

Virtual Address Resolution in Supernets: VARP

Distribution of Memo

Sun Proprietary/Confidential: Internal Use and Need to Know Only.

The goal of this memo is

- to document system interfaces,
- to document what has been designed and implemented, and
- to guide the implementation effort of this component.

Abstract

This document specifies design and implementation details of the address resolution portion for the Virtual Networking Project, called Viper. For the vision, and the overall architecture of Viper see other documentation ([1]).

Introduction

Each supernet contains one or more communication channels. A channel is a communication abstraction that fixes an association between supernet members through a shared key. Communication packets that live within supernets are always associated with exactly one channel and they are encapsulated in datagrams that can be interpreted and transmitted on the global IP network. This encapsulation provides - on a per group basis - security services, such as authenticity, integrity, and possibly confidentiality protection. Encapsulated packets cannot be interpreted outside the supernet channel they belong to, because the knowledge of the used key is restricted to group members.

These design choices imply the necessity for an address mapping between IP addresses that are used in supernet instances and IP addresses on the global IP internetwork. To identify the different types of addresses, we are using the following terminology and abbreviations:

- Sn - Supernet
- SnName - Supernet Name, e.g., swan.sun.com
- SnId - Supernet identifier, E.g., 0x0123456789abcdef0123, or 0x000000000000cafeface
- Ch - Channel
- ChId - Channel Identifier, E.g., 0x0123456789ab, or 0x000000004711
- Vaddr - IP address that has meaning only within a given supernet context, as identified by a SnName or SnId.

- RAddr - IP address of computer hosting one or multiple supernet nodes.
- Node - Nodes are identified by (SnId,Vaddr)
- Combo - Combos are identified by (SnId,ChId)
- BdNode - Bound Nodes are identified by (SnId,ChId,Vaddr)

One computer that is attached to the global IP network can participate in multiple supernets at the same time. Each instance is called a node and identified by the identifier tuple (SnId,Vaddr).

It is the purpose of the VARP service to maintain the mapping of tuples (SnId,Vaddr) to their associated (RAddr) and to make this address resolution service available to authorized clients. Any node that is part of a supernet, is authorized to request mappings of the form (SnId,Vaddr) from the VARP server for a given supernet.

Such a service can be achieved by various different mechanisms. For the first prototype in Viper we have settled on the architecture described below.

Architecture for Virtual Address Resolution

The VARP service architecture is client-server based. The VARP service belongs to the Supernet *skin*, i.e., communication traffic generated between VARP service components is not part of Supernet traffic. Therefore, it may be protected by different security services than those offered by the Viper architecture. The VARP service requires integrity and authenticity protection. This protection may be achieved through the use of SSL tunnels between all VARP participants. The VARP protocol then uses UDP on the real IP network for client-server communication.

The IP address for the VARP server for each Supernet needs to be known to each VARP client of that Supernet. In this version of Viper, the local VARP daemon is informed during the first attach action to a Supernet of its VARP server address. The group manager (aka *sasd*) can configure a VARP daemon to act as the VARP server for a single or a set of Supernets. The VARP protocol operates over a well known port (VARPD_VARP_PORT).

Each computer contains two VARP components:

1. VARP address cache (called *varpdb*)
2. VARP protocol engine (called *varpd*)

Refer to Figure 1 for an illustration of the system components and the interfaces between them and the rest of the world. The shaded area highlights the components that belong to the VARP system proper.

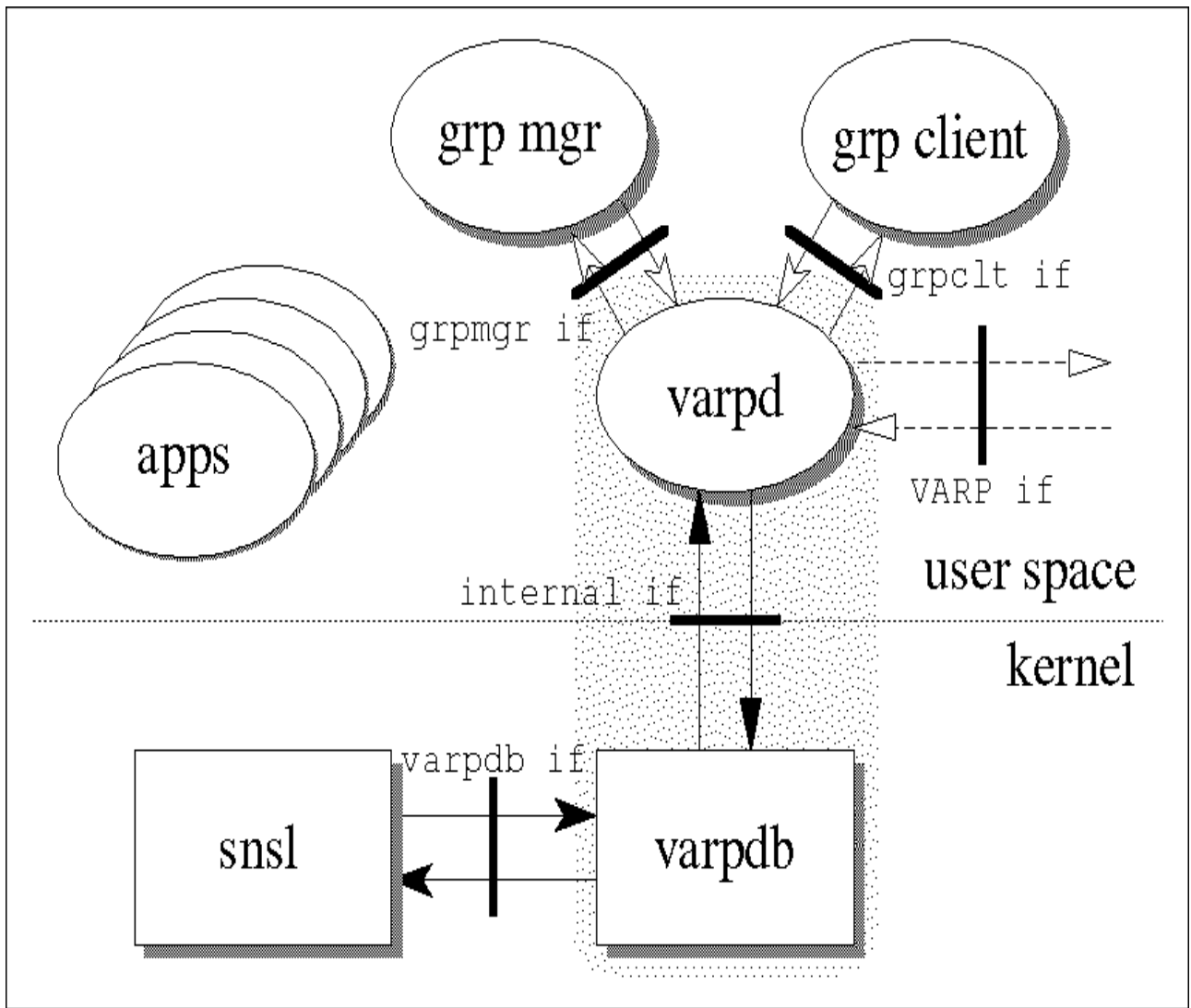


Figure 1: VARP system components

The address cache is located in the kernel and provides the typical database services for storing (SnId,Vaddr) to (RAddr) mappings. It also provides one external interface, a function call interface for address lookups (all mentioned interfaces are defined in subsequent sections):

- Varpdb interface (Fig. 1: varpdb if);
between the SNSL module and the varpdb module.

It exposes the following functionality: callback registration, monitor callback registration, and address resolution of virtual addresses (SnId,Vaddr) to real addresses (RAddr). The result of such a resolution request is immediately returned if it is found in the varpdb cache. If it is not found, the local VARP protocol engine is contacted for resolution through the VARP server. The result is communicated back to the SNSL module in an asynchronous fashion through the registered callback function.

The VARP protocol engine varpd is located in the user space. It provides VARP client and VARP server functionality. It provides client functionality for all Supernets a computer is connected to. The address of the VARP server for each connected Supernet must be configured (i.e., configured

through a command by the group client (e.g., the `snattach` program) - see below). The VARP protocol engine can be configured to become the server for one or several Supernets, while at the same time providing VARP client services. Thus, there is exactly one VARP protocol engine process (i.e., `varpd`) per computer.

The VARP protocol engine and the VARP address cache work together to provide the VARP service. To achieve an efficient address lookup, the address cache should be located in the kernel, *close* to the module that requests addresses for all datagrams leaving the computer over the network. The protocol engine, however, does not need to be located in the user space. Ideally it would be located *close* to the address cache because it is the entity that maintains the cache entries. Furthermore, VARP packets could be sent and received by the protocol engine without multiple kernel/user space traversals, similar to existing ARP and ATMARP implementations. We still chose to implement the VARP protocol engine in the user space for the first prototype to simplify system development. It allows us to use SSL protected connections between VARP clients and servers for securing VARP messages, and it does not require us to demultiplex VARP messages in the kernel and hand them off to the VARP protocol engine. The migration of the `varpd` functionality into the kernel is expected to be done at a later time.

The VARP protocol engine exposes three external interfaces:

- Virtual address resolution protocol (VARP) messages (Fig. 1: `VARP if`); between peer VARP protocol engines. This is the client-server VARP protocol.
- Group manager interface (Fig. 1: `grpMgr if`); between the Supernet group manager (i.e., `sasd`) and the VARP protocol engine.
- Group client interface (Fig. 1: `grpclt if`); between Supernet group clients (i.e., `snattach`) and the VARP protocol engine.

The first interface provides the specification for message formats for the VARP client-server protocol between VARP clients and servers. The messages are used for requesting (`SnId,Vaddr`) to (`RAddr`) address resolutions and for communicating the (positive or negative) result of lookups.

The second interface enables each Supernet group manager to maintain the database entries of the VARP server for their respective Supernet.

The third interface is necessary to inform a local `varpd` of the VARP server address of Supernets used on this machine.

We have decided on a clear text interface for inserting and removing database entries, as well as various administrative commands (e.g., initializing data structures, configuring client and server parameters.) Clear text interfaces allow for faster prototyping, because messages crossing the interface are easier understood and many standard programs can be used for simulating or processing interface I/O.

The VARP protocol engine and the VARP address cache communicate with each other over an internal interface:

- Internal interface (Fig. 1: `internal if`); between `varpd` and `varpdb`.

Although this interface is not exposed externally, we include its specification here.

Example

Datagram for previously unseen Supernet address

Refer to Figure 2 for an illustration of the first example.

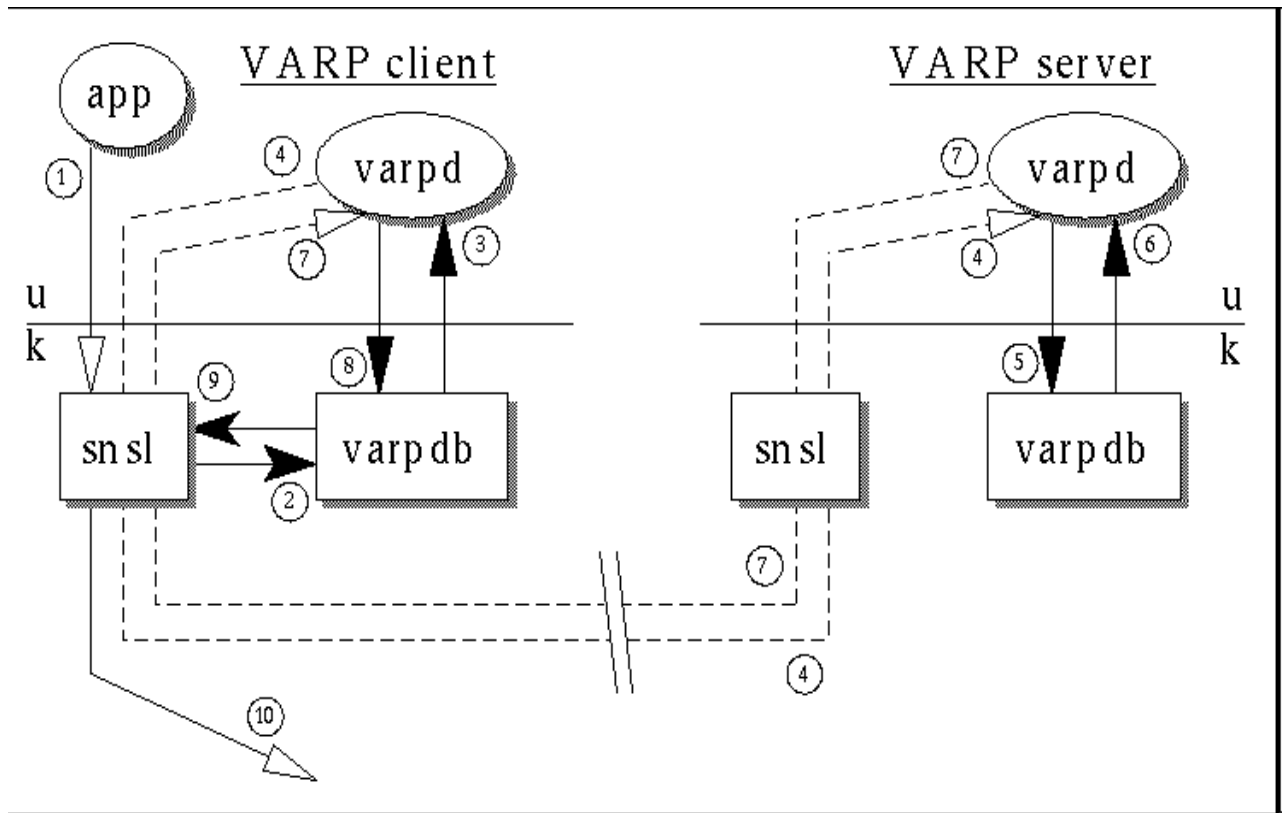


Figure 2: Example: App sends a datagram to previously unseen Supernet address.

This example illustrates the process that occurs when an application (app) first sends a datagram to a Supernet address (SnId,Vaddr) that has not previously been contacted from that machine (at least not within the timeout period that varpdb cache mappings have). The computer illustrated on the left runs the application app and originates the datagram. The computer illustrated on the right provides the VARP service for the Supernet snid. We assume that the Supernet node address (SnId,Vaddr) is properly registered in the VARP service, and that the VARP client code on the computer illustrated on the left is configured to contact its VARP server.

In step (1), the application originates traffic that arrives at the SNSL module. Among other tasks, the SNSL module needs to determine the real address (in SNSL terms, the *tunnel address*) for the encapsulated datagram. It calls function `varpdb_lookup(&(snid,vaddr,0.0.0.0), &status)` in step (2) to query the local VARP address resolution cache if the address is known. Since this is the first traffic to this address, the raddr associated with (snid,vaddr) is not known and `varpdb_lookup()` returns a status code `status=VARPDB_MISS`. The SNSL module now keeps the partially completed datagram enqueued, waiting for the varpdb module to call its callback function with the requested raddr filled in. If the reply does not arrive at the SNSL module within a certain period of time, the datagram is discarded.

Subsequently, the varpdb module contacts its varpd protocol engine with the resolution request in step (3). Varpd creates a `VARP_REQUEST` packet for the address and sends it via UDP to the VARP server for Supernet snid (step (4)). Note that this message is a *skin* message and does not require another SNSL-varpdb interaction, because it is not sent within supernet context. However, the message still passes through the SNSL module and may make use of a preestablished SNSL tunnel between the VARP client and server computers, or of SSL protected communications between the varpd peers.

The VARP server receives the `VARP_REQUEST` message and calls its own varpdb module to look up the Supernet address (snid,vaddr) in its authoritative database (step (5)). Varpdb returns the desired (raddr) (step (6)) and the VARP server creates a `VARP_REPLY` protocol message that is subsequently sent to the VARP client (step (7)).

The VARP client receives the `VARP_REPLY` and forwards it to its varpdb module (step (8)) where the new address mapping is recorded in the local varp cache. Furthermore, the varpdb module now calls SNSL's callback function with a status code of `VARPDB_HIT` and the completed mapping (snid,Vaddr) to (raddr) (step (9)). SNSL now enters the destination tunnel address (raddr) into the waiting datagram. As soon as the datagram is ready to be sent out, it is forwarded to the network output queue and sent to its destination (step (10)).

Datagram for previously seen Supernet address

Figure 3 illustrates the common case for which our design is optimized: An application sends a datagram to a destination address for which the raddr is already cached in the varpdb module.

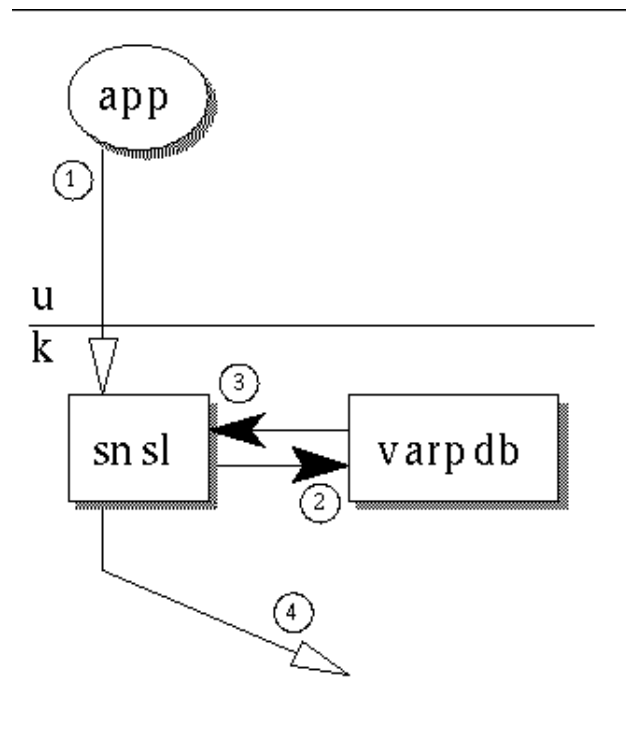


Figure 3: Example: App sends a datagram to cached Supernet address.

As in the previous example, in step (1), the application originates traffic that arrives at the SNSL module. SNSL calls the varpdb module for an address resolution (step (2)). Per assumption, the cache in varpdb contains the answer, and varpdb returns the desired raddr (step (3)). The packet is forwarded to its destination as soon as the SNSL module has completed all other per packet processing (step (4)).

Interfaces and Operational Requirements

Interface varpdb if between SNSL and varpdb

Given the following data types:

```
vid_t          for Supernet ID's

struct in_addr for Vaddr's and raddr's (as in <netinet/in.h>)

vstatus_t     for return codes for the lookup functions below
  VARPDB_NOTASSIGNED = 0x00 - initial value - not used as code.
                        Use of this value as a result code
                        is a bug.
  VARPDB_HIT         = 0x01 - address found in local cache
  VARPDB_MISS       = 0x02 - address not found in local
                        cache
  VARPDB_NOTKNOWN   = 0x03 - address not registered in varp database

vnode_t       for recording related SupernetIds, Vaddrs, and Raddrs

addrset_t     for communicating vnode_ts, caddrs, and
              ttls between the varpdb and user level apps,
              such as varpd.
```

Function call interface specification (in C syntax).

```
void varpdb_register    (void (*callback)(vnode_t  *vnodep,
                                         vstatus_t *statusp));

void varpdb_mon_register(void (*callback)(vnode_t  *vnodep,
                                         vstatus_t *statusp));

void varpdb_lookup      (vnode_t  *vnodep,
                       vstatus_t *statusp);
```

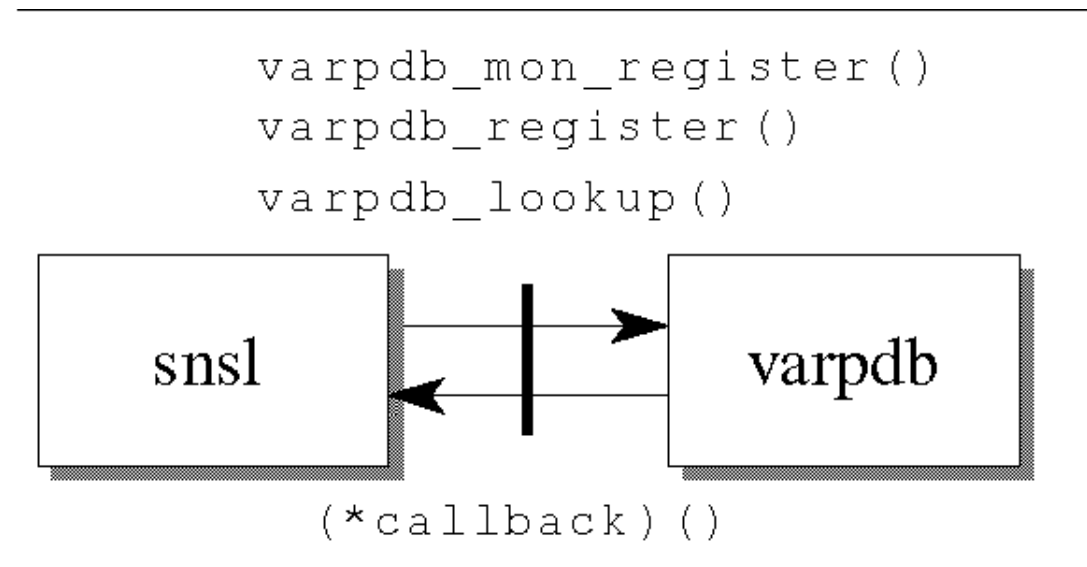


Figure 4: Interface varpdb if

Operational requirements: varpdb

When the varpdb kernel module is first installed (or loaded) in the kernel, its data structures are automatically initialized. At this point, the SNSL module is the only user of the varpdb module. The current interface supports two callback functions for the varpdb module. One is for lookups from the SNSL layer, the other for a monitoring application that can display callback events. If in the future varpdb is required to be able to serve a set of users, such as SNSL, this interface may need to change. If multiple users use different callback functions, a demultiplexing step needs to be introduced. If the demultiplexer is part of varpdb, the interface needs to change, if the demultiplexer is outside of varpdb, the interface can stay the same with the callback pointing to the demultiplexing code.

The main function of varpdb is to provide address mappings from (snid,Vaddr) to (raddr) addresses. Users of varpdb (currently only the SNSL module) use the `varpdb_lookup()` function for requesting a mapping for (snid,Vaddr). If the mapping is present in varpdb's cache, field raddr is filled with the found (raddr) IP address and the status code is set to `VARPDB_HIT`. If the mapping is not present in the cache, varpdb in turn requests an address resolution from the VARP protocol engine over the internal interface. This process can take some time and cannot block the varpdb user module. Therefore, `varpdb_lookup()` returns immediately with a status code of `VARPDB_MISS`. Argument raddr remains undefined.

Note, that varpdb caches the fact that a lookup failed for a particular address for a short period of time (`VARPDB_TTL_INITIAL_LOOKUP` seconds). Within that time only a single lookup request is forwarded over the internal interface to the protocol engine. This feature is present to avoid a `VARP_REQUEST` (and `VARP_REPLY`) storm from a VARP client to a VARP server, if the client has scheduled many datagrams to be sent to the same destination for which the address is unknown to the VARP client. While the `VARP_REPLY` for a requested Supernet address is pending and the initial lookup delay has not expired, calls to `varpdb_lookup()` return immediately with a status code of `VARPDB_MISS`. Result parameter (field) raddr remains undefined.

At a later time the VARP protocol engine (varpd) communicates the result of the lookup to the varpdb module. Two different answers are possible:

1. the mapping was found: `VARPDB_HIT`
 - `varpdb` enters the new mapping into its cache (or updates an existing one).
 - `varpdb` uses the `callback()` function (if registered) to communicate the result back to its user. The `raddr` argument is filled in and the status code is set to `VARPDB_HIT`.
2. the mapping cannot be found, because it is unknown to the supernet's `varpd` server, or because there is no server configured for the supernet: `VARPDB_MISS`
 - `varpdb` uses the `callback()` function to communicate this result back to its user. The `raddr` argument remains undefined and the status code is set to `VARPDB_MISS`.

Note, that there is a chance that the `varpdb` module never receives an answer back for the request it sent to its protocol engine (e.g., if the `VARP_REQUEST` over UDP is lost). The `varpdb` module therefore does not provide any guarantees that the callback function will be called at all! The VARP protocol provides a *best effort* service.

The callback function is registered in `varpdb` via the `varpdb_register()` function. It can be changed by simply registering a new function. It is unregistered by calling `varpdb_unregister()` with a NULL function pointer.

Interface `grp_mgr` if between Supernet group manager and `varpd`

Command interface between supernet group manager and `varpd`:

```
varpd_init()
varpd_add() <snid> <vaddr> <raddr> [<t1>]
varpd_remove() <snid> <vaddr>
varpd_server_set() <snid> [<t1>]
varpd_server_clear() <snid>
```

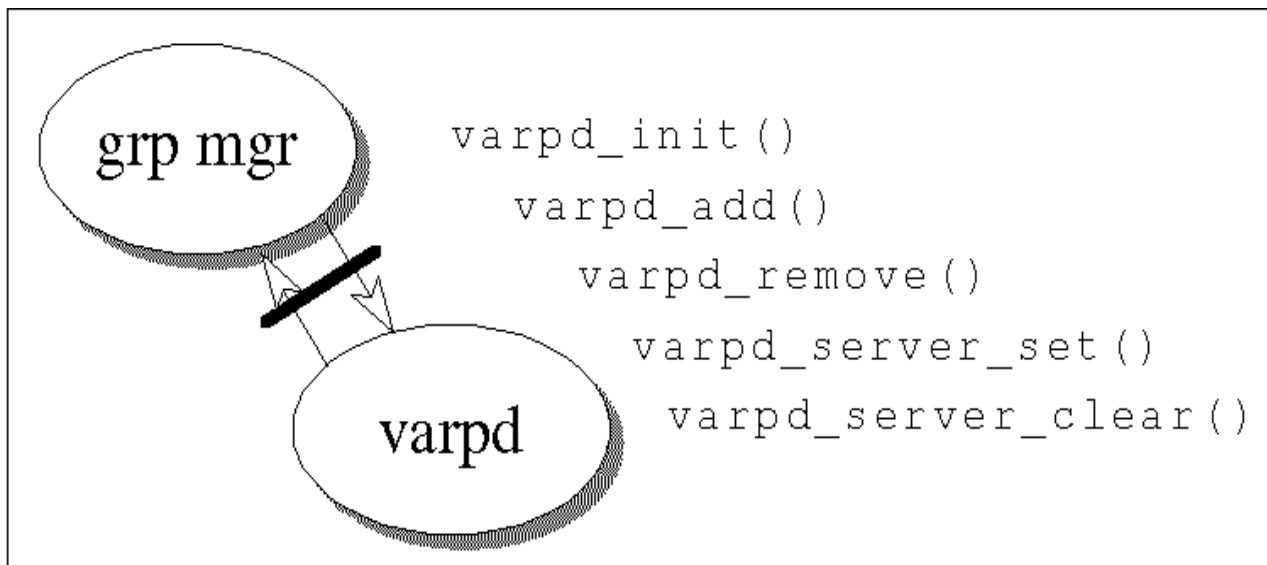


Figure 5(a): Interface `grp_mgr` if

Operational requirements: `varpd`

VARP server databases are maintained by the Supernet group manager. The Supernet group manager is the only entity in a Supernet that can add or remove (`snid,vaddr`) to (`raddr`) mappings in

the VARP server database. Varpd provides a clear text interface to the Supernet group manager through a socket. The TCP port number for the socket is `VARPD_GRP_MGR_PORT`. Commands and arguments must be separated by white space (<space> or <tab>). Each complete command must be terminated by line break or a semicolon. The maximal length of a command is limited to 255 characters.

- Command `varpd_init()`

The Supernet group manager can initialize the data structures of the varpd by issuing the command `varpd_init()`. When varpd starts, all data structures are initialized and it is not necessary to issue this command. It is provided in the case the group manager wishes to reset the varpd. This command also initializes all varpdb data structures (except for the callback function registration). The Supernet manager has no control over modifying the assignment of the callback function. When varpdb is first loaded it is already initialized.

- Command `varpd_add() <snid> <vaddr> <raddr> [<ttd>]`

The Supernet group manager can add new mappings into the database by issuing a `varpd_add()` command. The `varpd_add()` command takes three or four arguments, the <snid>, the <vaddr>, the <raddr>, and an optional <ttd>. If a mapping for (snid,vaddr) already exists in the database, its (raddr) entry is overwritten with the new value. The optional time-to-live value becomes the default ttl for the database entry. The command `varpd_add()` at the group manager interface translates into the command `VARPDB_OPCODE_INSERT` at the internal interface which causes the database transactions to happen. The default value for <ttd> is `VARPDB_TTL_ALWAYS`.

- Command `varpd_remove() <snid> <vaddr>`

The Supernet group manager removes mappings in the database by issuing a `varpd_remove()` command. The `varpd_remove()` command takes two arguments, the <snid> and the <vaddr>.

- Command `varpd_server_set() <snid> [<ttd>]`

Varpd can be configured to become the server for Supernet <snid>. The group manager has control over this configuration issue and executes it by issuing a `varpd_server_set()` command to the varpd. Great care has to be taken that exactly one varpd provides VARP service to an entire Supernet and that all VARP clients are configured to contact that one VARP server. The optional time-to-live value becomes the default ttl for mappings that are served by varpd for Supernet snid. The default value is `VARPDB_TTL_INITIAL_CLIENT`. This command can be used multiple times by the group manager for the same <snid> to change its default time-to-life.

- Command `varpd_server_clear() <snid>`

This command is used by the group manager to inform the varpd that it is no longer serving as the VARP server for Supernet <snid>.

At this time, none of these commands provide explicit feedback to the Supernet group manager. This may change in the future.

Interface `grpclt` if between Supernet group manager and `varpd`

Command interface between Supernet group client (i.e., `snattach`) and `varpd`:

```
varpd_client_set() <snid> <saddr>  
varpd_client_clear() <snid>
```

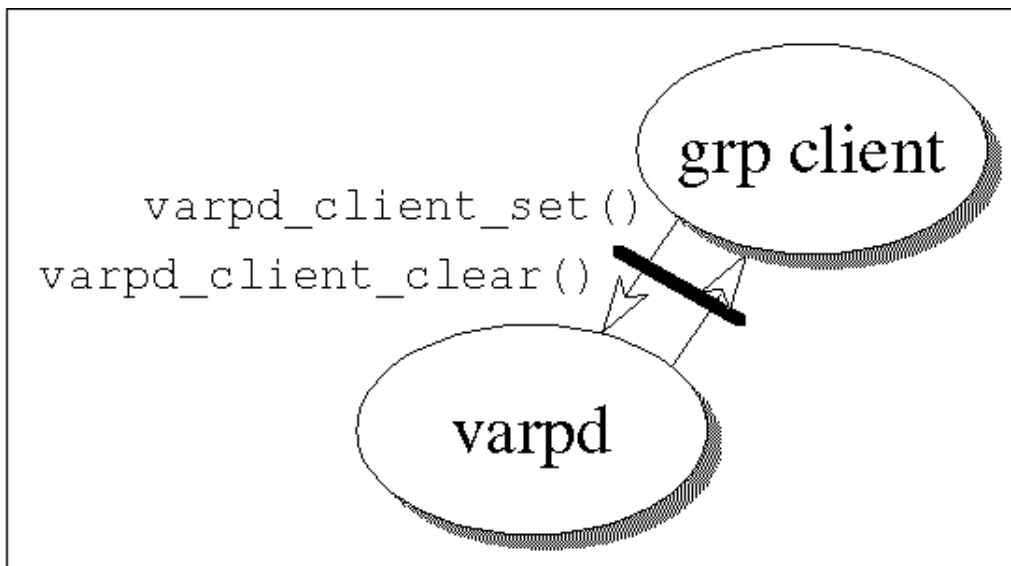


Figure 5(b): Interface `grpclt` if

Operational requirements: `varpd`

Each `varpd` that participates in a Supernet needs to be configured to contact its VARP server. `Varpd` provides an interface over TCP port `VARPD_GRP_CLT_PORT` to be informed of this configuration parameter. A single command is accepted over this interface, the `varpd_client_set()` command. Each connection to port `VARPD_GRP_CLT_PORT` can stay open at most `VARPD_TTL_CLT_PORT_ACCESS` seconds before it is closed. This feature protects `varpd` against an accidental denial of service by a legitimate client keeping the connection open too long.

- Command `varpd_client_set() <snid> <saddr>`

This command allows a group client to inform the `varpd` of the IP address (`<saddr>`) of the VARP server for Supernet `<snid>`. The `varpd` stores the information to be used for subsequent VARP requests to the VARP server for that Supernet. The command can be issued multiple times for the same `<snid>` to change the address of the VARP server.

- Command `varpd_client_clear() <snid> <saddr>`

This command allows a group client to inform the `varpd` that there is no longer a VARP server for the Supernet. This can be the case if the underlying host no longer participates in a given Supernet and has no longer a need to be configured with the VARP server address.

At this time, the commands does not provide explicit feedback to the client. This may change in the future.

Interface internal if between varpd and varpdb

This interface is implemented over a file interface via `ioctl()` calls. The file name is `VARPDB_DEVICENAME`.

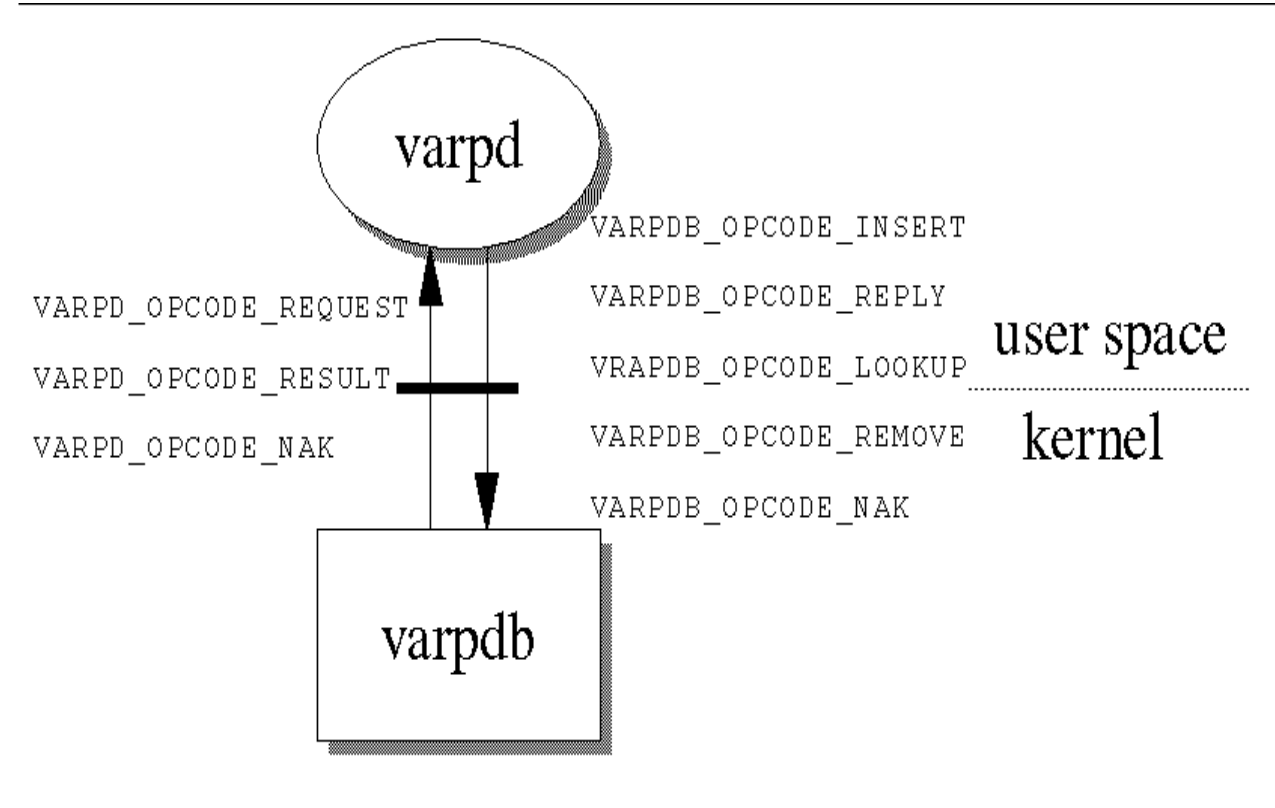


Figure 6: Interface internal if

Interface internal if between varpd to varpdb

```
#define VARPDB_DEVICENAME "/dev/varpdb0"  
  
#define VARPDB_IOCTLXRESET    _IO(VARPDB_IOC_MAGIC, 0)  
#define VARPDB_IOCTLXTIMSTART _IO(VARPDB_IOC_MAGIC, 1)  
#define VARPDB_IOCTLXTIMSTOP  _IO(VARPDB_IOC_MAGIC, 2)  
#define VARPDB_IOCTLWCMD      _IOW(VARPDB_IOC_MAGIC, 3, addrset_t)
```

Operational requirements: varpdb

There are four `ioctl`'s defined:

- `VARPDB_IOCTLXRESET`: resets the `varpdb` module and all its data structures.

When `varpdb` is first loaded it is already initialized. This `ioctl` is provided in case the group manager wished to reset the `varpdb` module.

- `VARPDB_IOCTLXTIMSTART`: starts the module's timer code that periodically updates the database entries. This command feature is implemented primarily for debugging purposes and will disappear in future releases.
- `VARPDB_IOCTLXTIMSTOP`: stops the module's timer code that periodically updates the database

entries. This command feature is implemented primarily for debugging purposes and will disappear in future releases.

- `VARPDB_IOCTLWCMD`: is used to exchange a more extensive command set between `varpdb` and user space applications, such as `varpd` or `varpmon`. The command opcodes take a pointer to a record data structure as argument. Not all commands use all fields of this record. The following paragraphs describe the command operation codes (opcodes) and their parameter usage.

List of `VARPDB_IOCTLWCMD` relevant data structures and opcodes:

```
typedef struct {
    vid_t      vid;
    struct in_addr vaddr;
    struct in_addr raddr;
} vnode_t;

typedef struct addrset {
    int      op_code;
    vnode_t  vnode;
    struct in_addr caddr;
    int      ttl;
} addrset_t;

#define VARPDB_OPCODE_INSERT      1
#define VARPDB_OPCODE_REPLY      2
#define VARPDB_OPCODE_LOOKUP     3
#define VARPDB_OPCODE_REMOVE     4
#define VARPDB_OPCODE_NAK       5

#define VARPDB_OPCODE_REQUEST    11
#define VARPDB_OPCODE_RESULT     12
#define VARPDB_OPCODE_NAK       13
```

Command opcodes are transmitted in the field `op_code` of the argument to the `VARPDB_IOCTLWCMD` ioctl. The remaining fields (`vnode.snid`, `vnode.vaddr`, `vnode.raddr`, `caddr`, and `ttl`) are used as required. We will abbreviate `vnode.snid` by `snid` (cf. `vaddr` and `raddr`).

- Command opcode `VARPDB_OPCODE_INSERT` `<snid>` `<vaddr>` `<raddr>` `<ttl>`

The command `varpd_add()` at the group manager interface, translates into the command opcode `VARPDB_OPCODE_INSERT()` at the internal interface. The only difference is that the `ttl` value is mandatory in this command. (See the description of command `varpd_add()` above.) If the value was not supplied as an argument to the `varpd_add()` command, the default `VARPDB_TTL_ALWAYS` is used.

- Command opcode `VARPDB_OPCODE_REMOVE` `<snid>` `<vaddr>`

The command `varpd_remove()` at the group manager interface, translates into the command opcode `VARPDB_OPCODE_REMOVE` at the internal interface. See the description of command `varpd_remove()` above.

- Command opcode `VARPDB_OPCODE_LOOKUP` `<snid>` `<vaddr>` `<caddr>`

This command allows the `varpd` to determine if the database maintained by `varpdb` contains the mapping (`snid,vaddr`). It requires the `VARPDB_OPCODE_LOOKUP()` and `varpd_result()`

(see below) functions to carry an additional argument: the requesting client's address (<caddr>). This argument enables the varpd to reply to the VARP client on which behalf the lookup happened, without having to maintain a list of outstanding requests to the database and a mapping to their respective requesters. In other words, the varpd client can remain stateless.

- Command opcode `VARPDB_OPCODE_REPLY()` <snid> <vaddr> <raddr> <t1>

This command opcode is issued by the varpd protocol engine in response to receiving a `VARP_REPLY` packet from a VARP server. The command causes varpdb to perform the same actions as for command opcode `VARPDB_OPCODE_INSERT`, plus to call the varpdb user's callback function, if registered.

- Command opcode `VARPDB_OPCODE_NAK` <snid> <vaddr>

This command opcode is issued by the varpd protocol engine in response to receiving a `VARP_NAK_NOTKNOWN` packet, a `VARP_NAK_NOAUTHORITY` packet, or when varpd received command opcode `VARPD_OPCODE_REQUEST` from the varpdb module for a snid for that it is configured to serve itself.

Operational requirements: varpd

The operational requirements related to the command opcodes that are sent from the varpdb module to the varpd are described as part of the VARP state machine processing. See below.

Varpmon - another varpdb consumer

There is a second kernel loadable module called varpmon that can register a callback with the varpdb module as described above. The purpose of the varpmon module is to relay a replica of the asynchronous callbacks from varpdb to a user space application for monitoring purposes. Furthermore, the varp system can be queried interactively for existing mappings. We created a Java™ application to create queries and to display any callback activity. This effort is primarily a debugging tool. The mechanisms used similar to the way varpd communicates with varpdb; the interface between varpdb and varpmon was already described above. We will therefore not present the details in this document.

Interface `VARP if` (Virtual Address Resolution Protocol) between VARP peers

In the role of a VARP server, varpd processes the following input events:

- VARP_REQUEST packets from VARP peers
- VARPD_OPCODE_REQUEST, VARPD_OPCODE_RESULT, and VARPD_OPCODE_NAK commands from its varpdb.

Upon receipt of a VARP_REQUEST for (snid,vaddr) from a VARP client at address caddr, the VARP server takes the following actions:

1. determine if varpd is authoritative for serving addresses for supernet snid.
If not,
 - return VARP_NAK_NOAUTHORITY packet.
 - The varpd should send an error message to the systems error log facility, because the use of message VARP_NAK_NOAUTHORITY indicates a misconfiguration of the Supernet client.
 - End processing of this request.
2. send command VARPD_OPCODE_LOOKUP snid vaddr caddr to varpdb.

Upon receipt of a VARPD_OPCODE_RESULT snid vaddr raddr caddr command from the varpdb, the VARP server creates a VARP_REPLY packet with the mapping (snid,vaddr) to (raddr) and sends it to client caddr. The packet contains the default ttl configured for this Supernet.

Upon receipt of a VARPD_OPCODE_REQUEST snid vaddr command from the varpdb, the VARP server determines if he is the server for Supernet snid. If that is the case, he issues the command VARPD_OPCODE_NAK snid vaddr to the varpdb. The other case (where varpd is just another client for Supernet snid) is described in the section VARP client operational requirements.

Upon receipt of a VARPD_OPCODE_NAK snid vaddr caddr command from the varpdb, the VARP server creates a VARP_NAK_NOTKNOWN packet for address (snid,vaddr) and sends it to client caddr. The packet contains a ttl of 0.

Operational Requirements: varpd as VARP Client

In the role of a VARP client, varpd processes the following input events:

- VARP_REPLY, VARP_NAK_NOTKNOWN, VARP_NAK_NOAUTHORITY packets from VARP peers
- VARPD_OPCODE_REQUEST commands from its varpdb

Upon receipt of a VARP_REPLY with contents (snid,vaddr,raddr,ttl) from a VARP server, the VARP client issues the following command to its varpdb: VARPD_OPCODE_REPLY snid vaddr raddr ttl

Upon receipt of a VARP_NAK_NOTKNOWN with contents (snid,vaddr) from a VARP server, the VARP client issues the following command to its varpdb: VARPD_OPCODE_NAK snid vaddr;.

Upon receipt of a VARP_NAK_NOAUTHORITY with contents (snid,vaddr) from a VARP server, the VARP client issues the following command to its varpdb: VARPD_OPCODE_NAK snid vaddr. It should also send an error message to some system console or system log facility, because the NOAUTHORITY command indicates a local misconfiguration.

Upon receipt of a `VARPD_OPCODE_REQUEST snid vaddr` command from the `varpdb`, the VARP client looks up the VARP server address for the requested Supernet snid in its internal data structure. If that information is not available, an error message is logged to the system log facility, because the VARP client is not completely configured. If the server's IP address is found, the VARP client creates a `VARP_REQUEST` packet for address (snid,vaddr) and sends it to the VARP server for Supernet snid.

VARP Table Aging

Database entries in `varpdb` are either server entries which are maintained by the Supernet group manager, or they are client entries learned via the VARP protocol.

Server entries do not time out and are kept in the database with a time to live (ttl) of `VARPDB_TTL_ALWAYS`. Client entries are entered into the database with the initial ttl value that was communicated in the `VARP_REPLY`.

Ttl values must be positive integers and they represent seconds. The maximum ttl is 2^{16} seconds, i.e. a little over 18 hours. The default value for initial client ttl are `VARPDB_TTL_INITIAL_CLIENT` seconds. A value of 900, for example, translates into 15 minutes.

The `varpdb` module must update the ttl values of client entries periodically (every `VARPDB_TTL_UPDATE` seconds) and automatically remove entries for which the ttl has expired. A future version of VARP may include an automatic refresh when a water mark value for the ttl is reached.

System Constants

```
#define  VARPDB_TTL_UPDATE          3
#define  VARPDB_TTL_ALWAYS         -1

#define  VARPDB_TTL_INITIAL_LOOKUP  5
#define  VARPDB_TTL_INITIAL_CLIENT 900
#define  VARPDB_TTL_INITIAL_SERVER VARPDB_TTL_ALWAYS

#define  VARPDB_TTL_CLT_PORT_ACCESS 3

#define  VARPDB_GRP_MGR_PORT        2500      /* TCP */
#define  VARPDB_GRP_CLT_PORT        2501      /* TCP */
#define  VARPDB_VARP_PORT           2500      /* UDP */
```

Open Issues

- Broadcast and multicast addresses.
- Reliability: single server => single point of failure.
- Concern over single `varpd` per machine: Is the separation of private address resolution data assured?

References

1. Caronni, Gupta, Kumar, Markson, Schuba, Scott. Viper: A Virtual Protected Extended Rampart. SunLabs work in progress.

Author Information

Christoph L. Schuba
christoph.schuba@sun.com
Networking and Security Center
Sun Microsystems Laboratories
901 San Antonio Rd
Palo Alto, CA 94303 - USA
Tel +1 (650) 786-3683
Fax +1 (650) 786-6445

SUN proprietary and confidential!!!
