Networking and Distributed Systems
Computing Science Laboratory
copyright 2015

Metis CCNx 1.0 Forwarder

Marc Mosko*[1]

Abstract
Metis is the CCNx 1.0 forwarder that implements the CCNx 1.0 Semantics and Messages draft standards. This document describes how to use Metis and the internal software architecture.

Keywords
Content Centric Networks, Forwarder, Metis

[1] *Computing Science Laboratory, PARC*
***Corresponding author**: marc.mosko@parc.com

Contents

1. Introduction

Metis is a CCNx 1.0 forwarder written in C using the PARCLibrary package. A forwarder is responsible for receiving wire format packets from one place and forwarding them to another. When a forwarder runs on an end host, it typically forwards packets between applications, themselves, and between applications and the network. When a forwarder runs as an intermediate system, it typically forwarders between peers, though it may have a small number of specialized applications, such as routing protocols, running on the device.

This document describes the Metis architecture and principle data structures and algorithms. Section 2 provides a general architecture overview. Section 3 describes how to use Metis as a command line program `metis_daemon` and how to configure Metis with the command line program `metis_control`. It also covers the syntax of a configuration file used by `metis_daemon`. Section 4 describes the inner workings of Metis through flow charts and key C structures.

2. Architecture

Metis is designed around the concept of a Connection as the atom of adjacency. A Connection can be a TCP or UDP connection ({src_ip, src_port, dst_ip, dst_port}), an Ethernet adjacency ({smac, dmac, etherType}), a UNIX

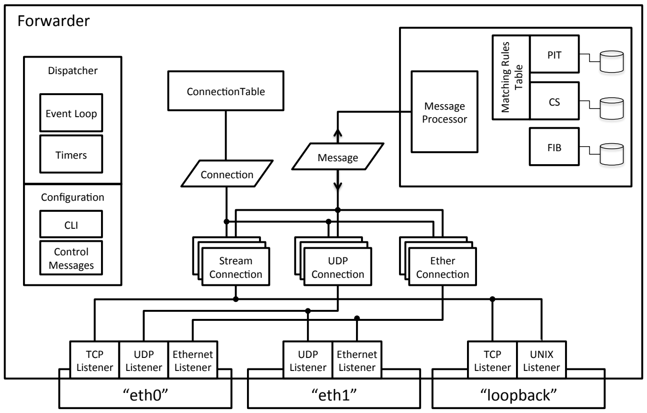

Figure 1. Metis Architecture

domain socket connection, or an IP multicast group ({src_ip, src_port, group_ip, group_port}). The ConnectionTable tracks all these adjacencies and provides a ConnectionID (CID). The CID is used in other tables, such as the forwarding table (FIB) to denote next (egress) hops and in the pending Interest table (PIT) to denote previous (ingress) hop.

The Metis forwarder is comprised of several major modules, the two principle ones being the IO module and the message processor module. The IO module consists of a set of Listeners that implement the MetisListenerOps interface and a set of protocol Connections (e.g. StreamConnection or UdpConnection) that implement the MetisIoOps interface. Because each Connection is protocol specific, it can implement the correct *send()* function for the protocol and keep the protocol specific state it needs.

When Metis receives a packet, it converts the packet into a *MetisMessage*, which is an extent (offset, length) map of important TLV fields to their location inside a packet. The *MetisMessage* also carries information about the ingress Connection.

The Message Processor receives all *MetisMessage* and directs Interests and Content Objects to the appropriate processing path. The Message Processor encapsulates the Pending Interest Table (PIT), Forwarding Information Base (FIB), and Content Store (CS). If a Content Object (in the form of a *MetisMessage*) is returned to the ingress port, it is sent by calling the ingress connection's *send()* function. If the Interest is to be forwarded, it is reference-count replicated to each next hop's Connection *send()* function.

A Content Store must comply with the *MetisContentStoreInterface* (see Section 4.8.3. Metis provides a memory-backed transient Content Store implementation, *MetisLRUContentStore*, that uses an LRU replacement strategy.

Metis includes a *MetisDispatcher* that is responsible for timers and socket polling. It is a single threaded, non-preemptive dispatcher. Timers are serviced at the nearest time no earlier than their expiration via a callback. When a socket is readable or writeable, the dispatcher calls a corresponding handler. The current implementation uses *PARCDispatcher*, which is based on Libevent.

A configuration module has a command parser used by both the configuration file and to parse received configuration messages from the network. Currently, the configuration messages over the network use a JSON encoding.

2.1 Future Work

While there is much yet left to do in Metis, these are some of the main future work items.

2.1.1 Threading

Metis will be threaded in a conventional "reader - parser - lookup - writer" model. This will generally correspond as reader and writer to the IO module, parser to *MetisMessage* construction, lookup to the message processor module, and writer to the IO module.

2.1.2 Interface Generalization

Not all interfaces are generalized to allow pluggable implementations. In particular, the FIB, CS, and Strategy sections still need work to bring up to a clean facade pattern.

2.1.3 Dispatcher and network I/O

The current reliance on Libevent and the PARCDispatcher will be replaced with a much leaner and properly generalized facade. This will allow one to substitute any suitable back-end for network IO.

2.1.4 Ethernet

There are plans on moving Ethernet to kernel bypass networking on Linux when supported by a backend like netmap/VALE or Intel DPDK.

The current Linux Ethernet, based on raw socket I/O, will be updated to use shared kernel memory even without netmap or DPDK. This is a small change to introduce the shared memory kernel buffers.

The current Darwin Berkeley Packet Filter will be updated to use PF_NDRV, which should make it very similar to the current (non-shared memory) Linux raw socket module.

2.1.5 Configuration Messages

The current use of a proprietary fixed header PacketType and embedded JSON string for a configuration message will be replaced with a CCNx 1.0 Control Message, which is a signed Content Object. We will be adding a certificate trust mechanism to Metis along with ways to restrict which connections can receive control messages.

3. Usage

This section describes how to run and configure Metis. The content of this section is the same as the man pages for metis_daemon, metis_control, and metis.cfg.

3.1 Metis Daemon

metis_daemon — Metis is the CCNx 1.0 forwarder, which runs on each end system and as a software forwarder on intermediate systems.

Synopsis

`metis_daemon` [--port *`port`*] [--daemon] [--capacity *`contentStoreSize`*] [--log *`facility=level`*...] [--log-file *`logfile`*] [--config *`configfile`*]

DESCRIPTION

metis_daemon is the CCNx 1.0 forwarder, which runs on each end system and as a software forwarder on intermediate systems. metis_daemon is the program to launch Metis, either as a console program or a background daemon (detatched from console). Once running, use the program *metis_control* to configure Metis.

Metis is structured as a set of Listeners, each of which handles a specific method of listening for packets. For example, a TCP listener will accept connections on a specific TCP port on a specific local IP address. An Ethernet listener will accept frames of a specific EtherType on a specific Interface.

When Metis accepts a connection, it will create a Connection entry in the ConnectionTable to represent that peer. For Ethernet, a Connection is the tuple {dmac, smac, ethertype}. For TCP and UDP, it is the tuple {source IP, source port, destination IP, destination port}. The connid (connection ID) becomes the reverse route index in the Pending Interest Table.

OPTIONS

--config *configfile* Reads configuration parameters from *configfile*. The `--port` option has no effect in this mode and Metis will not listen to any ports. This means that *metis_control* will not be able to connect to Metis to configure it further unless one includes at least a listener for TCP localhost or a unix domain socket.

--capacity *contentStoreSize* Sets the capacity of the Content Store to *contentStoreSize* content objects. Metis uses a least-recently-used eviction policy. A size of 0 will disable the Content Store.

The Content Store sits on the fast path of the forwarder, so there is a cost associated with adding and removing items to the Content Store tables.

--daemon Runs Metis in daemon mode, detaching from the console. It must be run with the `--log-file` option.

--log *facility=level* Sets the log level of the given *facility* to the given *level*. The `--log` option may be repeated several times setting the log level of different facilities. If the same facility is listed twice, only the last occurance takes effect. The default log level is Error for all facilities.

Facilities:

- all: All facilities.
- config: Configuration activies.
- core: Core forwarder, such as startup and shutdown.
- io: Listeners, connections, and all I/O related activities.
- message: CCNx messages, such as parsing.
- processor: Forwarding processor, such as CS, FIB, and PIT activities.

The log levels are: debug, info, notice, warning, error, critical, alert, off.

--log-file Specifies the *logfile* to write all log messages. This parameter is required with `--daemon` mode.

--port *port* The UDP and TCP port to listen on. If no *configfile* is specified, Metis will listen on this port on all interfaces including localhost.

If this parameter is not given, Metis uses the default port 9695.

USAGE

metis_daemon --config metis.cfg --log all=info --log config=debug --log-file metis.log

SEE ALSO

See *metis_control(1)* for a description of how to configure **metis_daemon**.

For a list of all configuration lines that may be used with *metis_control* and by `--config` configuration file, see *metis.cfg(5)*.

CAVEATS

- A given interface may only have one Ethernet listener on one EtherType.
- If there are multiple longest matching prefix entries that match an Interest, it will be forwarded to all those routes (i.e. multicast).
- Ethernet fragmentation will only use the interface MTU and there is no MTU discovery. If Metis is used in a bridged environment, this may lead to errors if the MTU changes on different segments, such as a 10G link at 9000 bytes and a 100 Mbps link at 1500 bytes.

3.2 Metis Control

metis_control — Metis is the CCNx 1.0 forwarder, which runs on each end system and as a software forwarder on intermediate systems. metis_control is the program to configure the forwarder, metis_daemon.

Synopsis

`metis_control` [--keystore *`keystore`*] [--password *`password`*] [*`commandline`*]

DESCRIPTION

metis_control is the program used to configure a running forwarder *metis_daemon*. It will connect to the forwarder over a local listener (e.g. TCP to localhost or a unix domain socket). If a *commandline* option is specified, **metis_control** will send that one command to Metis and then exit. If no *commandline* is specified, *metis_command* will enter interacitve mode where the user can issue multiple commands.

metis_control requires a signing keystore for communicating over the network. The *keystore* file is a standard PKCS12 keystore, and may be created using *parc_publickey(1)*. If no *keystore* is specified, **metis_control** will look in the standard path ~/.ccnx/.ccnx_keystore.p12. The keystore password is specified in *password*. If not specified, no password is used. If the keystore does not open, the user will be prompted for a password.

See *metis.cfg(5)* for a specification of the available *commandline*.

The environment variable METIS_PORT may be used to specify what TCP port to use to connect to the local Metis. The environment variable METIS_LOCALPATH may be used to specify the UNIX domain socket to connect to the local Metis and takes priority over METIS_PORT.

OPTIONS

--keystore ***keystore***
: **metis_control** requires a signing keystore for communicating over the network. The *keystore* file is a standard PKCS12 keystore, and may be created using *parc_publickey(1)*. If no *keystore* is specified, **metis_control** will look in the standard path ~/.ccnx/.ccnx_keystore.p12.

--password ***password***
: The keystore password is specified in *password*. If not specified, no password is used. If the keystore does not open, the user will be prompted for a password.

commandline The remainder of the arguments are the commandline to send to Metis. See USAGE.

USAGE

metis_control --keystore keystore.p12

metis_control --keystore keystore.p12 list interfaces

SEE ALSO

See *parc_publickey(1)* for a utility to create a PKCS keystore.

For a list of all configuration lines that may be used with **metis_control** and by `--config` configuration file, see *metis.cfg(5)*.

The default keystore is ~/.ccnx/.ccnx_keystore.p12.

3.3 Metis Configuration File

metis.cfg is an example of a configuation file usable with metis_daemon(1), though there is nothing special about the actual filename. Each line of the configuration file is also usable with metis_control(1). This document specifies all available command lines used to configure and query Metis. All commands have a 'help', so typing 'help command' will display on-line help. In a configuration file, lines beginning with '#' are comments.

ADD COMMANDS

add connection ether *symbolic dmac interface* Adds an Ethernet connection on *interface* to the given destination MAC address. The *symbolic* name is a symbolic name for the connection, which may be used in later commands, such as *add route*. There must be an Ethernet Listener on the specified interface (see *add listener*), and the connection will use the same EtherType as the Listener. The *dmac* destination MAC address is in hexidecimal with optional "-" or ":" separators.

A connection is a target for a later route assignment or for use as an ingress identifier in the PIT. When using a broadcast or group address for a connection, an Interest routed over that connection will be broadcast. Many receivers may respond. When Metis receives a broadcast Interest it uses the unicast source MAC for the reverse route -- it will automatically create a new connection for the source node and put that in the PIT entry, so a Content Object answering the broadcast Interest will only be unicast to the previous hop.

add connection ether conn7 e8-06-88-cd-28-de em3

add connection ether bcast0 FFFFFFFFFFFF eth0

add connection (tcp — udp) *symbolic remote_ip remote_port local_ip local_port*
Opens a connection to the specific *remote_ip* (which may be a hostname, though you do not have control over IPv4 or IPv6 in this case) on *remote_port*. The local endpoint is given by *local_ip local_port*. While the *local_ip local_port* are technically optional parameters, the system's choice of local address may not be what one expects or may be a different protocols (4 or 6). The default port is 9695.

A TCP connection will go through a TCP connection establishment and will not register as UP until the remote side accepts. If one side goes down, the TCP connection will not auto-restart if it becomes availble again.

A UDP connection will start in the UP state and will not go DOWN unless there is a serious network error.

Opens a connection to 1.1.1.1 on port 1200 from the local address 2.2.2.2 port 1300
add connection tcp conn0 1.1.1.1 1200 2.2.2.2 1300

opens connection to IPv6 address on port 1300
add connection udp barney2 fe80::aa20:66ff:fe00:314a 1300

add listener (tcp—udp) ***symbolic ip_address port*****, add listener ether** ***symbolic interfaceName ethertype*****, add listener local** ***symbolic path***

Adds a protocol listener to accept packets of a given protocol (TCP or UDP or Ethernet). The *symbolic* name represents the listener and will be used in future commands such as access list restrictions. If using a configuration file on *metis_daemon*, you must include a listener on localhost for local applications to use.

The *ip_address* is the IPv4 or IPv6 local address to bind to. The *port* is the TCP or UDP port to bind to.

The *interfaceName* is the interface to open a raw socket on (e.g. "eth0"). The *ethertype* is the EtherType to use, represented as a 0x hex number (e.g. 0x0801) or an integer (e.g. 2049).

The *path* parameter specifies the file path to a unix domain socket. Metis will create this file and remove it when it exits.

Listens to 192.168.1.7 on tcp port 9695 with a symbolic name 'homenet'
add listener tcp homenet 192.168.1.7 9695

Listens to IPv6 localhost on udp port 9695
add listener udp localhost6 ::1 9695

Listens to interface 'en0' on ethertype 0x0801
add listener ether nic0 en0 0x0801

add route ***symbolic prefix prefix***

Adds a static route to a given *prefix* to the FIB for longest match.

Currently, the *symbolic* and *cost* are not used.

LIST COMMANDS

list connections

Enumerates the current connections to Metis. These include all TCP, UDP, Unix Domain, and Ethernet peers. Each connection has an connection ID (connid) and a state (UP or DOWN) followed by the local (to metis) and remote addresses.

list interfaces

Enumerates the system interfaces available to Metis. Each interface has an Interface ID, a 'name' (e.g. 'eth0'), an MTU as reported by the system, and one or more addresses.

list routes Enumerates the routes installed in the FIB. The *iface* is the out-bound connection. The *protocol* is the the routing protocol that injected the route. 'STATIC' means it was manually entered via *metis_control*. *route* is the route type. 'LONGEST' means longest matching prefix and 'EXACT' means exact match. Only 'LONGEST' is supported. *cost* is the cost of the route. It is not used. *next* is the nexthop on a multiple access interface. it is not used because the current implementation uses one connection (iface) per neighbor. *prefix* is the CCNx name prefix for the route.

Examples

```
> list connections
23   UP inet4://127.0.0.1:9695 inet4↩
     ://127.0.0.1:64260 TCP

> list interfaces
int        name lm      MTU
24          lo0 lm    16384 inet6://[::1\%0]:0
inet4://127.0.0.1:0
inet6://[fe80::1\%1]:0
25          en0  m     1500 link://3c-15-c2-e7-c5-ca
inet6://[fe80::3e15:c2ff:fee7:c5ca\%4]:0
inet4://13.1.110.60:0
inet6://[2620::2e80:a015:3e15:c2ff:fee7:c5ca\%0]:0
inet6://[2620::2e80:a015:a4b2:7e10:61d1:8d97\%0]:0
26          en1  m     1500 link://72-00-04-43-4e-50
inet4://192.168.1.1:0
27          en2  m     1500 link://72-00-04-43-4e-51
28      bridge0  m     1500 link://3e-15-c2-7e-96-00
29         p2p0  m     2304 link://0e-15-c2-e7-c5-ca

> list routes
iface  protocol   route     cost                  ↩
     next prefix
23    STATIC LONGEST        1 ---.---.---.---/.... ↩
     lci:/foo/bar
Done
```

REMOVE COMMANDS

remove connection Not implemented.

remove route Not implemented.

MISC COMMANDS

quit In interactive mode of *metis_control*, it cause the program to exit.

set debug Turns on the debugging flag in *metis_control* to display information about its connection to Metis.

unset debug Turns off the debugging flag in *metis_control* to display information about its connection to Metis.

USAGE

Example Linux metis.cfg configuration file

```
#local listeners for applications
add listener tcp local0 127.0.0.1 9695
add listener udp local1 127.0.0.1 9695
add listener local unix0 /tmp/metis.sock

# add ethernet listener and connection
add listener ether nic0 eth0 0x0801
add connection ether conn0 ff:ff:ff:ff:ff:ff eth0
add route conn0 lci:/ 1

# add UDP tunnel to remote system
add connection udp conn1 ccnx.example.com 9695
add route conn1 lci:/eample.com 1
```

Example one-shot metis_control commands

```
metis_control list routes
metis_control add listener local ↩
   unix0 /tmp/metis.sock
```

4. Internal Structure

... put stuff here ...

4.1 Connection State Machine

A Connection (see below, Section 4.6) follow this state machine:

```
initial   -> CREATE
CREATE    -> (UP | DOWN)
UP        -> (DOWN | DESTROYED)
DOWN      -> (UP | CLOSED | DESTROYED)
CLOSED    -> DESTROYED
DESTROYED -> terminal
```

These states should be signaled via the *MetisMessenger* (see Section 4.2 to any component that wishes to subscribe to connection event messages. It is the responsibility of the Listener (Section 4.4) and IO Connection (Section 4.5) to generate these signals.

4.2 Messanger

The *MetisMessenger* interface inside Metis is to send internal signals of events. A module can subscribe to receive messages via *metisMessenger_Register()*. When any component signals a message via *metisMessenger_Send()*, all *MetisMessengerRecipient* callbacks will receive the message *in a later dispatcher scheduling time*.

The essential element of the Messenger is the *in a later dispatcher scheduling time* condition. This avoids preemption and circular callback firing.

Figure 2 shows the interesting API functions of the messenger. Currently, a *MetisMissive* can only signal the state machine for a Connection ID.

4.3 Configuration

FINISH

```
void metisMessenger_Send(MetisMessenger *messenger,↩
      MetisMissive *missive);
void metisMessenger_Register(MetisMessenger *↩
     messenger, const MetisMessengerRecipient *↩
     recipient);
void metisMessenger_Unregister(MetisMessenger *↩
     messenger, const MetisMessengerRecipient *↩
     recipient);
```

Figure 2. MetisMessenger API

```
struct metis_listener_ops {
    void *context;
    void (*destroy)(MetisListenerOps **↩
         listenerOpsPtr);
    unsigned (*getInterfaceIndex)(const ↩
         MetisListenerOps *ops);
    const CPIAddress * (*getListenAddress)(const ↩
         MetisListenerOps *ops);
    MetisEncapType (*getEncapType)(const ↩
         MetisListenerOps *ops);
    int (*getSocket)(const MetisListenerOps *ops);
};
```

Figure 3. MetiisListenerOps

4.4 Listeners

When Metis starts up, it will either create a set of default listeners (TCP, UDP) on a given port or only create those listeners specified in the configuration file. All listeners implement the *MetisListenerOps* interface (see Figure 3).

The job of a listener is to receive a packet from the network, associate it with a *MetisConnection*, create a *MetisMessage*, and send it to the *MetisMessageProcessor*. For stream listeners, the *accept()* happens in the listener, and from then on the per-client socket IO happens in *MetisStreamConnection*. For datagram listeners (UDP and Ethernet), the Listener has to do all the initial IO to at least match against a Connection. In the current code, the Listener does all the IO – matching to a Connection and creating the *MetisMessage*.

The *destroy()* function is called during cleanup to release the listener. The *getInterfaceIndex()* function returns which host interface the listener is bound to. The *getListenAddress* is the host address the listener is bound to. The *getEncapType()* function is used to display listener information and indicates the encapsulate (TCP, Ethernet, etc.) used by the listener. The *getSocket()* function is used by some protocol connections when they need to send a packet from the listeners socket address, such as UDP.

In the case of Ethernet, the Listener is split between a platform-specific module for the low-level IO module called *MetisGenericEther* (see Figure 4) and the high-level *MetisListener*. *MetisGenericEther* is the header that each platform-specific Ethernet module implements. It is not a structure-style facade, but a straight header as we expect only one platform-specific object file per platform.

```
MetisGenericEther *metisGenericEther_Create(↩
     MetisForwarder *metis, const char *deviceName,↩
      uint16_t etherType);
MetisGenericEther *metisGenericEther_Acquire(const ↩
     MetisGenericEther *ether);
void metisGenericEther_Release(MetisGenericEther **↩
     etherPtr);
int metisGenericEther_GetDescriptor(const ↩
     MetisGenericEther *ether);
bool metisGenericEther_ReadNextFrame(↩
     MetisGenericEther *ether, PARCEventBuffer *↩
     buffer);
bool metisGenericEther_SendFrame(MetisGenericEther ↩
     *ether, PARCEventBuffer *buffer);
PARCBuffer *metisGenericEther_GetMacAddress(const ↩
     MetisGenericEther *ether);
uint16_t metisGenericEther_GetEtherType(const ↩
     MetisGenericEther *ether);
unsigned metisGenericEther_GetMTU(const ↩
     MetisGenericEther *ether);
```

Figure 4. MetisGenericEther platform-specific interface

4.4.1 Stream Listeners (TCP, Unix)

Figure 5 shows the process of *TcpListener*, *UnixListener*, and *StreamConnection* when receiving a packet. Because it is a stream connection, we must do our own framing based on the Fixed Header. *StreamConnection* currently does not have any framing error recovery. *TcpListener* and *UnixListener* are invoked to accept a new connection, and go through the (for example) TCP Accept process. This creates a *StreamConnection* and associates it with the client socket, creates the *MetisIoOps* associated with TCP, and adds it to the Connection Table. Once the connection is ready to go, it also sends a Metis Messenger signal that the connection is in the UP state.

Inside *StreamConnection*, we need to maintain state about framing because bytes may arrive with arbitrary delineation not corresponding to CCNx 1.0 packets. If we do not know the PacketLength, then we have not read a Fixed Header yet. We buffer until we have read 8 bytes and can parse the Fixed Header. Once we know the PacketLength from a Fixed Header, then we read the socket up to PacketLength bytes or the end of the available bytes (non-blocking). Once we have read PacketLength bytes, we can create a *MetisMessage* from the buffer and pass it to the Message Processor.

4.4.2 UDP Listener

Figure 6 shows the process of *UdpListener* receiving a packet. Because UDP is datagram based, we do not need to manage framing as in the *StreamConnection*. However, as there is no dedicated client socket, the *UdpListener* must construct a key for the Connection Table from the source and destination socket addresses to lookup (or create) a corresponding Connection. Creating a connection is the same as previously described for TCP, except the *MetisIoOps* concrete class is *MetisUdpConnection*. The UDP receive process currently does not have a buffer pool, so it peeks at the FixedHeader bytes to determine how bit a buffer to allocate and then reads the packet in to that buffer. This process is inefficient because it requires two system calls per read. Once the packet is read, we proceed as above creating a *MetisMessage* and passing it to the Message Processor.

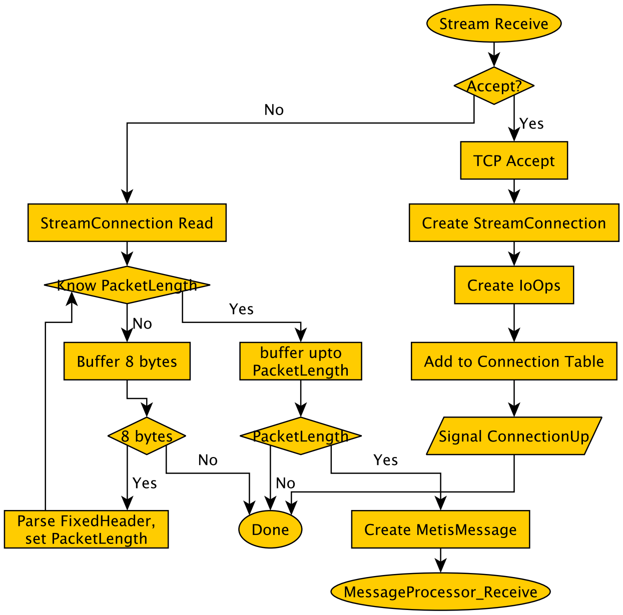

Figure 5. Stream Receive

4.4.3 Ethernet Listener

Figure 7 shows the Ethernet receive process down to the *GenericEther* abstraction level, which does not include the low-level platform specific parts. These will differ between linux and Mac and other paltforms. The platform Ethernet implementation may need to trim the CRC from the packet, as some platforms strip it and some do not.

The Ethernet process is similar to the UDP process in that there is no client socket, so the Ethernet listener needs to resolve the Connection by doing its own query to the Connection Table.

The first steps are to ensure the received Ethernet frame matches our EtherType, an acceptable destination MAC address (dmac) and is not our source MAC address (smac). Acceptable dmac addresses include the interface hardware address, the broadcast address, and the CCNx Ethernet group address. If the packet passes these tests, we lookup the address tuple {smac, dmac, etherType} in the Connection Table and create a new *MetisEtherConnection* if needed. Creating a new *MetisIoOps* proceeds as above.

Once we are past the Ethernet header, we can read the Fixed Header, allocate a buffer for the *MetisMessage* and read the packet in to that buffer. The exact memory mechanics that happen here can vary depending on the platform Ethernet implementation. Once we have a *MetisMessage* , it is passed to the Message Processor.

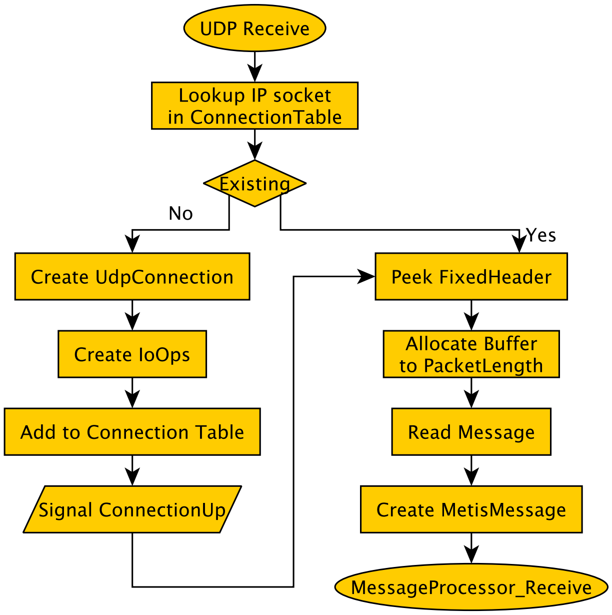

Figure 6. UDP Receive

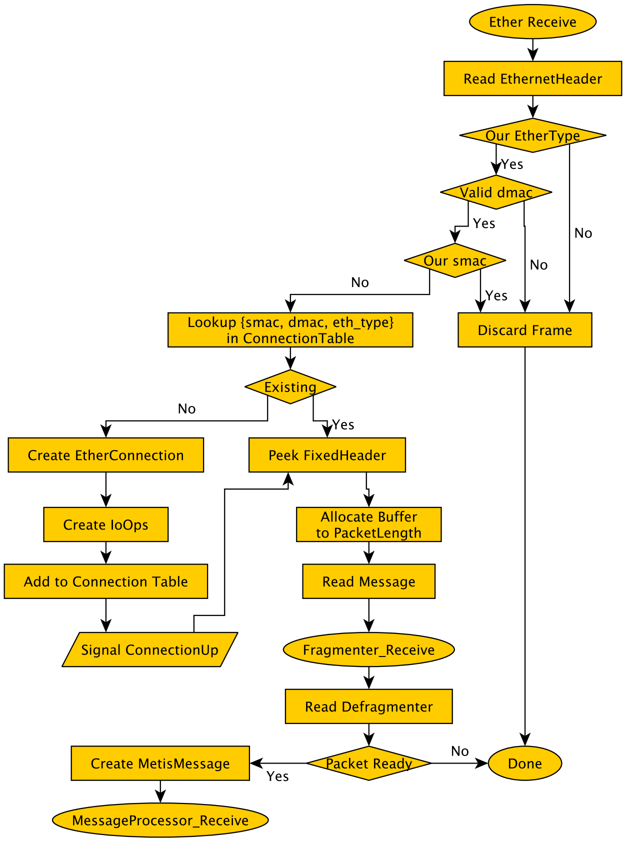

Figure 7. Ethernet Receive

4.5 IO Connections

An IO Connection is code that implements the `MetisIoOps` interface, shown in Figure 8. Each `MetisIoOps` represents a connection, so it has a *getAddressPair()* function. The addresses are of type *CPIAddress* which holds IP and UNIX and Ethernet addresses.

The *send()* function is used by the Message Process to send a Content Object back along an Interest reverse path and to forward an Interest to next hops in the FIB.

The *isUp()* function indicates if the connection is able to send packets. Sometimes a connection is valid, but is not up. For example a TCP connection will be valid but not Up during the time it is connecting to a remote peer.

The *isLocal()* function indicates if the remote address is local to the host. Ethernet is never local. IP addresses to the IPv4 and IPv6 loopback address are always local. UNIX domain sockets are always local.

The *getConnectionId()* function returns an integer representing the connection. It may be used as a foreign key in other tables.

The *destroy()* function will release the connections memory.

The *class()* function returns a unique *void ** for the connection that represents the underlying protocol. It is used by function like *metisEtherConnecion_IsInstanceOf()* to determine if a connection is of a particular type.

4.6 MetisConnection

A *MetisConnection* is a PARC-style object that encapsulates a `MetisIoOps` for storage in the Connection Table. It supports the common functions like *acquire()* and *release()*. Other tables store the Connection ID instead of a reference to a Connection. This allows the connection to be taken down or removed without needing to flush all other objects that reference the Connection, such as *MetisMessage* and FIB entries.

If a connection is removed while there are still references to its connection ID in the system, they will be lazily purged when they try to reference the connection ID in the connection table.

4.7 Connection Table

The ConnectionTable stores the state of every connection known to Metis. These include configured connections and tunnels (connections to remote systems) and ephemerally learned connections such as receiving a UDP or Ethernet packet.

Ephemeral connections will timeout. TCP connections automatically timeout when the TCP session ends, as that causes a socket error that causes the connection to go to DOWN then CLOSED state and the Connection Manager will remove it. UDP and Ethernet connections need to manage their own timeout and eventually go to DOWN and CLOSED state to be removed from the Connection Table.

Currently, UDP and Ethernet connections are not timing out.

```
struct metis_io_ops {
    void *closure;
    bool (*send)(MetisIoOperations *ops, const ↩
        CPIAddress *nexthop, MetisMessage *message↩
        );
    const CPIAddress *        (*getRemoteAddress)(↩
        const MetisIoOperations *ops);
    const MetisAddressPair * (*getAddressPair)(↩
        const MetisIoOperations *ops);
    bool (*isUp)(const MetisIoOperations *ops);
    bool (*isLocal)(const MetisIoOperations *ops);
    unsigned (*getConnectionId)(const ↩
        MetisIoOperations *ops);
    void (*destroy)(MetisIoOperations **opsPtr);
    const void * (*class)(const MetisIoOperations *↩
        ops);
    CPIConnectionType (*getConnectionType)(const ↩
        MetisIoOperations *ops);
};
```

Figure 8. MetisIoOps

4.8 Message Processor

The Message Processor has an Interest and a Content Object processing path. These paths execute the normal CCNx 1.0 algorithm for each message type. Figure 9 shows the two processing paths.

An Interest message carries a HopLimit, which must be decremented if received from a remote system. If the Interest is from a local application, the HopLimit is not decremented on receive. If an interest is aggregated in the PIT, then the message processor is done. If the message is not aggregated – it's a new Interest or the PIT determines it should be forwarded anyway – then the Message Processor tries to satisfy from the Content Store (if configured), and the tries to forward via the FIB.

If an Interest is satisfied from the Content Store, the corresponding Content Object is sent to the ingress Connection's *send()* function. If the interest is forwarded via the FIB, it is replicated for each next hop and sent via each next hop's *send()* function.

If the message is a Content Object, it is matched against the PIT. If a hit is found, the message is replicated for each previous hop and sent to that connection's *send()* function.

Finally, if the message is a Control packet, it is sent to the Configuration module. If the message is not any of a Content Object, Interest, or Control it is dropped.

Metis currently does not implement the *InterestReturn* message.

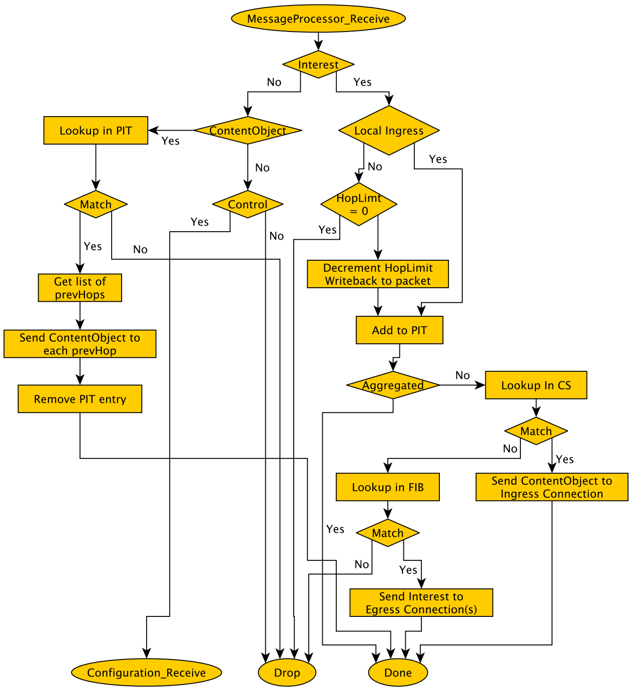

Figure 9. Message Processor Receive

```
struct metis_pit {
    void (*release)(MetisPIT **pitPtr);
    MetisPITVerdict (*receiveInterest)(MetisPIT *↩
        pit, MetisMessage *interestMessage);
    MetisNumberSet * (*satisfyInterest)(MetisPIT *↩
        pit, const MetisMessage *objectMessage);
    void (*removeInterest)(MetisPIT *pit, const ↩
        MetisMessage *interestMessage);
    MetisPitEntry * (*getPitEntry)(const MetisPIT *↩
        pit, const MetisMessage *interestMessage);
    void *closure;
};
```

Figure 10. MetisPIT Interface

4.8.1 PIT Table

Metis includes one PIT implementation, *MetisStandardPIT* which implements the *MetisPIT* interface, shown in Figure 10. When the Message Processor receives an Interest, it calls *receiveInterest()* and the PIT table returns a PIT Verdict indicating if the Interest should is aggregated or should be forwarded. When the Message Processor receives a Content Object, it calls *satisyInterest()* and gets back a set of Connection IDs to forward the Content Object to. The list could be empty if no match is found.

4.8.2 FIB Table

TBD

4.8.3 ContentStore

A ContentStore implements the *MetisContentStoreInterface*, shown in Figure 11. When the Message Processor receives a Content Object that it wishes to cache, it calls *putContent()*, which may evict an older item. The eviction policy is up to the content store implementation. When the Message Processor receives an Interest that is not already in the PIT, it tries to satisfy it by calling *matchInterest*. If a match is found, it returns the MetisMessage of the corresponding Content Object.

4.9 Connection Manager

The Connection Manager is a *MetisMissive* listener. When it receives connection event messages, it forwards them

```
struct metis_contentstore_interface {
    bool (*putContent)(MetisContentStoreInterface *storeImpl, MetisMessage *content, uint64_t currentTimeTicks);
    bool (*removeContent)(MetisContentStoreInterface *storeImpl, MetisMessage *content);
    MetisMessage * (*matchInterest)(MetisContentStoreInterface*storeImpl, MetisMessage *interest);
    size_t (*getObjectCapacity)(MetisContentStoreInterface *storeImpl);
    size_t (*getObjectCount)(MetisContentStoreInterface *storeImpl);
    void (*log)(MetisContentStoreInterface *storeImpl);
    MetisContentStoreInterface *(*acquire)(const MetisContentStoreInterface *storeImpl);
    void (*release)(MetisContentStoreInterface **storeImpl);
    void *_privateData;
};
```

Figure 11. ContentStore Interface

to applications that have registered (TBD) and cleans up the Connection Table for connections that have gone away.

For example, when the Connection Manager receives a CLOSED signal for a connection, it will remove that connection from the connection table and remove it as a next hop from all routes.

The connection manager queues received Missives and processes them in a later Dispatcher scheduling time. This avoid conflict with other Missive receivers.

5. Programming Tasks

This section describes how to modify certain components of Metis to evaluate different technologies or change the behavior. The modular pieces are the PIT, FIB, Content Store, and protocol Listeners and IO Connections.

5.1 Replacing the PIT Table

A PIT table must implement the *MetisPIT* interface. The included *MetisStandardPIT* implements the CCNx 1.0 specification for the PIT table.

To replace the standard PIT with a customized PIT, change the call to *metisStandardPIT_Create()* in *metisMessageProcessor_Create()* to the new constructor. There should be no additional changes.

There is currently no means to choose a PIT table implementation by configuration.

5.2 Replacing the Content Store

A Content Store must implement the *MetisContentStoreInterface* interface. In the *metisMessageProcessor_Create()* function, simply replace the call to *metisLRUContentStore_Create()* with your own Content Store implementation.

Metis allows the size of the Content Store to be set via configuration. This results in a call to *metisMessageProcessor_SetContentStoreSize()*. You should edit this function to use whatever means you implement for a replacement Content Store. The LRU ContentStore simply releases itself and creates a new one, which does result in losing all cached content.

There is currently no means to choose a Content Store implementation by configuration.

5.3 Adding a new I/O Protocol

An I/O Protocol has four pieces: the ProtocolListener, the ProtocolConnection, the ProtocolTunnel, and the ProtocolConfiguration. The first three pieces live in the “io” directory and the configuration piece lives in the “config” directory.

For purposes of explanation, lets use SCTP as an example new protocol. The new modules to add to Metis would be *SCTPListener*, *SCTPConnection*, *SCTPTunnel*, and add configuration options to *metisControl_AddListener*, *metisControl_RemoveListener*, *metisControl_AddConection*, *metisControl_RemoveConection*, *metis_Configuration*, and *metis_ConfigurationListeners*.

The *MetisConfiguation* components will be refactored to allow a more modular approach to adding protocols.

5.3.1 The I/O pieces

The protocol listener, in our example *SCTPListnener*, sets up the server socket for the protocol. The listener would function much like the UDP listener, using *bind(), listen(), and recvmsg()* with a `SOCK_SEQPACKET` socket type. It would accept packets and determine if it matched an existing connection. If not, it would create a connection and *SCTPConnection* object to put in the Connection Table.

Because one would want to send a reply packet from the server socket address, the *SCTPConnection* would use the same socket as the *SCTPListener*.

The *SCTPTunnel* module is used by the configuration system to create an out-bound connection to a remote system. Like the *UDPTunnel*, its main job is to lookup the appropriate *SCTPListener* – so it can borrow the socket – and then create a *SCTPConnection* and put it in the connection table.

An alternate approach would be use SCTP in one-to-one mode, in which case it would follow the *TCPListener, TCPConnection* model.

5.3.2 The Configuration pieces

The configuration process requires updates to each of these sections to enable configuration via `metis_control` and a configuration file.

metisControl_AddListener Define the “ADD LISTENER” command syntax for the listener.

metisControl_RemoveListener Define the “REMOVE LISTENER” command syntax for the listener.

metisControl_AddConection Define the "ADD CONNECTION" command syntax. For IP based protocols, it will likely fall in to the *_metisControlAddConnection_ParseIPCommandLine* format and use the *_metisControlAddConnection_IpHelp* help display.

metisControl_RemoveConection Define the "REMOVE CONNECTION" command syntax.

The result of these *metisControl_X* functions is a CPI control object that can be sent down the protocol stack and encoded to Metis for configuration. These code modules create a *CCNxMetaMessage* and pass it to *ccnxControlState_WriteRead()*.

The *ccnxControlState_WriteRead()* function is program specific. For a program like `metis_control`, it will result in the *CCNxMetaMesage* being sent down the protocols stack to in to Metis via a network channel. For parsing the configuration file within Metis, it will result in the message being handed off directly to *MetisConfiguration*.

metisConfiguration_ProcessCreateTunnel Add a handler to *SCTPTunnel_Create()*.

metisConfiguration_ProcessRemoveTunnel Add a handler to move the connection to CLOSED state.

metisConfigurationListeners_Add Add a handler to *SCTPListener_Create()*..

metisConfigurationListeners_Remove Add a handler to close all connections using the listener and remove the Listener.