A Versatile Tunneling Interface

Vipul Gupta

vipul.gupta@Eng.Sun.COM

May 15, 1997

1. Introduction

This document describes a software-only network interface, *vtunl*, capable of tunneling IPv4 datagrams. Such tunneling is required by the IETF Mobile-IP protocol [RFC 2002] and may also be used to transport multicast packets across multicast-oblivious portions of a TCP/IP network. The interface is based on *atun*, which is used to tunnel IPv6 packets through an IPv4 network in the Solaris IPv6 prototype.

2. Basic Operation

Vtunl is a dynamically loadable kernel module for the Solaris 2.x operating system from SUN Microsystems. The networking code in Solaris is based on the System V STREAMS framework and protocols belonging to the TCP/IP protocol suite, e.g. ARP, IP, UDP and TCP, are implemented in neatly separated modules. Figure 1 shows the basic STREAMS configuration for Solaris. Several instances of the same module may exist in the system simultaneously, e.g. all four instances of IP in Figure 1 share the same code. This is emphasized by enclosing multiple instances of the same module in a light gray rectangle. Note that IP is configured into the kernel both as a device and a module.

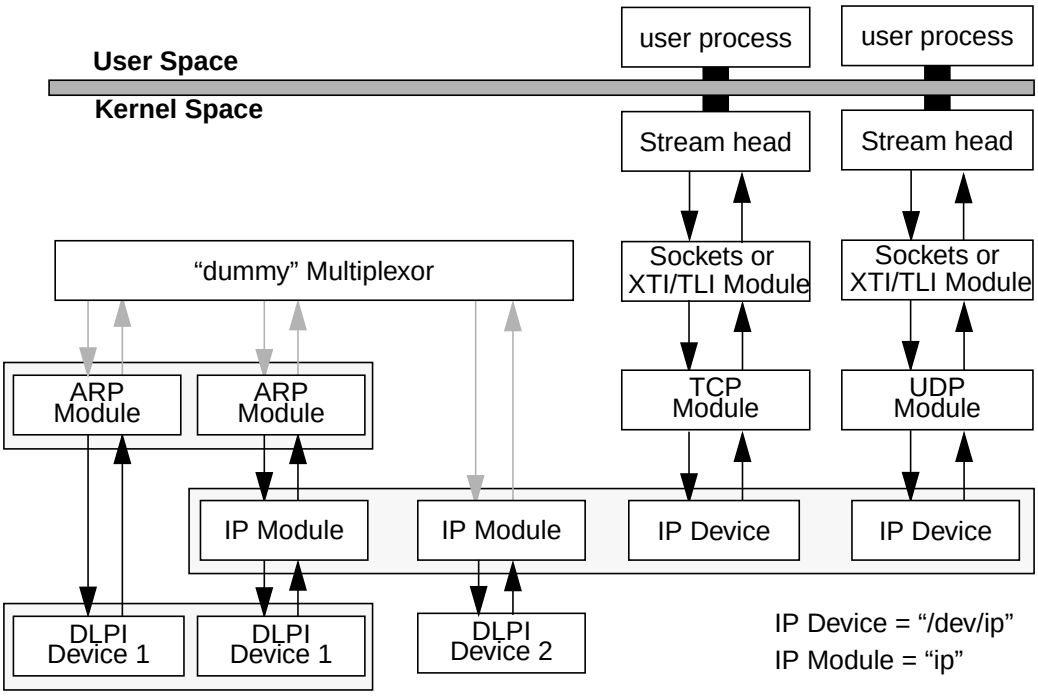

Figure 1 STREAMS configuration for Solaris TCP/IP.

A vtunl module plumbed between an instance of IP as a driver (`/dev/ip`) and another instance of IP as a module (`ip`) can perform both tunneling (*a.k.a.* encapsulation) and detunneling (*a.k.a.* decapsulation). This is shown in Figure 2(a). In the encapsulating mode, it receives a datagram from the IP module (after the routing code decides to send the datagram through the tunnel), inserts an encapsulation header, and passes the new datagram to the IP driver for rerouting. Vtunl registers with `/dev/ip` to receive a copy of all encapsulated packets (e.g. datagrams with the IP protocol field set to IPPROTO_ENCAP). It strips off the outer IP header and passes the newly exposed datagram to `ip` for rerouting. To the IP module above, vtunl appears as a data link device with a DLPI interface. To the IP driver below, vtunl appears as a STREAMS module. Like IP, vtunl exhibits dual personality acting

both as a STREAMS driver and module. By I_PLINKing the vtunl stream under UDP, this plumbing can be made to outlive the user process that created it.

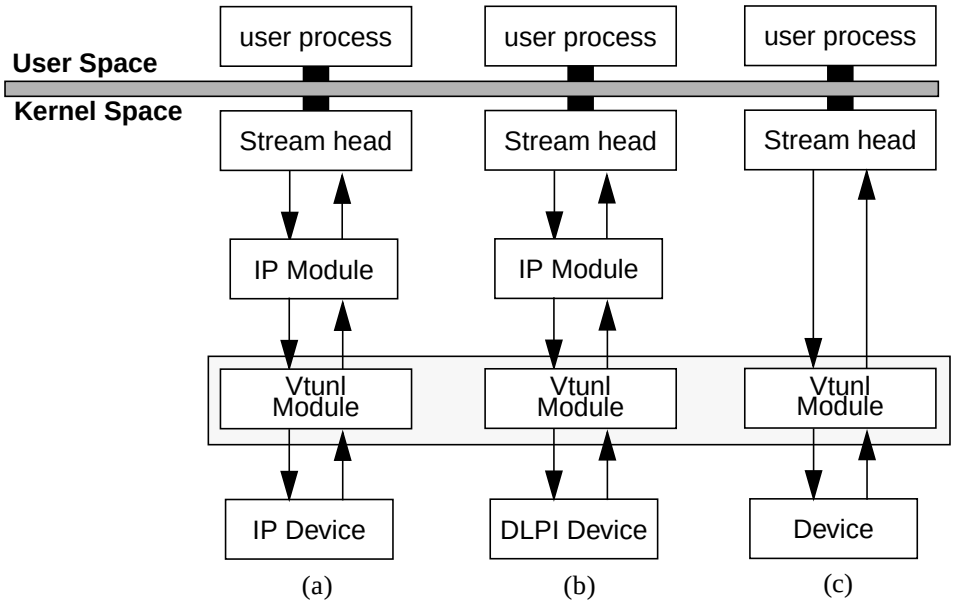

Figure 2 Three different types of STREAMS involving a Vtunl module.

The plumbing in Figure 2(a) creates a point-to-point interface. Assuming no other IP Devices are linked under an IP Module, this interface gets the name “ip0”. Since ip0 is a point-to-point interface, it can be configured with an address that may already be assigned to another interface, e.g. le0. This address sharing is desirable so that packets originating at the encapsulating host and passing through ip0 acquire a source address belonging to one of its “regular” interfaces.

Datagrams headed to specific destinations can be forced to undergo encapsulation by setting appropriate routing table entries. The following example clarifies this further. Suppose ip0 is configured as a point-to-point link between nobel1 and dummy (using an equivalent of the command **ifconfig ip0 nobel1 dummy up**). To force packets meant for cali through vtunl, one can add a new route with the command **route add cali dummy 1.** At a later time, one may use the command **route delete cali dummy** to stop vtunl from receiving

packets destined to cali. Table 1 shows how the routing table at nobel1 may appear after a route is added for cali through dummy. In this table, mpk15-net is the network connected to nobel1 and rmpk15 is the default router.

Table 1 Sample routing table entries at encapsulating host nobel1.

Destination	Mask	Gateway	Device
127.0.0.1	255.255.255.255	127.0.0.1	lo0
cali	255.255.255.255	dummy	
dummy	255.255.255.255	nobel1	ip0
mpk15-net	255.255.255.0	nobel1	le0
224.0.0.0	240.0.0.0	nobel1	le0
default	0.0.0.0	rmpk15	

Unfortunately, this mechanism is insufficient if encapsulation of broadcast or multicast packets is also desired since these packets bypass IP routing table lookups. For example, even if a route is created for 224.159.3.20 through dummy, packets sent to that multicast address shall not be routed through vtunl. This situation can be countered by the plumbing shown in Figure 2(b). A vtunl module plumbed between a DLPI device and an IP module can subscribe to all multicast and broadcast packets seen by the device; and encapsulate them on their way up before handing them off to IP for rerouting.

Vtunl has the ability to tunnel packets destined to a specific IP address (or target) to multiple exit points. The number of tunnel exits and their addresses are user-configurable. Each target for which encapsulation service is provided is represented by the `encap_target` data structure and each tunnel exit is represented by the `t_exit` structure. The important fields in these structures are shown below:

```
struct encap_target {
    u32         encap_target_addr; /* target's IPv4 address */
    short int    exit_count;  /* no. of exits for this target */
    u16         encap_target_encapflags[MAXEXITCOUNT];
                /* these flags control encapsulation type */
    struct t_exit *encap_target_exit[MAXEXITCOUNT];
}
```

```
struct t_exit {
    u32   t_exit_addr; /* IPv4 address of the tunnel exit */
    int   t_exit_ref_count;  /* no. of targets using this exit */
    u16   t_exit_pmtu;  /* current tunnel MTU */
    u16   t_exit_maxpmtu; /* current bound on Path MTU */
    u16   t_exit_state; /* other softstate information */
}
```

These structures are maintained in separate hash tables (`encap_table` and `texit_table`, respectively) hashed on their 32-bit address fields. The maximum number of tunnel exits that may be simultaneously associated with a target is MAXEXITCOUNT. When different targets request a common tunnel exit for encapsulation, they share the latter's `t_exit` structure. If vtunl receives a packet for a destination that does not have a `encap_target` structure, encapsulation is denied and the packet dropped.

Figure 2 explains the relationship between these data structures. Part (a) shows four hosts MH1 through MH4 that have requested encapsulating service. Datagrams for MH1 and MH4 are to be sent to FA2 after encapsulation. Datagrams for MH2 are to be tunneled to FA3 and those for MH3 are to be tunneled to both FA1 and FA2. Part (b) shows the corresponding

encapsulation related data structures. The figure assumes that the 32-bit addresses for MH3 and MH4 hash to the same value and the same is true for FA1 and FA2.

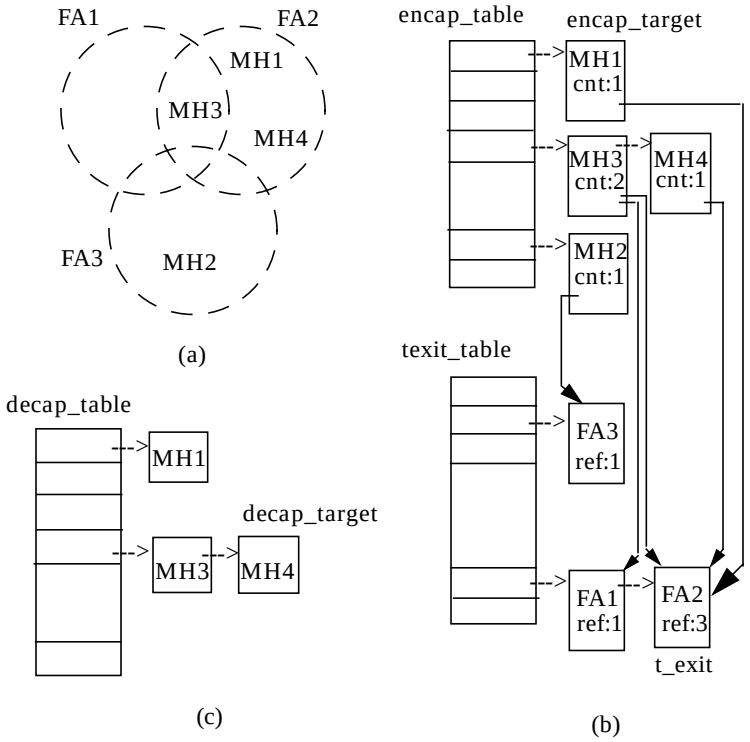

Figure 3 Data structures used in vtunl's internal configuration.

A target can specify different types of tunneling for each of its exit points. Tunneling behavior is controlled by the encapsulation flags stored in the `encap_target` structure, e.g. one may specify IPIP encapsulation or Minimal encapsulation. Currently, only IPIP tunneling is supported but the flag field allows additional functionality to be incorporated seamlessly. Flags may even be used to identify which protocol/program, e.g. Mobile-IP, *mrouted* created the entry -- this could be used to prevent one program from flushing entries created by another.

Instead of providing decapsulation service indiscriminately, vtunl can be configured to serve only selected hosts. Vtunl maintains a hash table called `decap_table` for this purpose. When selective decapsulation is turned on, only hosts that have a `decap_target` structure in this table are offered decapsulation service. Figure 1 (c) shows this hash table for FA2.

Under Solaris, any module subscribed to IP protocol *x* (other than IPPROTO_UDP and IPPROTO_TCP) automatically receives a copy of all ICMP messages generated in response to protocol *x* datagrams. Vtunl modules subscribed to encapsulating protocols, e.g. IPPROTO_ENCAP, use these ICMP messages to participate in tunnel soft-state management (as outlined in RFC 1933, “IPv6 Transition Mechanisms”). The vtunl code ensures that at most one instance is subscribed to IPPROTO_ENCAP at any given time.

Note – The vtunl driver is able to process only those packets that are sent through ip0, e.g. if the second routing entry (shown in Table 1) is deleted, the presence or absence of a `encap_target` structure for cali in vtunl’s internal configuration becomes irrelevant.

---

3. Configuration Notes

There are two classes of vtunl-specific ioctl commands depending on the data structures they affect. Commands VTUNL_SETCONF_TRANSPORT, VTUNL_SETCONF_DLSUSER and VTUNL_GETCONF operate on instance-specific data structures; only the vtunl instance in the stream targeted by the ioctl is affected. Other commands affect global data structures shared across all vtunl instances.

All vtunl commands must be issued using the I_STR ioctl (see `streamio(7)`) call. The call must be made on a stream in which vtunl appears directly underneath the stream head (see Figure 2(c)). Doing so prevents an intervening module (e.g. IP) from misinterpreting vtunl-specific commands. For example, a VTUNL_GETCONF ioctl for the vtunl instance shown in Figure 2(a) must be issued *before* the IP module is pushed.

Almost all commands use a `vtunl_req` structure as argument. This structure is defined as a union and has several fields including the following:

```
u32     vr_target_addr;
u32     vr_t_exit_addr;
```

```
u16     vr_encap_flags;
u16     vr_decap_flags;
u32     vr_address;
u32     vr_addr_mask;
u_char  vr_ppa;
u_char  vr_ip_protocol;
u16     vr_mtu_bound;
```

The first two specify 32-bit IPv4 addresses and the next two specify flags that affect the encapsulation / decapsulation service provided. The lower byte of the encapsulation flags could be the Mobile-IP registration flags S,B,D,M,G,V,x,x respectively. For now, only the S bit is supported. If the S bit is set, prior mobility bindings are retained. The `vr_mtu_bound` field is used by the VTUNL_MAXPMTU_{S,G}ET commands. The remaining fields are used for the two VTUNL_SETCONF commands.

Table 2 Instance-specific ioctls for the Vtunl interface.

Command	Parameters	Description
VTUNL_SETCONF_TRANSPORT	vr_address vr_ip_protocol	Configure a vtunl instance of the type shown in Fig. 2(a). vr_ip_protocol is the protocol subscribed to, e.g. IPIP.
VTUNL_SETCONF_DLSUSER	vr_address vr_addr_mask vr_ppa	Configure a vtunl instance of the type shown in Fig. 2(b), e.g. for le0, vr_ppa is 0.
VTUNL_GETCONF	vtunl_conf	Get information about a specific vtunl instance.

Table 2 describes the interface-specific ioctl commands defined for vtunl. The VTUNL_GETCONF command uses a vtunl_conf structure with the following fields:

```
#define MAXMNAMESZ  11
struct vtunl_conf {
  int    type;        /* vtunl instance type */
  u_char ppa;         /* ppa of DLS provider */
  u_char ip_protocol; /* ip protocol subscribed to */
  u32    address;     /* IP addr of DLS provider or
                         tunnel src */
  u32    addr_mask;   /* netmask assigned to DLS provider */
  ulong  last_req;    /* last request sent to ip or DLS
```

```
                              provider */
  ulong  last_resp;   /* last response from ip or DLS
                              provider */
  ulong  state;       /* as determined by msg with ip or
                              or the DLS provider below */
  char   modname[MAXMNAMESZ+1];
       /* either "ip" or name of DLS provider, e.g. "le" */
};
```

Table 3 lists the global ioctl commands, their parameters and a brief description. The VTUNL_ENCAP_OK command is useful in determining

Table 3 Global ioctl commands for the vtunl interface

Command	Parameters	Description
VTUNL_ENCAP_ADD	vr_target_addr vr_t_exit_addr vr_encap_flags	Provide encapsulation for target_addr using vr_t_exit_addr as exit, encapsulation flags SBDMGV are specified in the lower byte of target_encap_flags.
VTUNL_ENCAP_REM	vr_target_addr vr_t_exit_addr	Cancel encapsulation service. If vr_target_addr is 0, cancel all encapsulation services, if t_exit_addr is 0, stop encapsulating for specified target else stop encapsulating only for the given combination.
VTUNL_ENCAP_OK	vr_target_addr	Returns 0 if encapsulation service is offered to specified address, ENXIO otherwise.
VTUNL_ENCAP_SIZE	integer	Returns the minimum buffer size required for a successful VTUNL_ENCAP_GET.
VTUNL_ENCAP_GET	character buffer	Dumps encapsulation configuration into buffer.
VTUNL_DECAP_ADD	vr_target_addr vr_decap_flags	Provide decapsulation if specified address appears in the inner datagram as indicated by the flags. Possible values of the flags are DECAP_IF_{SRC | DST | EITHER}. A zero target enables decapsulation of ALL datagrams.
VTUNL_DECAP_REM	vr_target_addr	Cancel decapsulation for specified address.
VTUNL_DECAP_OK	vr_target_addr	Returns 0 if decapsulation is provided for specified address, ENXIO otherwise.

Table 3 Global ioctl commands for the vtunl interface

Command	Parameters	Description
VTUNL_DECAP_SIZE	integer	Returns the minimum buffer size required for a successful VTUNL_DECAP_GET.
VTUNL_DECAP_GET	character buffer	Dumps decapsulation configuration into buffer.
VTUNL_DEBUG_SET	integer (0= quiet, 3 = most verbose)	Set debug level.
VTUNL_DEBUG_GET	integer	Get debug level.
VTUNL_MAXPMTU_SET	vr_address, vr_mtu_bound	Set an upper bound on the Path MTU associated with the tunnel exit specified in vr_address.
VTUNL_MAXPMTU_GET	vr_address, vr_mtu_bound	Get the PMTU upper bound associated with given address.

whether a routing table entry through ip0 is required for vr_target_addr on the encapsulating host.

The buffer returned by a successful VTUNL_ENCAP_GET contains the following: number of bytes filled (4 bytes), number of encapsulation targets (4 bytes), for each target, its address (4 bytes), number of tunnel exits (4 bytes) and the address of each exit (4 bytes), number of tunnel exits (4 bytes), for each exit, its address (4 bytes), current PMTU estimate (2 bytes), maximum PMTU (2 bytes) and state (2 bytes).

The buffer returned by a successful VTUNL_DECAP_GET contains the following: number of bytes filled (4 bytes), value of *vtunldecapall* (4 bytes) which currently is used as a boolean variable (TRUE only if *all* datagrams are decapsulated), number of decapsulation targets (4 bytes), and for each target, its address (4 bytes) and flags (2 bytes).

Note – All IPv4 addresses must be passed in network byte order. For the VTUNL_ENCAP_ADD and VTUNL_ENCAP_REM commands, vr_target_addr may be set to DEFAULT_ADDR (defined in vtunl.h). An entry created with this target address is used to encapsulate packets for all those destinations that do not have their own entries in the encap_table.

4. Users' Guide

This section describes the vtunl installation and testing procedure which requires superuser privileges (the process has been tested on Solaris 2.5.1 and Solaris 2.6). While we have been using vtunl for over six months, it should still be considered an experimental module and users should backup all significant data before using it. We also recommend backing up the original kernel image using the command **cp -r /platform/<arch>/kernel /platform/<arch>/kernel.orig** (**<arch>** is sun4m for a SparcStation2, sun4u for Ultra1 etc. If you are unsure of what to use for **<arch>**, use the output of the **uname -i** command). If the working kernel is ever corrupted, the original can be booted by specifying **boot kernel.orig/unix** at the boot prompt.

To install vtunl, copy the **vtunl** and **vtunl.conf** files to **/platform/<arch>/kernel/drv** and execute **add_drv vtunl.** The `add_drv (1M)` command is used to add a new device driver to the system. Use **modinfo | grep vtunl** to verify that vtunl has been successfully loaded. The output should indicate two entries for vtunl -- one as a module and the other as a driver. The command **add_drv vtunl** needs to be executed afresh each time the machine is booted. The `rem_drv (1M)` command can be used to remove a device driver from the system.

The program **vtunlconf** can be used to test vtunl. It creates multiple streams containing the vtunl module. One stream plumbs vtunl between `/dev/ip` and `ip`, others plumb vtunl between DLPI devices and `ip` and still another plumbs vtunl directly underneath the stream head. This last stream is used for issuing global vtunl ioctls. The vtunlconf program (along with **/sbin/route**) allows a user to control vtunl's behavior interactively. The vtunlconf program offers on-line help, and a UNIX style manpage is also available. Tunneled packets produced by vtunl can be monitored by programs such as **snoop** or **tcpdump** (encapsulated datagrams have the protocol field in their outermost IP header set to 4).

5. Possible Enhancements

Supporting registration lifetimes within vtunl can reduce the amount of state maintained by user level programs like the Mobile-IP agent daemon. The STREAMS framework has built-in support for asynchronous event notification, e.g. a user process can issue the I_SETSIG ioctl call on a stream to receive a SIGPOLL signal whenever a M_PCPROTO message appears at the stream head. The vtunl code can be altered to send such a message to one or more

connected streams when a registration expires. It may be desirable to send these notifications on just one of the streams involving the vtunl. This special stream may be identified by sending a particular ioctl command (e.g. VTUNL_NOTIFY) through it. As part of processing this command, the vtunl code can mark that stream for special handling, e.g. it can store the queue argument passed to its ioctl handler and use that as an argument to `qreply()` for sending the M_PCPROTO message. If registration lifetimes are supported within vtunl, timer management should be based on an event list.

The DEFAULT_ADDR concept used in VTUNL_ENCAP_{ADD, REM} can be generalized by associating netmasks with each `encap_target` address. Netmasks provide a convenient mechanism for specifying multiple IP addresses that need to be processed in the same manner.