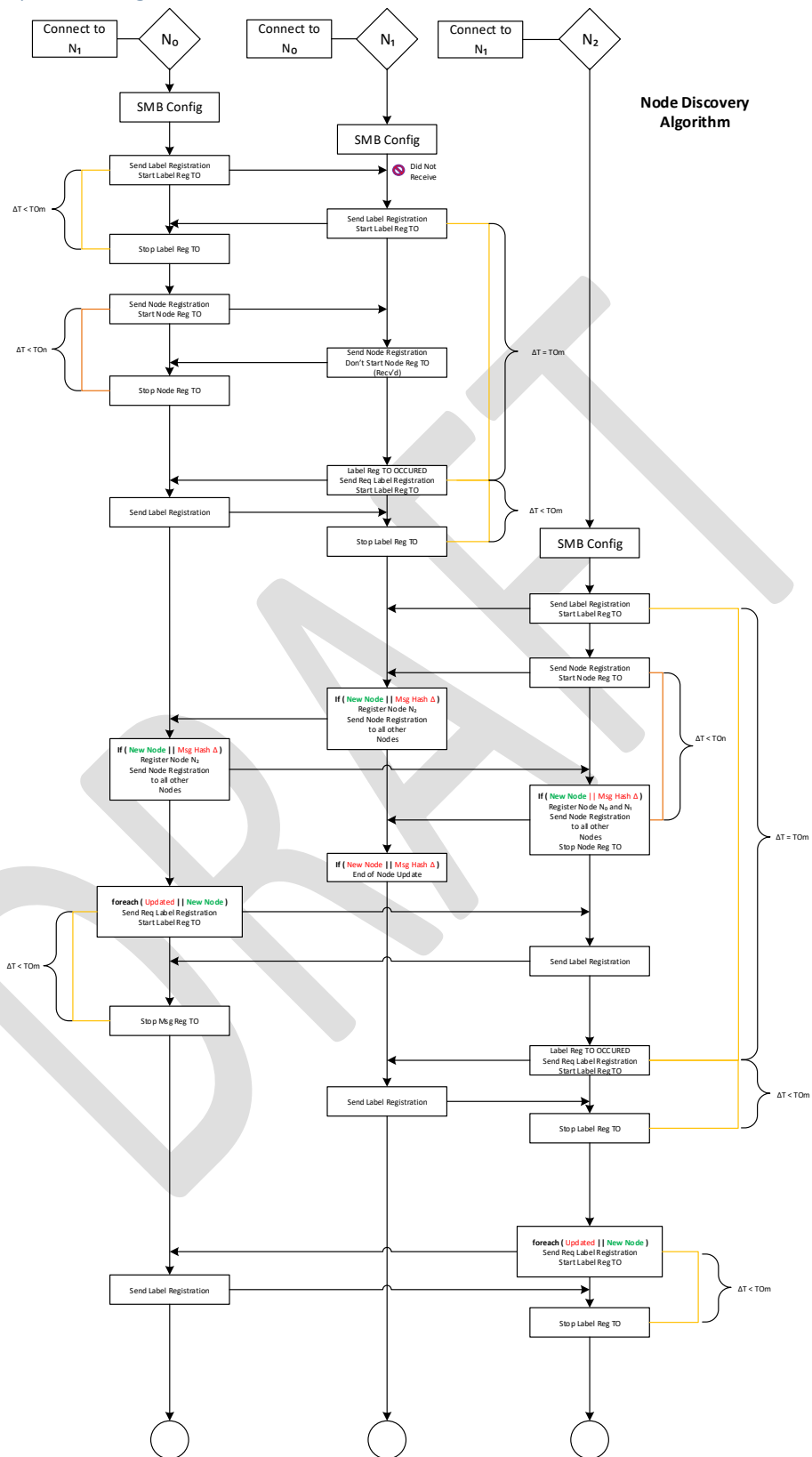
When a Node Receives a Request Node Registration message it will send a Node Registration message to the remote Node id in the Request Node Registration message [need to discuss if we want to support Nodes that will ignore this request if the Node id is not in the configuration list].

When a Node receives a Label Registration message it will stop any timers associated with the remote Node id for Label Registration messages.

When a Node Receives a Request Label Registration message it will send a Label Registration message to the remote Node id in the Request Label Registration message [need to discuss if we want to support Nodes that will ignore this request if the Node id is not in the configuration list].

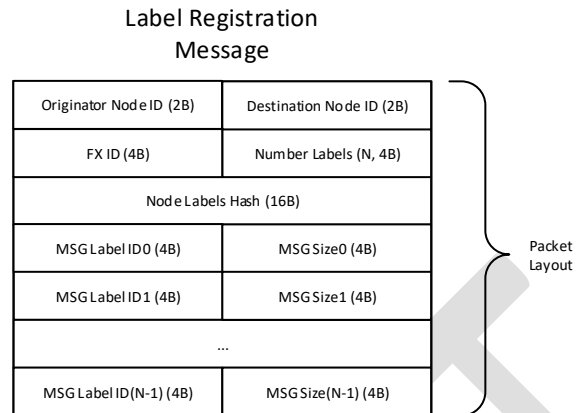
DRAFT

Node Discovery Flow Diagram

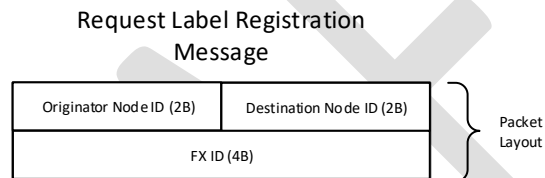


Node Discovery Message Data Structures

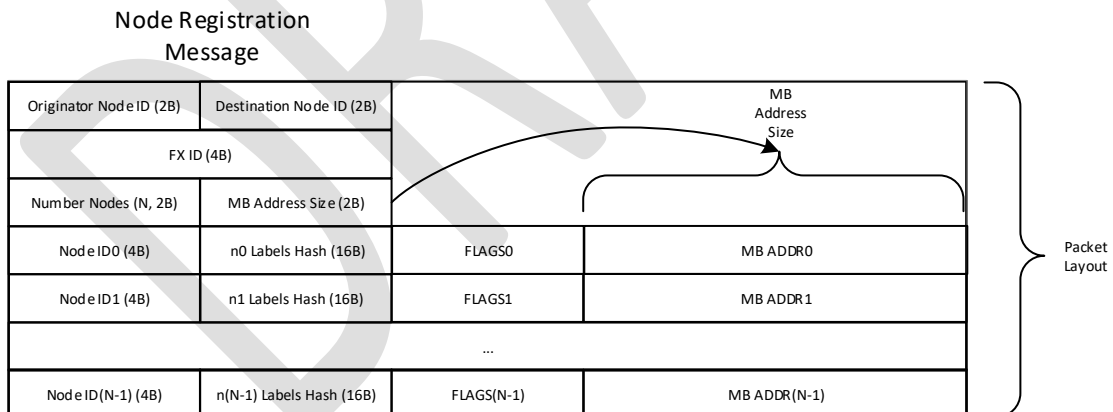
Label Registration Message



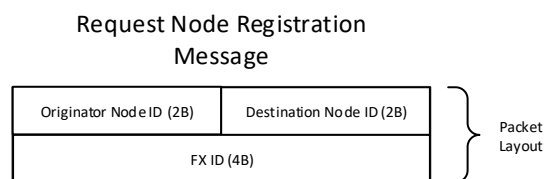
Request Label Registration Message



Node Registration Message



Request Node Registration Message



COE Initialization

[TODO]

OE Message

[TODO]

OE Endpoint

Object Interface

Allocation Methods

- **OE_Endpoint_Create_Dynamic**

This routine creates an Endpoint object for use by an application thread.

The input Name parameter specifies the name attribute of the object and is used to locate the resultant object within the system via the *oe_name_server*.

Following creation of the object, it is initialized with the specified input attributes. The size parameters are used to size the memory resources needed to support the Endpoint's send and receive operations. Endpoints are capable of sending and receiving, so an application thread can function using a single Endpoint to serve its transmitting and receiving needs. In a deviation from WSTAWG, Endpoints are also available with non-queued behavior. Pass in the value 0 for the *Max_Receive_Buffer_Messages* parameter to indicate that non-queued behavior is desired. When sizing the maximum receive message size, the maximum message size must encompass any overhead for cryptographic functions. Encryption of the message may or may not require extra buffer space above the plain text message size, depending on the algorithm used. Signing always requires extra buffer space; the amount depends on the algorithm used. It is the responsibility of the application to determine which cryptographic algorithms will be used on incoming messages, and to properly size the receive buffer to contain a message with either encryption or a signature.

- **OE_Endpoint_Create_Dynamic_With_Domain**

This routine creates an Endpoint object for use by an application thread.

The input Name parameter specifies the name attribute of the object and is used to locate the resultant object within the system via the *oe_name_server*.

Following creation of the object, it is initialized with the specified input attributes. The size parameters are used to size the memory resources needed to support the Endpoint's send and receive operations. Endpoints are capable of sending and receiving, so an application thread can function using a single Endpoint to serve its transmitting and receiving needs. In a deviation from WSTAWG, Endpoints are also available with non-queued behavior. Pass in the value 0 for the *Max_Receive_Buffer_Messages* parameter to indicate that non-queued behavior is desired.

- **OE_Endpoint_Abort**

This routine aborts any pending activity on the given Endpoint object. This routine can be used to terminate wait and receive calls pending indefinitely. This is useful prior to deleting endpoints to allow a thread to unregister labels and to perform cleanup activities.

- **OE_Endpoint_Delete**

This routine is called to delete an endpoint and all resources associated to it.

- **OE_Endpoint_Release**

This routine is called to release (reactivate) any threads pending on the endpoint. This routine is available for use before deleting a thread. Usually, it is a good idea for the thread "owning" (or using) the endpoint to be the one to delete it. But this can be difficult if the thread is pending on an incoming message when the application needs to delete the thread and its endpoint. One common approach will be to use an event flag to signal both incoming messages as well as a terminate request. This routine can be used when the thread simply does a loop around an endpoint receive. Calling this routine will cause any thread(s) pending on the endpoint to be reactivated with a *port_ABORT* error being returned from *OE_Endpoint_Receive* or *OE_Endpoint_Wait* operation. This can be taken by the thread to be a termination request.

Registration Methods

- **OE_Endpoint_Register**

This routine registers an Endpoint for receipt of labeled messages in accordance with the specified input values. The input Label indicates the label for which the Endpoint should be registered. When this routine is invoked, the Endpoint shall accept any message sent via a labeled send operation whose label corresponds to the input Label value. When this routine is called in succession, the specified labels are to be processed in a cumulative manner, allowing for the registration of multiple label values by the requesting application. In the event that the implementation cannot support the operation due to a resource limitation, the operation shall fail and an error is generated.

The *Format_Packet_Function* parameter, which is a deviation from the WSTAWG specification, is a function that provides a pointer to a data formatting structure that describes the data associated with the Label. This structure is important because it is used to support Endian Conversion for marshalling and unmarshalling data.

- **OE_Endpoint_Register_With_Domain**

This routine registers an Endpoint for receipt of labeled messages in accordance with the specified input values. The input Label indicates the label for which the Endpoint should be registered. When this routine is invoked, the Endpoint shall accept any message sent via a labeled send operation whose label corresponds to the input Label value. When this routine is called in succession, the specified labels are to be processed in a cumulative manner, allowing for the registration of multiple label values by the requesting application. In the event that the implementation cannot support the operation due to a resource limitation, the operation shall fail and an error is generated.

The *Format_Packet_Function* parameter, which is a deviation from the WSTAWG specification, is a function that provides a pointer to a data formatting structure that describes the data associated with the Label. This structure is important because it is used to support Endian Conversion for marshalling and unmarshalling data.

- **OE_Endpoint_Register_Ex**

This routine is an extension of the *OE_Endpoint_Register* routine. The difference here is the extended *Format_Packet_Function* that is accepted. This *Format_Packet_Function* not only takes the Format Packet as a parameter, but it also provides the label to the developer to allow more control in returning an appropriate Format Packet. This can open possibilities for dynamic message creation that were complicated by the simplified *Format_Packet_Function*. This function assumes a label with domain of 0.

- **OE_Endpoint_Register_With_Domain_Ex**

This routine is an extension of the *OE_Endpoint_Register* routine. The difference here is the extended *Format_Packet_Function* that is accepted. This *Format_Packet_Function* not only takes the Format Packet as a parameter, but it also provides the label to the developer to allow more control in returning an appropriate Format Packet. This can open possibilities for dynamic message creation that were complicated by the simplified *Format_Packet_Function*.

- **OE_Endpoint_Register_Ex2**

This routine is an extension of the *OE_Endpoint_Register* routine. The difference here is the extended *Format_Packet_Function* that is accepted. This *Format_Packet_Function* takes the label as a parameter and returns the Format Packet as a return value. This form is more conducive for use with different languages. This function assumes a label with domain of 0.

- **OE_Endpoint_Register_With_Domain_Ex2**

This routine is an extension of the *OE_Endpoint_Register* routine. The difference here is the extended *Format_Packet_Function* that is accepted. This *Format_Packet_Function* takes the label as a parameter and returns the Format Packet as a return value. This form is more conducive for use with different languages.

- **OE_Endpoint_Deregister**

This routine deregisters a message label from the router database. Effectively it removes the endpoint reference out of the router database for the given domain:label_id, where the domain is provided by the endpoint.

- **OE_Endpoint_Deregister_Ex**

This routine deregisters a message label from the router database. Effectively it removes the endpoint reference out of the router database for the given domain:label_id, where the domain is explicitly passed in.

General Use Methods

- **OE_Endpoint_Send_Labeled**

This routine sends a Message to any destination Endpoints that have registered for the label value contained within the input Message in accordance with the specified input values.

The input *Handling_Policy* indicates the manner in which the implementation is to utilize the Message object with respect to the message transmission. In the event that the message to be sent exceeds the maximum transmit message size the operation shall fail and an error shall be generated.

If a condition is encountered where the size of *The_Message* is too large to fit within the *Max_Message_Byte_Size* of one of the receiving Endpoints on the local processor that have registered to receive this message, the following text message is output:

"Message label xxx and size xxx will not fit into the given endpoint size of xxx."

- **OE_Endpoint_Send_Labeled_Exclusive**

This routine sends a Message to any destination Endpoints that have registered for the label value contained within the input Message in accordance with the specified input values. This is similar to the *OE_Endpoint_Send_Labeled* routine except that the sending endpoint will always be specifically excluded from receiving the message if it is registered to receive the message that is being sent.

The input *Handling_Policy* indicates the manner in which the implementation is to utilize the Message object with respect to the message transmission. In the event that the message to be sent exceeds the maximum transmit message size the operation shall fail and an error shall be generated.

If a condition is encountered where the size of *The_Message* is too large to fit within the *Max_Message_Byte_Size* of one of the receiving Endpoints on the local processor that have registered to receive this message, the following text message is output:

"Message label xxx and size xxx will not fit into the given endpoint size of xxx."

- **OE_Endpoint_Peek**

This routine examines the next message contained within the Endpoint's receive queue. The outputs *Message_Label* and *Message_Size* indicate the label and size in bytes associated with the Message at the front of the receive buffer is examined in order to determine the associated output attributes. The execution of this routine does not remove the Message or modify the content of the Endpoint receive buffer in any way. In the event that no Message is available within the receive buffer, the operation fails and an error is generated.

- **OE_Endpoint_Wait**

This routine will wait for the endpoint to contain a message. This provides an alternative to using an associated event flag. The caller will be suspended if the endpoint's queue is empty until a message is received, or a timeout occurs. The execution of this routine does not remove the Message or modify the content of the Endpoint receive buffer in any way.

- **OE_Endpoint_Receive**

This routine receives a Message from the Endpoint in accordance with the specified input values. The input Message object specifies the message that is to contain the read results. The Message object is specified as an in/out parameter in order to accommodate reference reads. The input *Handling_Policy* indicates the manner in which the implementation is to utilize the Message object with respect to the message transmission. The input Timeout specifies the amount of time to wait in order for the operation to complete (either successfully or unsuccessfully). Timeout values greater than zero shall cause the application to block until the operation has completed.

In the event that the specified timeout expires prior to the successful completion of the operation, the operation shall be implicitly cancelled and a *Failed_Timeout* status shall be returned. The returned *The_Source* Endpoint object is not implemented.

In the event that the input Message object is too small to accommodate the next message in the receive buffer or no Message is available, the operation shall fail and an error is generated. In addition, if the input Endpoint object is not valid, the operation fails and an error will be generated. If the endpoint becomes empty as a result of the execution of the operation, the *Endpoint_Empty* trigger will be asserted, signaling any associated events.

- **OE_Endpoint_Empty**

This routine is called to delete all queued messages that exist on the endpoint queue.

Special Use Methods

- **OE_Endpoint_Associate**

This routine associates the Endpoint's Data_Received or Resources_Exhausted Trigger with the specified Event. This enables applications to perform event-driven processing when a message arrives at the Endpoint, or the Endpoint has run out of receive or transmit buffers to allocate.

- **OE_Endpoint_Get_Statistics**

This routine returns the statistics structure of the endpoint. The statistics structure is a structure maintained by the endpoint that contains statistics about the operation of the endpoint. These values are maintained by the endpoint, but are not critical to the endpoint's operation, so the application is free to read these values and to change them (ie, reset them to 0) as desired.

- **OE_Endpoint_Auto_Decrypt**

This routine will enable and disable the auto decrypt function. When the endpoint is first created, the auto decrypt function is enabled. This means that when a message is read via the *OE_Endpoint_Receive* function, the Message object is automatically verified if it is signed, and decrypted if it is encrypted. If the auto decrypt function is disabled, the receive Message object will be returned to the application without verification or decryption. It will be up to the application to make the appropriate calls to interact with the message data.

- **OE_Endpoint_Set_Callback**

This routine will enable or disable a callback routine that will be called when a message has been received and is being enqueued on the queue of the endpoint. This callback routine can override the queueing function or provide processing on the incoming data. There can be only one callback routine defined for any endpoint.

OE Thread

[TODO]

OE Semaphore

[TODO]

OE Mutex

[TODO]

OE Event

[TODO]

OE EventFlag

[TODO]

OE Clock Handler

[TODO]

OE Encryptor

[TODO]

OE Synchronization

[TODO]

Nameserver

[TODO]

Data Interchange

[TODO]

Abstract Data Types

[TODO]

C++ Shell

[TODO]