

MAC Type Independent Nemo Architecture

Sebastien Roy

Project Clearview I-Team
clearview-iteam@sun.com

Network Approachability
Sun Microsystems, Inc.

Revision 0.1
December 6, 2005

Contents

1	Introduction	1
2	MAC Type Plugins	1
2.1	Plugin Registration	1
2.2	Plugin Operations	2
2.2.1	mtops_unicst_verify()	2
2.2.2	mtops_multicst_verify()	3
2.2.3	mtops_sap_verify()	3
2.2.4	mtops_header()	4
2.2.5	mtops_header_info()	4
2.3	Loading Plugin Kernel Modules	5
3	Internal Nemo Changes	5
3.1	Responding to DL_INFO_REQ	5
3.2	Multicast and Broadcast	6
3.2.1	Enabling a Multicast Address	6
3.3	i_dls_link_ether*()	6
3.4	Link Aggregations	7
3.5	Ethernet VLANs	7
A	DLEther Plugin	7
A.1	mac_ether_unicst_verify()	7
A.2	mac_ether_multicst_verify()	7
A.3	mac_ether_sap_verify()	7
A.4	mac_ether_header()	8
A.5	mac_ether_header_info()	8

1 Introduction

The GLDv3 framework (aka Nemo) currently supports Ethernet drivers that implement a DL_ETHER MAC type. Although Ethernet is by far the most common networking media in use, if GLDv3 is to be as generic as its acronym suggests, all networking drivers implementing arbitrary MAC types should be able to use it.

This design describes modifications to Nemo that will allow the framework to be extended to support arbitrary MAC types. A plugin architecture will be implemented that will allow the framework to be media type independent and allow it to execute media type specific tasks by using registered plugin entry points. Each plugin will implement media specific tasks for a given media type.

The driving force behind this work is the Clearview project¹. The IP Tunneling Device Driver² component of Clearview implements a Nemo MAC driver that define the DL_IPV4, DL_IPV6, and DL_6T04 MAC types. The Vanity Naming and Nemo Unification³ component of Clearview implements a MAC shim that allows drivers of arbitrary MAC type to use the Nemo framework. Both of these Clearview components will provide plugins for this architecture.

A plugin architecture was chosen for three main reasons:

1. To remove MAC type specific code from the Nemo framework⁴.
2. To allow support for new MAC types to be added at any time without recompiling the framework.
3. To not burden driver writers with tasks that are MAC type specific but not specific to their particular driver.

In addition to the MAC type plugin architecture, other changes will need to be made to the Nemo kernel modules to offload functionality to the plugins and to remove functionality that is unnecessarily MAC type specific.

2 MAC Type Plugins

MAC Type plugins will register with the Nemo `mac` module. Each plugin will define its MAC type, MAC address length, media broadcast address, and provide callbacks for various MAC type specific operations.

As the Nemo framework currently only supports the DL_ETHER MAC type, a DL_ETHER plugin will be initially delivered as part of this proposal to maintain support for existing Nemo drivers. This plugin is described in appendix A.

2.1 Plugin Registration

Each plugin module will register itself with the `mac` module using the following new function:

¹<http://clearview.east>

²http://clearview.east.sun.com/wiki/index.php/Clearview:IP_Tunnel_Device

³http://clearview.east.sun.com/wiki/index.php/Clearview:Vanity_Naming

⁴The implementation of Ethernet VLANs will remain in the core Nemo framework (in the `dls` module), as this feature's design is too tightly intertwined with the framework to modularize. The Ethernet link aggregation feature is already a separate module that does not need to be untangled from the core framework.

```
int mac_type_register(char *plugin_ident, uint_t mac_type, uint_t addr_length,
    uint8_t *brdcst_addr, mactype_ops_t *ops);
```

The `plugin_ident` argument is a NULL terminated string used to uniquely identify the plugin. MAC drivers will use this identification string to choose which MAC type plugin to use. For example, a generic “Ethernet” plugin implementing the `DL_ETHER` MAC type will be provided (see appendix A). The reason that the MAC type itself won’t be used by drivers to choose plugin modules is that developers may wish to have multiple plugins provide the same MAC type as far as Nemo and DLPI consumers are concerned, but for different underlying technology (WIFI vs. Ethernet for example).

The `mac_type` argument is type implemented by the plugin, and is the value that the `dld` module will use as the `dl_mac_type` in `DL_INFO_ACK` messages.

The `addr_length` argument is the length of MAC addresses for the given type.

The `brdcst_addr` argument is an optional pointer to a buffer of length `addr_length` that contains the media broadcast address for the MAC type. The pointer can be NULL if no broadcast address is supported. The assumption made here is that there can’t be more than one broadcast address for a given media type.

The `ops` argument is pointer to a structure containing pointers to functions that implement the MAC type specific operations. The structure is described in section 2.2.

2.2 Plugin Operations

When registering with `mac_type_register()`, plugins supply a set of callbacks in a structure defined as follows:

```
typedef mac_type_ops {
    uint_t                mtops_ops;
    mtops_addr_verify_t  *mtops_unicst_verify;
    mtops_addr_verify_t  *mtops_multicst_verify;
    mtops_sap_verify_t   *mtops_sap_verify;
    mtops_header_t       *mtops_header;
    mtops_header_info_t  *mtops_header_info;
} mac_type_ops_t;

/*
 * mtops_ops allows the plugin to enumerate the callback entrypoints it has
 * defined.
 */
#define MTOPT_UNICST_VERIFY    0x001
#define MTOPT_MULTICST_VERIFY 0x002
#define MTOPT_SAP_VERIFY      0x004
#define MTOPT_HEADER          0x008
#define MTOPT_HEADER_INFO     0x010
```

Plugins will set bits in `mtopt_ops` corresponding to the callbacks that have been set. This allows the framework to define additional callbacks without having to recompile plugins.

Each callback is described in subsequent sections.

2.2.1 mtops_unicst_verify()

This callback is of the following type:

```
typedef int (mtops_addr_verify_t)(const uint8_t *addr);
```

The `mac` module exports `mac_unicst_set()` for setting the MAC address. It is used by the `dld` module when a DLPI consumer issues a `DL_SET_PHYS_ADDR_REQ`, and by the `aggr` module to set the MAC address of an aggregation.

The `mac_unicst_set()` function calls a driver specific callback stored in the `mac_t` (`m_unicst()`) that sets the physical address of the device. Before it issues this callback, it attempts to validate the format of the address using the `mi_unicst_verify()` function pointer stored in `mac_impl_t`, which is currently hard-coded to the Ethernet specific function `i_mac_ether_unicst_verify()`.

This callback supersedes the `mac_impl_t`'s existing `mi_unicst_verify()` function pointer. It takes a single argument, a pointer to a buffer containing the unicast address to verify. The callback does not take a length argument, as the address is guaranteed by the framework to be of the length specified by the MAC type plugin when it registered. The callback returns 0 for success, or a non-zero error code such as `EINVAL` for failure.

2.2.2 mtops_multicst_verify()

This callback is of the following type:

```
typedef int (mtops_addr_verify_t)(const uint8_t *addr);
```

The `mac` module exports `mac_multicst_add()` for adding a multicast group address. It is used by the `dld` module when a DLPI consumer issues a `DL_ENABMULTI_REQ`, and by the `aggr` module to add a group to its underlying MACs when one of its `dld` consumers issues a `DL_ENABMULTI_REQ`.

The `mac_multicst_add()` function calls a driver specific callback stored in the `mac_t` (`m_multicst()`) to add the group. Before it issues this callback, it attempts to validate the format of the address using the `mi_multicst_verify()` function pointer stored in `mac_impl_t`, which is currently hard-coded to the Ethernet specific function `i_mac_ether_multicst_verify()`.

This callback supersedes the `mac_impl_t`'s existing `mi_multicst_verify()` function pointer. It takes a pointer to a buffer containing the multicast address to verify. Like the unicast callback, the address is guaranteed to be of the length specified by the MAC type plugin when it registered. The callback returns 0 for success, or a non-zero error code such as `EINVAL` or `ENOTSUP` for failure.

2.2.3 mtops_sap_verify()

This callback is of the following type:

```
typedef boolean_t (mtops_sap_verify_t)(uint_t sap, uint_t *bind_sap);
```

The `<sys/dls.h>` header file currently defines the `SAP_LEGAL()` macro that returns true or false depending on whether a given SAP value is a valid Ethernet SAP. This macro will be removed

and replaced by this plugin callback. The callback will return true or false depending on whether the given SAP value is legal for the MAC type.

In addition, the caller will optionally pass in a pointer to a variable in the `bind_sap` argument to request the callback to fill in the value of the SAP that should be used internally by the Nemo framework. For example, Ethernet SAPs 0-1500 are all LLC channels, and all map to the same SAP value of 0. Passing in a SAP of 15 to the Ethernet plugin's function would yield a `bind_sap` of 0.

In order to make this mechanism available to the Nemo layers above the `mac` module, a new `mac` client interface named `mac_sap_verify()` will be defined which will in turn invoke this plugin callback.

2.2.4 `mtops_header()`

This callback is of the following type:

```
typedef mblk_t *(mtops_header_t)(uint8_t *saddr, const uint8_t *daddr,  
    uint_t sap, void *mac_header_data, size_t extra_len);
```

The `dls` module's `dls_impl_t` contains a function pointer named `di_header` that is called to construct a link-layer header. It is used by the `dld` module through `dls_header()` to pass up to DLPI consumers when negotiating fast-path⁵ and to send packets when `DL_UNITDATA_REQ` is used. Currently, this function pointer is unconditionally set to `i_dls_ether_header()`, as Ethernet is the only supported media.

This callback supersedes the `dls_impl_t`'s existing `di_header()` function pointer. It allocates an `mblk_t` containing a link-layer header that has been initialized using the parameters given.

The `saddr` and `daddr` arguments are the desired source and destination link-layer addresses respectively, and the `sap` argument is the consumer's bound SAP.

The `mac_header_data` argument is MAC type specific data that is optionally provided by each driver at driver registration time. Each MAC type plugin can define the format of its `mac_header_data`, and most types (including Ethernet) will not define any required data.

The `extra_len` argument tells the callback how many additional bytes to allocate following the link-layer header. In most cases, the caller will pass in 0, but this allows things like the VLAN implementation in `dls` to fill in the additional bytes of data following the Ethernet header with the contents of the VLAN header. The callback will set `b_wptr` to the end of the link-layer header, and it will be the responsibility of the caller to advance `b_wptr` if it fills in the additionally allocated data.

In order to make this mechanism available to the Nemo layers above the `mac` module, a new `mac` client interface named `mac_header()` will be defined, which will in turn invoke this plugin callback.

2.2.5 `mtops_header_info()`

This callback is of the following type:

```
typedef int (mtops_header_info_t)(mblk_t *mp, mac_header_info_t *mhip);
```

⁵It does this when processing `DLIOCHDRINFO` ioctls

The `dls` module's `dls_impl_t` contains a function pointer named `di_header_info` that is called to extract the link-layer header from a given packet. It is used to construct `DL_UNITDATA_IND` messages when in non-fast-path mode. Currently, this function pointer is set to `i_dls_ether_header_info()`, as Ethernet is the only supported media.

This callback supersedes the `dls_impl_t`'s existing `di_header_info` function pointer.

The `mp` argument points to an `mblk_t` containing a packet that begins with a link-layer header. The `mhip` argument is a pointer to a structure to be filled out by the callback.

The `mac_header_info_t` structure will be as follows:

```
typedef struct mac_header_info_s {
    size_t          mhi_length;
    const uint8_t   *mhi_daddr;
    const uint8_t   *mhi_saddr;
    uint32_t        mhi_sap;
    mac_addrtype_t  mhi_dsttype;
    boolean_t       nosource;
} mac_header_info_t;
```

`mhi_length` is the length of the header. `mhi_daddr` and `mhi_saddr` are pointers to the destination and source addresses contained in the header. `mhi_sap` is the destination SAP. `mhi_dsttype` is either `MAC_ADDRTYPE_UNICAST`, `MAC_ADDRTYPE_MULTICAST`, or `MAC_ADDRTYPE_BROADCAST`.

In order to make this mechanism available to the Nemo layers above the `mac` module, a new `mac` client interface named `mac_header_info()` will be defined, which will in turn invoke this plugin callback.

2.3 Loading Plugin Kernel Modules

MAC type plugins must be loaded and registered with the Nemo framework before any MAC drivers attempt to register themselves. Otherwise, a driver's `mac_register()` would fail if its MAC type plugin was itself not yet registered.

A few solutions exist to ensure that the plugins are loaded before their dependent drivers. One could be to have the plugins reside in a dedicated directory, and have the `mac` driver load them all explicitly using `modload()` in its `mac_init()` routine.

Another is to require that each MAC driver module be compiled with a `DT_NEEDED` dependency on the MAC type plugin(s) that implements its MAC type(s)⁶. This is simpler, and is the method that will be implemented.

3 Internal Nemo Changes

In addition to the plugin framework described in this document, changes internal to the Nemo framework need to be made to make Nemo truly MAC type independent. The following sections detail those changes.

⁶`DT_NEEDED` entries are added using `ld(1)`'s `-N` option.

3.1 Responding to DL_INFO_REQ

The `dl_info_ack_t` structure contains a broadcast address length and offset for DLPI providers that support broadcast. The existing Nemo framework assumes that broadcast is always supported and always includes a broadcast address in its `DL_INFO_ACK` messages. The `mac_info_t` structure will be changed so that instead of an in-line `mi_brdcst_addr` address of size `MAXADDRLEN`, `mi_brdcst_addr` will become a pointer to the broadcast address registered by the MAC type plugin. If the pointer is `NULL`, then `dld` will know not to include a broadcast address in its `DL_INFO_ACK` messages.

The `mi_unicst_addr` field of the `mac_info_t` structure will be modified to also be a pointer for uniformity.

3.2 Multicast and Broadcast

Not all media types support the concepts of multicast and broadcast. Specifically, the IP tunnel types implemented by the IP tunnel driver support neither multicast nor broadcast. Given that the only media currently supported by Nemo is Ethernet, the framework assumes that the MAC driver supports both multicast and broadcast.

3.2.1 Enabling a Multicast Address

One such assumption is made when `dld` receives a `DL_ENABMULTI_REQ` primitive to enable a multicast address. The request goes through `proto_enabmulti_req()`, then `dls_multicst_add()`, then `mac_multicst_add()`, which then invokes a registered callback in the driver called `m_multicst`. This will be changed so that the driver has the option of setting this callback to `NULL` if it does not support multicast, and `mac_multicst_add()` will return `ENOTSUP` if it encounters a `NULL` callback. This error will trickle back up to `proto_enabmulti_req()`, which will send back a `DL_ERROR_ACK` message with a `dl_errno` of `DL_NOTSUPPORTED` in this case.

3.3 `i_dls_link_ether*()`

Almost every aspect of receiving packet mblk chains in `dls_link.c` is currently Ethernet specific. The `i_dls_link_ether*()` routines, all private to the `dls` module, implement essential data manipulation functions that can and need to be generalized.

1. `i_dls_link_ether_rx()`

This function is registered with the `mac` module as a receive function using `mac_rx_add()`. It will be renamed to `i_dls_link_rx()`. It currently calculates the SAP associated with packet subchains based on the ether-type of the Ethernet headers. It will be modified to rely on the `i_dls_link_subchain()` function (described below) to provide it with the correct SAP for each packet subchain.

2. `i_dls_link_ether_rx_promisc()`

This function is analogous to `i_dls_link_ether_rx()`, except it is used to receive packets when promiscuous mode has been enabled on the link. The same changes will be made to this function as to `i_dls_link_ether_rx()`, and it will be renamed to `i_dls_link_rx_promisc()`.

3. `i_dls_link_ether_loopback()`

This function is used to loopback transmitted packets when the MAC is in promiscuous mode. It performs similar functions to the `rx` functions above. The same changes will be made to this function, and it will be renamed to `i_dls_link_txloop()`.

4. `i_dls_link_ether_subchain()`

This function is used by the above three functions to separate a chain of packets (implemented as `mblk` structures linked by `b_next`) into sub-chains of related packets (Ethernet packets with the same addresses and ethertypes). It will be renamed to `i_dls_link_subchain()`, and will be changed to handle non-Ethernet packets. It will group packets that have identical source and destination addresses as well as identical destination SAPs. It will obtain this information in a MAC type independent way by calling `mac_header_info()` (described in section 2.2.5) for each packet.

3.4 Link Aggregations

The link-aggregation feature implemented by Nemo assumes that the links it aggregates are all Ethernet links. The implementation will be modified to check for a MAC type of `DL_ETHER` when adding MACs to an aggregation. Specifically, the `aggr_port_create()` function will verify that the `mi_media` field of the underlying MAC's `mac_info_t` structure is `DL_ETHER`, and return an error if it is not.

3.5 Ethernet VLANs

Nemo's Ethernet VLAN creation function, `dls_vlan_create()`, does not verify if the underlying MAC is of type `DL_ETHER`. It will be modified to do this check, and to return an error if it is not.

A DL_ETHER Plugin

The `DL_ETHER` plugin will register as follows:

```
mac_type_register('Ethernet', DL_ETHER, ETHERADDRL, ether_brdcst,
&mac_ether_type_ops);
```

The `ether_brdcst` and `mac_ether_type_ops` arguments will be defined as follows:

```
static uint8_t ether_brdcst[] = { 0xff, 0xff, 0xff, 0xff, 0xff, 0xff };

static mactype_ops_t mac_ether_type_ops = {
    mac_ether_unicst_verify,
    mac_ether_multicst_verify,
    mac_ether_sap_verify,
    mac_ether_header,
    mac_ether_header_info
};
```

A.1 `mac_ether_unicst_verify()`

This function will return 0 if the Ethernet address given is not a group address, and `EINVAL` otherwise. A group address is one whose first byte has the least significant bit set.

A.2 `mac_ether_multicst_verify()`

This function will return 0 if the Ethernet address given is a group address and is not equal to the Ethernet broadcast address. It will return `EINVAL` otherwise.

A.3 `mac_ether_sap_verify()`

This function will return `B_TRUE` if the `sap` argument is in the range 0 to 1500, or in the range `ETHERTYPE_802_MIN` (1536) to `ETHERTYPE_MAX` (65535). It will return `B_FALSE` otherwise.

If the `bind_sap` argument is non-null, it will set it to `DLS_SAP_LLC` (0) if the given `sap` is in the range 0 to 1500 (these represent LLC channels), or to the value of `sap` if `sap` is in the range `ETHERTYPE_802_MIN` to `ETHERTYPE_MAX`.

A.4 `mac_ether_header()`

This function will allocate an `mblk_t` of length (`sizeof (struct ether_header) + extra_len`). It will then fill in the data at `b_rptr` by casting it to `struct ether_header` and filling in the Ethernet header information using the parameters given. `b_wptr` will be advanced to point at the end of the Ethernet header.

The `mac_header_data` argument is not used, as this plugin does not require additional data from drivers in order to construct Ethernet headers.

A.5 `mac_ether_header_info()`

This function will fill in the given `mac_header_info_t` using the Ethernet header provided in `mp`.

The `mhi_sap` member will be set to 0 if the `ether_type` of the Ethernet header is between 0 and 1500, or to the `ether_type` value otherwise.