Multidata Interface Design SpecificationenoteCopyright 2002, Sun Microsystems

Revision 1.2 – September 11, 2002

Adi Masputra
Internet Engineering - Data Movement
multidata-core@Sun.COM

ABSTRACT

The current model for packet transmission and reception throughout the Solaris TCP/IP stack and network device driver is geared towards sending and receiving one packet at a time. Because of the high per-packet costs in the Solaris stack, this model consumes a lot of the system processing time and has an impact on the networking performance. For example, it makes it hard for Solaris to saturate the 1 gigabit per second (Gbps) line rate on a single TCP connection.

This document introduces a Multidata mechanism which allows for more than one packets to be sent from one module to another in a given call, thereby reducing the per-packet processing costs. This translates to improving the host CPU utilization and/or network throughput.

1. INTRODUCTION

We encourage the reader to familiarize himself with our Multidata experiments and findings [5] before proceeding further with this document. Other related information can also be found in the submitted one-pager [4].

In the following sections, we describe how a Multidata message is defined and used. We also elaborate on the supplied kernel library function calls (APIs) which allow for a module to easily create, access and manipulate the contents of a Multidata message. We then describe how we implement the Multidata framework, and list the required operating system changes.

2. MULTIDATA MESSAGE

A Multidata message is made up of a metadata, and a *header* and/or a *payload* buffer. The metadata describes various properties of the Multidata related to the packets it represents, such as their count, layout, and any optional attribute they may have.

The naming of the buffers does not correlate to any requirement about their format and/or contents. In other words, Multidata places no restrictions on the type or format of data contained in either buffers, as it expects the modules communicating using Multidata messages to be knowledgeable about each other, either *a priori* or through some form of negotiation.

The main difference between the buffers, is that the header buffer allows for gaps to be defined between each fragment. The gap or spacing between each header fragment provides support for header alignment, and enables an intermediate module to insert additional protocol headers to the existing one(s) given enough space before and/or after the header fragment. Ideally, the latter would be done by prepending the new header and adjusting the "current" header pointer, but the final decision as to what this extra space is for is left to the users of Multidata.

The normal form of Multidata is somewhat similar to that of the *header-splitting* scheme where the header and payload of a packet reside in separate buffers, except that more than one packets can be represented within a Multidata message. This form requires both header and payload buffers to be associated with a Multidata message (Figure 1), where each contains user-defined data fragments opaque to Multidata.

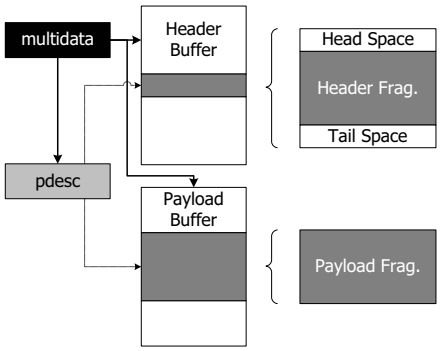

Figure 1: Two Buffers Multidata

The other Multidata form involves only one buffer association, on which the buffer type (header or payload) is insignificant to Multidata (Figure 2).

Multidata also allows for attributes to be associated with the packets it represents, e.g. source/destination address, checksum information, etc. An attribute may be shared by all packets or is specific only to a packet.

2.1 M_MULTIDATA Data Block Type

Four data block types are currently used by IP and the Data Link Provider (DLP) to exchange messages/packets: M_PROTO and its high-priority sibling M_PCPROTO are used for DLPI; M_DATA is used for the "M_DATA fastpath" [2]; and M_CTL is used for various purposes by IPSEC [1].

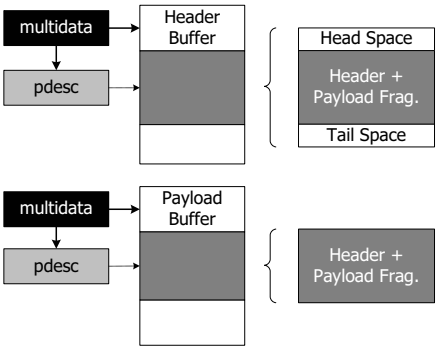

Figure 2: One Buffer Multidata

While it is tempting to overload one of these types to represent a Multidata message, there are several limitations and drawbacks to this approach -

- Multidata is not a DLPI extension, but is rather generic to STREAMS. The module producing and/or consuming a Multidata message may not be using DLPI to communicate. For instance, a Multidata message may be produced by a transport layer or its upper layer protocol, and consumed by the network driver during transmission – the reverse may hold true during reception.

  To make matters worse, various networking modules already use M_PROTO or M_PCPROTO to communicate to each other, i.e. using such data block type to represent Transport Provider Interface (TPI) messages.

  Overloading these types would be far from ideal, since doing so would require every module in the corresponding stream to be modified to correctly interpret the Multidata message on top of the existing M_PROTO or M_PCPROTO layer-specific handlings (TPI, DLPI, et al).
- The ad hoc nature of M_DATA data block type seems appealing at first glance, but a closer inspection reveals limitations similar to those of M_{PC}PROTO. More importantly, the "M_DATA fastpath" interface has never been made public (nor has it been ARC'd), despite its heavy use in a variety of network drivers and third-party intermediate modules.

  The dynamic STREAMS module insertion feature [6] is an example which exposes potential problems related to overloading M_DATA type. An intermediate module inserted between IP and DLP having *a priori* knowledge of "M_DATA fastpath" data format will cease to function (or exhibit undesirable behaviors) when presented with a M_DATA message containing the Multidata metadata.
- Not only M_CTL has all of the above limitations, it is already overloaded by many subsystems. In addition, STREAMS discards M_CTL messages at the stream head, making it close to impossible for future Multidata enhancements to be made at that layer.

We therefore propose a new data block type M_MULTIDATA to be used to identify messages containing Multidata metadata, and the advantages include -

- No namespace collision with other data block types, because M_MULTIDATA is strictly defined to carry only Multidata metadata.
- Modules unaware of the new data block type will simply revert to the default action taken when receiving an unregonized message; either forward the message to the next queue or free it. This eliminates any misinterpretation that may arise with "M_DATA faspath" overloading, and at the same time highlights the need for Sun to publish that and the Multidata interfaces to the ISVs.
- A new data block type will ease debugging efforts, and any breakage it introduces will be more tractable compared to the overloaded one.

3. ADMISSIBLE CASES

Multidata may be applied to cases where a source module "packetizes" a block of data prior to sending it to a destination module, such as TCP bulk-data transmission and reception – the latter requires networking hardwares and drivers capable of TCP/IP *header-splitting* combined with *flow-reassembly* features, such as Cassini [7].

3.1 TCP Bulk-Data Transmission

As elaborated in [5], TCP's transmission dynamics may be improved by sending large blocks of application data to it. Depending on the amount of transmittable data, a large portion or perhaps all of each data block in the transmission queue may be "packetized" into smaller pieces, each being at most 1 TCP MSS octets. Such process involves allocating the header buffer with enough space to hold the maximum number of allowable packets for a given data block (i.e. payload buffer) and filling it with the IP and TCP headers. TCP then creates a Multidata message associated with a pair of header and payload buffers, and adds to it the description of the packets, such as their locations and sizes. This process is repeated for each data block in the queue, until there is no more allowable data to transmit.

The Multidata messages (each representing multiple packets) are then sent down to IP for further processings, where it performs route lookups, checksumming, etc., before sending it to the network driver. IP may also add attributes such as those related to physical destination address/SAP and hardware checksumming information.

The driver can then perform further optimizations, such as performing only two IOMMU operations (mapping and flushing/unmapping for the header and payload buffers) for a number of packets represented by each Multidata message.

In the above case, TCP creates both the header buffer and Multidata message, while the payload buffer is supplied by the module above, e.g. stream head, rpcmod, etc.

3.2 TCP Bulk-Data Reception

The Cassini hardware is capable of placing multiple packets of the same flow in a buffer. Each packet received by

the hardware is classifed by the header parser, and in-order packets belonging to the same flow may be broken into header and payload pieces and further "re-assembled". The headers are stored in a page with 256-byte room for each header, i.e. a single 8KB page is broken up into 32 buffers. The payloads are packed together in another page to form a contiguous stream of bytes.

The driver then creates a Multidata message, associates it with the header and payload buffers, and adds the descriptions of the packets to it. It also adds several attributes to the Multidata, such as the physical source and destination address/SAP as well as hardware checksumming information before passing the message to IP.

The above shows a case where both header and payload buffers originate from the same module, in this case the network driver.

3.3 One Buffer Multidata

A slight variation to the above "two buffers" cases is where a Multidata is associated with only a header or payload buffer (see Figure 2). In this case, the buffer contains one or more "complete" packets, where each portion of the buffer representing a packet contains both the protocol header(s) and payload.

3.4 Multidata Chaining

In certain cases, a module may wish to send more than one Multidata messages to a target module on a given call. Because a Multidata is represented by the `mblk_t` structure, the `b_cont` field may be used to link the messages. Modules intending to chain Multidata messages are expected to inform the target modules through some form of negotiation.

4. INTERFACE DESCRIPTIONS

In order to hide the implementation details, most of Multidata processings are done through a set of functions or APIs. The goal is to expose them as DDI/DKI (public) functions so that third-party module and device driver writers are able to easily support Multidata[1].

4.1 Multidata Basics

Most of the data structures used and returned by the functions are opaque to the Multidata client. Those structures (we use the term "handles" hereafter to describe their pointers) are as follows -

- `multidata_t *` - the handle to a Multidata message, which represents a pointer to the opaque metadata structure.
- `pdesc_t *` - packet descriptor handle, which represents a pointer to the opaque packet descriptor structure.
- `pattr_t *` - the handle used to identify and manage an attribute, which is a pointer to the opaque attribute structure.

[1]They are initially classified as Solaris OS/Net *consolidation private*.

4.1.1 Creating Multidata

The first thing we need is the ability to create a Multidata message containing the metadata. Typically, a module calls `allocb(9F)` or `esballoc(9F)` to allocate a message block; the latter allows for user-supplied data buffer to be attached. We can't use `allocb(9F)` to allocate a Multidata message, because the message contains metadata which refers to dynamically-allocated resources requiring special handling during `freeb(9F)`.

We provide a `mmd_alloc()` interface which allocates the Multidata message and initializes the metadata stored within. This interface requires at least one associated buffer.

- **Example.** To associate `hmp` and `pmp` message blocks to the newly-created Multidata message, we say:

```
multidata_t *mmd;
mblk_t *mmd_mp;
(void ) mmd_alloc(hmp, pmp, &mmd_mp,
    &mmd, KM_NOSLEEP);
```

(Note: `hmp` or `pmp` may be substituted with `NULL` to associate only one buffer to the Multidata.) In the above example, we request that the Multidata handle be stored in `mmd` upon success. The handle can also be obtained at any point by calling `mmd_getmultidata()`.

Buffer association with a Multidata message can only be made at creation time, and the relationship stays intact until the message is freed.

4.1.2 Destroying Multidata

The usual `freeb(9F)` or `freemsg(9F)` may be used to deallocate Multidata message block(s). This allows for modules unaware of Multidata to free the resources associated with the message.

4.1.3 Duplicating and Copying Multidata

`dupb(9F)` or `dupmsg(9F)` may be used to duplicate a Multidata message. The metadata contained in the `dblk_t` isn't copied, but a new `mblk_t` pointing to the Multidata data block is created instead.

`copyb(9F)` or `copymsg(9F)` may be used to copy a Multidata message, as it calls an internal copy procedure supplied by the Multidata framework that correctly copies the metadata.

4.1.4 Obtaining Buffer Regions

Properties of the associated buffer(s) may be obtained by calling `mmd_getregions()`. This is necessary for network device drivers in dealing with the IOMMU-related operations, especially those involved in DVMA mappings during transmission [5]. The properties are contained in the `mbufinfo_t` data structure, defined as follows -

```
typedef struct mbufinfo_s {
    uchar_t *hbuf_rptr; /* start address */
    uchar_t *hbuf_wptr; /* end address */
    uchar_t *pbuf_rptr; /* start address */
    uchar_t *pbuf_wptr; /* end address */
} mbufinfo_t;
```

- **Example.** To get the buffer regions associated with Multidata handle `mmd`, we say:

```
mbufinfo_t bufinfo;
(void) mmd_getregions(mmd, &bufinfo);
```

`mmd_getregions()` returns the start and end virtual addresses of the associated buffer(s), which essentially are the `b_rptr` and `b_wptr` values of the corresponding buffer message block(s).

4.1.5 Unsupported STREAMS Operations

Because a Multidata message points to a data buffer containing the metadata, certain operations which modify the contents or size of the message block, such as `msgpullup(9F)`, `pullupmsg(9F)`, or `adjmsg(9F)`, will not apply. A module should not be calling any of those functions, and doing so may damage the metadata integrity[2].

4.2 Packet Descriptor

A packet descriptor contains information about the packet's layout in the associated buffer(s), as represented by the following structure -

```
typedef struct pdescinfo_s {
    uint_t  flags;      /* PDESC_{HBUF,PBUF}_REF */
    uchar_t *hdr_base;  /* start of hdr entry */
    uchar_t *hdr_rptr;  /* start of hdr fragment */
    uchar_t *hdr_wptr;  /* end of hdr fragment */
    uchar_t *hdr_lim;   /* end of hdr entry */
    uchar_t *pld_rptr;  /* start of pld fragment */
    uchar_t *pld_wptr;  /* end of pld fragment */
} pdescinfo_t;
```

As previously mentioned, placing data in the header buffer allows for gaps to be defined between each header fragment, where the empty spaces defining the gaps may exist before and/or after each fragment. The entire space used by each entry in the header buffer is defined as a contiguous area of virtual memory whose span is `[hdr_base, hdr_lim]`, and is a superset of the header fragment `[hdr_rptr, hdr_wptr]`. The header fragment *head gap* is defined as an area which spans `[hdr_base, hdr_rptr)`, whereas *tail gap* is defined as `(hdr_wptr, hdr_lim]`.

The payload fragment is a contiguous area of virtual memory whose span is `[pld_rptr, pld_wptr]`. In essence, the payload buffer is a header buffer without gaps.

The existence of data fragment in the header or payload buffer for the packet is described by the value of `flags` -

- `PDESC_HBUF_REF`. Specifies that the descriptor is referring to a memory region in the header buffer.
- `PDESC_PBUF_REF`. Specifies that the descriptor is referring to a memory region in the payload buffer.

The validity of a packet descriptor is defined by the following criterias -

[2]The STREAMS framework is modified to safeguard against this by means of assertion in debug kernel, and returning failure in non-debug kernel.

1. The descriptor `flags` must specify at least one buffer reference. If a buffer reference is specified, the corresponding buffer must have been associated with the Multidata at creation time.
2. The start and end addresses refer to valid memory regions in the associated buffer(s). Specifically for the header buffer case, `[hdr_rptr, hdr_wptr]` must be a subset of `[hdr_base, hdr_lim]`.

The size of each referred memory region is not restricted. This implies that a zero-length region referenced by a descriptor is allowed.

4.2.1 Adding & Removing Packet Descriptor

Descriptors may be freely added and removed by a module. In the typical case, they are added by the module which creates the Multidata message, i.e. the one which calls `mmd_alloc()`. Adding a descriptor is done by calling `mmd_addpdesc()`.

- **Example.** To create a packet descriptor which refers to the associated header and payload buffers, we say:

```
pdescinfo_t pdi = {
    PDESC_HBUF_REF | PDESC_PBUF_REF,
    0x10000, 0x10020, 0x10048, 0x10068,
    0x20000, 0x205B4 };

pdesc_t *pd;
(void) mmd_addpdesc(mmd, &pdi, &pd, KM_NOSLEEP);
```

  In this example, we specify `0x10000` as the base address of the header area, and that we reserve 32-bytes of head space before the header fragment, which is 40-bytes in length and followed by 32-bytes tail space. The payload fragment begins at `0x20000` with a length of 1460-bytes. The handle of the newly-created descriptor is stored in `pd` upon success. (Note: in the typical usage, the addresses would not be hard-coded.)

Descriptor removal typically happens only in pathological cases (e.g. packet corruption), and can be done by calling `mmd_rempdesc()`. The module acting as the final consumer to a Multidata message doesn't need to explicitly remove individual descriptors prior to freeing the Multidata, because it will be done automatically when `freeb(9F)` or `freemsg(9F)` is called.

4.2.2 Obtaining Packet Descriptor

Descriptors may be accessed sequentially by obtaining the first or last descriptor and iterating through the remaining descriptors in either direction, using the provided set of `mmd_getXXXpdesc()` functions.

- **Example.** To walk forward through all of the packet descriptors, we say:

```
pdescinfo_t pdi;
pd = getfirstpdesc(mmd, &pdi);
while (pd != NULL)
    pd = getnextpdesc(mmd, &pdi);
```

  (Note: We requested the packet layout to be stored

in `pdi`. The information may also be obtained at any moment by calling `mmd_getpdescinfo()`.)

- **Example.** To walk backward through all of the packet descriptors, we say:

```
pd = getlastpdesc(mmd, &pdi);
while (pd != NULL)
    pd = getprevpdesc(mmd, &pdi);
```

4.2.3 Converting Packet Descriptor

A packet descriptor may be "converted" into a M_DATA message, by calling `mmd_transform()`. If the descriptor refers to more than one buffers, the packet fragments will be consolidated. In all cases, this results in one M_DATA message, which points to a newly-allocated memory containing a copy of the packet.

Any attribute associated with the packet will not be converted, simply due to the lack of generic message attribute support in the STREAMS framework[3].

4.2.4 Obtaining Packet Descriptor Statistics

A module may wish to check on the number of packet descriptors before processing the Multidata message. In addition, network drivers may be interested in knowing the amount of header and payload buffer references made by the descriptors, in order for it to obtain the right amount of resources needed to transmit the packets described in the Multidata message [5]. The `mmd_mmd_getcnt()` interface is provided for such purpose.

4.3 Attributes

An attribute is defined as a placeholder for an auxiliary piece of information related to one or more packets in a Multidata message. It may be associated with a specific packet descriptor (*local*), or be shared among all descriptors (*global*). In any scope, the association is unique, i.e. an attribute type may not be associated more than once.

Attributes are optional, user-defined and opaque to Multidata. The supplied attribute-related functions are there to merely manage the resource allocations and associations of the attributes. A module adding an attribute to a Multidata message should not expect Multidata framework to enforce the processing of such attribute on the next consuming module. The modules are expected to be knowledgable about each other, either *a priori* or through negotiation.

Parameters of an attribute are represented by the following structure -

```
typedef struct pattrinfo_s {
    uint_t  type;       /* attribute type */
    uint_t  len;        /* attribute length */
    void    *buf;       /* buffer address */
} pattrinfo_t;
```

The `type` is a user-defined value used to identify the attribute. Each attribute type may be paired with a user-defined data structure representing the contents of the attribute. The type is used by Multidata to detect duplicate association, and to enable searches against a specific attribute. The `length` is the size of memory used by the user-defined data structure of an attribute. The address of Multidata-allocated user data attribute area is specified in `buf`.

[3]Hopefully, this would one day be addressed by the "`mblk_t` attribute extension" effort.

4.3.1 Adding and Removing Attribute

Adding a particular attribute involves figuring out the type of association (local or global), as well as supplying the attribute type and the required memory size. The actual memory allocation and association management are done by the Multidata framework through the `mmd_addpattr()` interface.

- **Example.** To allocate space for a global attribute of type `DUMMY_ATTR`, we say:

```
#define DUMMY_ATTR 0x100
typedef struct dummy_s {
    uint_t arg1;
    uint_t arg2;
} dummy_t;

pattrinfo_t pai = {
    DUMMY_ATTR, sizeof (dummy_t) };

pattr_t *pa;
pa = mmd_addpattr(mmd, NULL,&pai, KM_NOSLEEP);
if (pa != NULL) {
    dummy_t *attr = (dummy_t *)pai.buf;
    attr->arg1 = 0x1973;
    attr->arg2 = 0x1020;
}
```

(Note: A packet descriptor handle may be specified in the second argument to establish a local association to the corresponding descriptor.)

As in the packet descriptor case, attribute removal typically applies to pathological cases, and can be done by calling `mmd_rempattr()`. All attributes related to a Multidata will be freed automatically when `freeb(9F)` or `freemsg(9F)` is called on the message.

4.3.2 Obtaining Attribute

Unlike the sequential access nature of packet descriptors, accessing an attribute requires an up-front knowledge of the expected attribute and its type. This is designed to increase efficiency throughout the intermediate modules along the path, since it is assumed that they don't share identical interest on the attributes, and that the ordering is not important. The `mmd_getpattr()` function is provided for this.

- **Example.** To obtain a known global attribute of type `DUMMY_ATTR`, we say:

```
pattrinfo_t pai = { DUMMY_ATTR };

pa = mmd_getpattr(mmd, NULL, &pai);
if (pa != NULL) {
    uint_t arg1, arg2;
    dummy_t *attr = (dummy_t *)pai.buf;
```

```
        ASSERT(pai.len >= sizeof (dummy_t));
        arg1 = attr->arg1;
        arg2 = attr->arg2;
    }
```

5. MULTIDATA INTERFACE SUMMARY

```
int mmd_alloc(
    mblk_t *hdr_mp,      /* header buffer */
    mblk_t *pld_mp,      /* payload buffer */
    mblk_t **pmmd_mp,    /* new mblk */
    multidata_t **pmmd,  /* pointer to handle */
    int kmflags);        /* kmem flags */
```

`mmd_alloc()` creates a Multidata message and requires the caller to supply at least one buffer represented by a message block. It performs the necessary work to allocate and initialize the Multidata message, and returns the newly allocated message block of type M_MULTIDATA to the caller on success through `pmmd_mp`. In addition, the Multidata handle of the newly-created message may be returned through the optional `pmmd` argument. It returns 0 on success, or `ENOMEM` if allocation fails. If both `hdr_mp` and `pld_mp` are `NULL`, or if `pmmd_mp` is not supplied, it returns `EINVAL`. (Note: Upon success, the caller must not free the associated buffer message blocks).

```
multidata_t *mmd_getmultidata(
    mblk_t *bp);         /* message block */
```

`mmd_getmultidata()` accepts a Multidata message block and returns the associated Multidata handle. It returns `NULL` if `bp` does not point to a Multidata message block.

```
int mmd_getregions(
    multidata_t *mmd,    /* multidata handle */
    mbufinfo_t *info);   /* pointer to info */
```

`mmd_getregions()` returns the start and end virtual memory addresses of the associated buffer(s) into the `mbufinfo_t` structure. It returns 0 upon success, or `EINVAL` if both arguments are not supplied.

```
int mmd_getcnt(
    multidata_t *mmd,    /* multidata handle */
    uint_t *hbuf_ref     /* header buffer ref. */
    uint_t *pbuf_ref);   /* payload buffer ref. */
```

`mmd_getcnt()` returns the number of packet descriptors in a Multidata. If the optional `hbuf_ref` or `pbuf_ref` argument is specified by the caller, the total reference count to the corresponding buffer made by the descriptors will be returned through the argument. It returns `EINVAL` if the Multidata handle argument is not supplied.

```
int mmd_addpdesc(
    multidata_t *mmd,    /* multidata handle */
    pdescinfo_t *info,   /* packet layout */
    pdesc_t **ppd,       /* pointer to handle */
    int kmflags);        /* kmem flags */
```

`mmd_addpdesc()` adds a packet descriptor whose layout across the associated buffer(s) is described in the `pdescinfo_t` structure. Upon success, the newly-created descriptor handle is returned through the `ppd` argument. It returns 0 on success, `EINVAL` if all of the arguments are not supplied, `ENOMEM` if the allocation fails, or `EFAULT` if the packet layout is not valid – see §4.2 for the criterias. (Note: Upon succes, the new descriptor is added to the end of the descriptor list.)

```
void mmd_rempdesc(
    pdesc_t *pd);        /* descriptor handle */
```

`mmd_rempdesc()` removes a packet descriptor from the descriptor list in the Multidata. This function will also free any local attribute associated with corresponding descriptor.

```
pdesc_t *mmd_getfirstpdesc(
    multidata_t *mmd,    /* multidata handle */
    pdescinfo_t *info); /* packet layout */

pdesc_t *mmd_getlastpdesc(
    multidata_t *mmd,    /* multidata handle */
    pdescinfo_t *info); /* packet layout */
```

`mmd_getfirstpdesc()` and `mmd_getlastpdesc()` return the first and last packet descriptor from the descriptor list in the Multidata. The packet layout is returned through the optional `pdescinfo_t` argument. The functions return `NULL` if the descriptor list is empty.

```
pdesc_t *mmd_getprevpdesc(
    pdesc_t *pd,         /* descriptor handle */
    pdescinfo_t *info); /* packet layout */

pdesc_t *mmd_getnextpdesc(
    pdesc_t *pd,         /* descriptor handle */
    pdescinfo_t *info); /* packet layout */
```

`mmd_getprevpdesc()` and `mmd_getnextpdesc()` return the packet descriptor before and after `pd` descriptor. The packet layout is returned through the optional `pdescinfo_t` argument. The functions return `NULL` if the end of the descriptor list has been reached.

```
int mmd_adjpdesc(
    pdesc_t *pd,         /* descriptor handle */
    pdescinfo_t *info); /* packet layout */
```

`mmd_adjpdesc()` allows for the layout of the corresponding packet descriptor to be modified. If returns 0 on success, `ENOENT` if the descriptor has already been removed, or `EINVAL` if both arguments are not supplied. It returns `EFAULT` if the layout is invalid, or if any of the requested fragment span is not a subset of the current span – see §4.2 for details.

```
int mmd_transform(
    pdesc_t *pd,         /* descriptor handle */
    mblk_t **pbp);       /* new mblk */
```

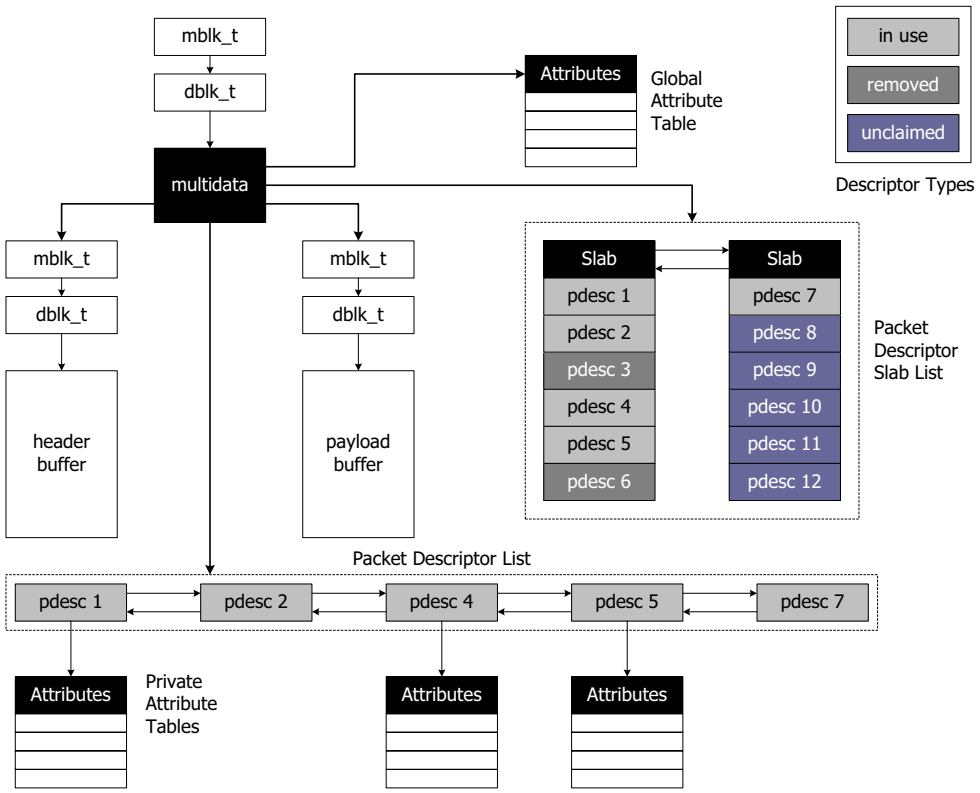

Figure 3: Structure of a Multidata

`mmd_transform()` allocates a M_DATA message and copies over the fragments associated with the descriptor into a newly-created area. It returns 0 on success, along with the newly-created M_DATA message through `pbp`. Otherwise, it returns `EINVAL` if both arguments are not supplied, `ENOENT` if the descriptor has already been removed, or `ENOMEM` if allocation fails.

```
int mmd_dupbufs(
    multidata_t *mmd,    /* multidata handle */
    mblk_t **phmp,       /* duplicated hdr mblk */
    mblk_t **ppmp);      /* duplicated pld mblk */
```

`mmd_dupbufs()` returns a duplicate of the corresponding buffer message blocks specified by the `phmp` or `ppmp` argument. It returns 0 on success, `EINVAL` if both arguments are not specified, or `ENOMEM` if allocation fails.

```
int mmd_getpdescinfo(
    pdesc_t *pd,         /* descriptor handle */
    pdescinfo_t *info); /* packet layout */
```

`mmd_getpdescinfo()` returns the packet layout through the `pdescinfo_t` argument. It returns 0 on success, `ENOENT` if the descriptor has already been removed, or `EINVAL` if both arguments are not supplied.

```
int mmd_addpattr(
    multidata_t *mmd,    /* multidata handle */
    pdesc_t *pd,         /* descriptor handle */
    pattrinfo_t *info,   /* attribute parameters */
    pattr_t **ppa,       /* pointer to handle */
    int kmflags);        /* kmem flags */
```

`mmd_addpattr()` allocates an attribute whose parameters are described by the `pattrinfo_t`. The newly-allocated attribute may be associated globally by specifying `NULL` as the descriptor handle, or locally by supplying a valid packet descriptor. It returns 0 upon success and stores the attribute handle in `ppa`; `EINVAL` if `mmd`, `info`, or `ppa` is not supplied; `ENOMEM` if allocation fails; or `EEXIST` if an attribute of the same type already exists.

```
void mmd_rempattr(
    pattr_t *pa);        /* attrribute handle */
```

`mmd_rempattr()` removes and frees an attribute (and its allocated resources) described by the attribute handle.

```
pattr_t *mmd_getpattr(
    multidata_t *mmd,    /* multidata handle */
    pdesc_t *pd,         /* descriptor handle */
    pattrinfo_t *info); /* attribute parameters */
```

`mmd_getpattr()` returns the handle of an attribute whose type is described in the `pattrinfo_t` argument. Passing in `NULL` as the packet descriptor handle implies that a search on the global attribute list is requested. Otherwise, the search is to be done on the local list associated with packet descriptor `pd`. Upon success, the function returns the attribute han-

dle, along with the length and address of the buffer containing the attribute data through the `pattrinfo_t` argument. It returns `NULL` if a match is not found.

6. MULTIDATA IMPLEMENTATION

In this section we describe how Multidata actually works. Figure 3 depicts the Multidata structure.

6.1 Usage Model

We made several assumptions about the way Multidata is used in the system -

1. Inter-module communication using Multidata messages is done with the modules having full knowledge of the data format and any possible attribute. This is typically achieved through some form of negotiation.
2. A module would use Multidata if the "packetization" process of the payload buffer yields more than one packets. The legacy mechanism should be used instead for the case of one packet, in order to avoid any Multidata overhead.
3. Packet descriptor removal is only for pathological cases. In general, the module which generates the Multidata message typically adds the packet descriptors prior to sending the message over to another module, and intermediate modules typically don't create additional ones. A module may want to remove a descriptor if it detects corruption in the packet.
4. The amount of user-defined attributes is expected to be small. Because attributes contain ancillary information separate from the header and/or payload buffers, there is no hard-rule enforcement on their usages. Therefore, modules expecting such enforcement should embed the information in the associated buffer(s).
5. Duplicating a Multidata message is an uncommon case; modules are expected to check the Multidata data block reference count (`db_ref`) and to take the appropriate actions as necessary, such as making a copy of the message, or risk sharing the metadata and buffer(s) with another thread.

6.2 Packet Descriptor Slab

Packet descriptors are allocated one slab at a time. During system startup time, a kmem cache for the descriptor slab is created, and all slab allocations come out from the descriptor slab cache.

A newly-created slab contains allocated spaces for unclaimed packet descriptors. A separate descriptor list keeps track of the currently-used descriptors in the Multidata. Allocating a packet descriptor involves looking at the last-added slab and checking for any unclaimed descriptor. A new slab is allocated if there are no existing slabs, or if all of the descriptors in the last-added slab have been claimed (either currently-used or removed). Otherwise, the first unclaimed descriptor in the slab is inserted to the end of the descriptor list. Figure 3 illustrates the slab and descriptor list structures.

Descriptor slabs grow as descriptors are added; they never shrink and are not removed until the message is freed.

6.3 Attribute Hash Table

To speed up searching for a particular attribute, the attributes are stored in a hash table with the attribute type as the hash key. The attribute hash table is created on demand during the creation of the first attribute. A Multidata may point to a global attribute hash table, which contains attributes that apply to all packet descriptors. Each packet descriptor may also point to its own individual attribute hash table containing attributes private to the descriptor.

Attribute hash table allocations come out from the hash table cache, which is created during system startup time.

6.4 Locking Strategy

Locks in Multidata exist in order to provide serialization over simultaneous usages by more than one threads. For simplicity, the slab list, descriptor list, and the descriptor reference counts to the buffer(s) are protected with a single lock. Each bucket in the attribute table also holds a lock to protect the chain.

We rely on the assumption that Multidata duplication is a rare event, and that the cost of obtaining uncontended locks are insignificant.

7. REQUIRED OS CHANGES

This section describes the required changes at the operating system level, mostly the STREAMS framework.

7.1 Larger Data Block Cache Sizes

As described in [5], when Multidata transmit is enabled, TCP requests a larger allocation unit at the stream head based on the SO_SNDBUF setting for the socket, which often exceeds the maximum size of data block allocation through the `dblk_t` cache. This may be addressed by the dynamic data block allocator [3], or by increasing the static cache sizes.

8. REFERENCES

[1] Jack Bochsler. Tagging of STREAMS message blocks for IPsec hardware acceleration. September 2001. PSARC/2001/578.

[2] David Butterfield. Solaris network fastpath technical description. November 1998. http://devi.eng/ dab/fastpath.html.

[3] Alexander Kolbasov. Dynamic dblock caches. August 2002. http://streams.eng/alloc/dblk_alloc.pdf.

[4] Adi Masputra. 1-pager: TCP multi-data transmit (MDT). May 2002. http://sac.eng/Archives/Projects/2002/20020513_adi.masputra.

[5] Adi Masputra, Frank Dimambro, and Kacheong Poon. An efficient networking transmit mechanism for solaris: Multi-data transmit (MDT). May 2002. http://arachnid.eng/inet/InternetPerf/multidata/mdt.pdf.

[6] Kacheong Poon. I_MUXID2FD, I_INSERT, and I_REMOVE. 1999. PSARC/1999/348.

[7] Inc Sun Microsystems. *Cassini ASIC Specification Rev. 1.0*, February 2001. http://npweb.ebay/asics/nw_asics/cassini/cas.pdf.