# FIB 2.0: Hierarchical, Protocol Independent.

# Table of Contents

# Prerequisites

This section describes some prerequisite topics and nomenclature that are foundational to understanding the FIB architecture.

## Graphs

The FIB is essentially a collection of related graphs. Terminology from graph theory is often used in the sections that follow. From Wikipedia:

> *"... a graph is a representation of a set of objects where some pairs of objects are connected by links. The interconnected objects are represented by mathematical abstractions called vertices (also called nodes or points), and the links that connect some pairs of vertices are called edges (also called arcs or lines) ... edges may be directed or undirected."*

In a directed graph the edges can only be traversed in one direction – from child to parent. The names are chosen to represent the many to one relationship. A child has one parent, but a parent many children. In undirected graphs the edge traversal can be in either direction, but in FIB the parent child nomenclature remains to represent the many to one relationship. Children of the same parent are termed siblings. When the traversal is from child to parent it is considered to be a forward traversal, or walk, and from parent to the many children a back walk. Forward walks are cheap since they start from the many and move toward the few. Back walks are expensive as the start from the few and visit the many.

The many to one relationship between child and parent means that the lifetime of a parent object must extend to the lifetime of its children. If the control plane removes a parent object before its children, then the parent must remain, in an 'incomplete' state, until the children are themselves removed. Likewise if a child is created before its parent, the parent is completed in an incomplete state. These incomplete objects are needed to maintain the graph dependencies. Without them when the parent is added finding the affected children would be search through many databases for those children. To extend the lifetime of parents all children thereof hold a 'lock' on the parent. This is a simple reference count. Children then follow the add-or-lock/unlock semantics for finding a parent, as opposed to a malloc/free.

## Prefixes

Some nomenclature used to describe prefixes;

- 1.1.1.1 – this is an address since it has no associated mask
- 1.1.1.0/24 – this is a prefix.
- 1.1.1.1/32 – this is a host prefix (the mask length is the size of the address).

Prefix A is more specific than B if its mask length is longer, and less specific if the mask is shorter. For example, 1.1.1.0/28 is more specific than 1.1.1.0/24. A less specific prefix that overlaps with a more specific is the 'covering' prefix. For example, 1.1.1.0/24 is the covering prefix for 1.1.1.0/28 and 1.1.1.0/28 is termed the 'covered' prefix. A covering prefix is therefore always less specific than its covered prefixes.

# The Data Model

The FIB data model comprises two parts; the control-plane (CP) and the data-plane (DP). The CP data model represents the data that is programmed into VPP by the upper layers. The DP model represents how VPP derives actions to be performed on packets are they are switched.

## Control Plane

The control plane follows a layered data representation. This document describes the model starting from the lowest layer. The description uses IPv4 addresses and protocols, but all concepts apply equally to the IPv6 equivalents. The diagrams all portray the CLI command to install the information in VPP and an [approximation of] a UML diagram[1] of the data structures used to represent that information.

### ARP Entries

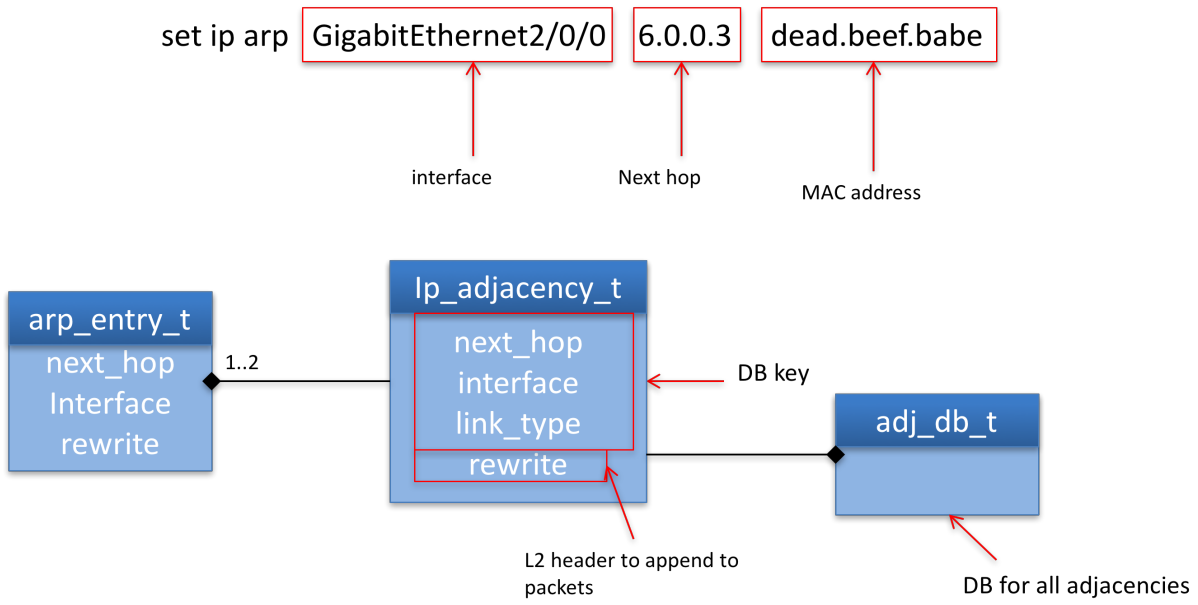


Figure 1: ARP data model

Figure 1 shows the data model for an ARP entry. An ARP entry contains the mapping between a peer, identified by an IPv4 address, and its MAC address on a given interface. The VRF the interface is bound to, is not part of the data. VRFs are an ingress function not egress. The ARP entry describes how to send traffic to a peer, which is an egress function.

The *arp_entry_t* represents the control-plane addition of the ARP entry. The *ip_adjacency_t* contains the data derived from the *arp_entry_t* that is need to forward packets to the peer. The additional data in the adjacency are the *rewrite* and the *link_type*. The *link_type* is a description of the protocol of the packets that will be forwarded with this adjacency; this can be IPv4 or MPLS. The *link_type* maps directly to the ether-type in an Ethernet header, or the protocol filed in a GRE header. The rewrite is a byte string representation of the header that will be prepended to the packet when it is

[1] The arrow indicates a 'has-a' relationship. The object attached to the arrow head 'has-a' instance of the other. The numbers next to the arrows indicate the multiplicity, i.e. object A has n to m instances of object B. The difference between a UML association and aggregation is not conveyed in any diagrams. To UML aficionados, I apologize. Word is not the best drawing tool.

sent to that peer. For Ethernet interfaces this would be the src,dst MAC and the ether-type. For LISP tunnels, the IP src,dst pair and the LISP header.

The *arp_entry_t* will install a *link_type=IPv4* when the entry is created and a *link_type=MPLS* when the interface is MPLS enabled. Interfaces must be explicitly MPLS enabled for security reasons.

So that adjacencies can be shared between route, adjacencies are stored in a single data-base, the key for which is *{interface, next-hop, link-type}*.

## Routes

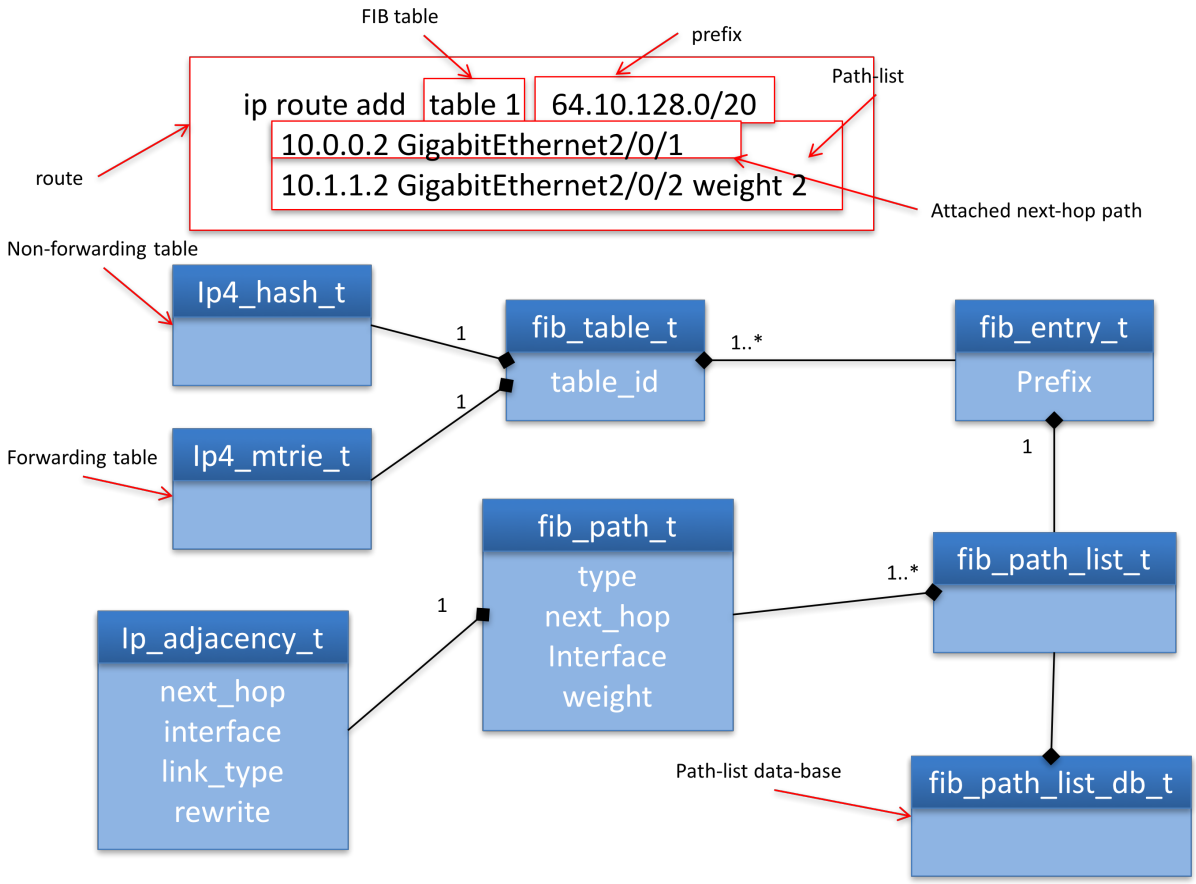


**Figure 2: Route data model – class diagram**

The control plane will install a route in a table for a prefix via a list of paths. The prime function of the FIB is to 'resolve' that route. To resolve a route is to construct an object graph that fully describes all elements of the route. In Figure 3 the route is resolved as the graph is complete from *fib_entry_t* to *ip_adjacency_t*.

In some routing models a VRF will consist of a set of tables for IPv4 and IPv6, and unicast and multicast. In VPP there is no such grouping. Each table is distinct from each other. A table is indentified by its numerical ID. The ID range is separate for each address family.

A table is comprised of two route data-bases; forwarding and non-forwarding. The forwarding data-base contains routes against which a packet will perform a longest prefix match (LPM) in the data-plane. The non-forwarding DB contains all the routes with which VPP has been programmed – some

of these routes may be unresolved for reasons that prevent their insertion into the forwarding DB (see section: Adjacency source FIB entries).

The route data is decomposed into three parts; entry, path-list and paths;

- The fib_entry_t, which contains the route's prefix, is representation of that prefix's entry in the FIB table.
- The *fib_path_t* is a description of where to send the packets destined to the route's prefix. There are several types of path.
    - Attached next-hop: the path is described with an interface and a next-hop. The next-hop is in the same sub-net as the router's own address on that interface, hence the peer is considered to be 'attached'.
    - Attached: the path is described only by an interface. All address covered by the prefix are on the same L2 segment to which that router's interface is attached. This means it is possible to ARP for any address covered by the prefix – which is usually not the case (hence the proxy ARP debacle in IOS). An attached path is only appropriate for a point-to-point (P2P) interface where ARP is not required, i.e. a GRE tunnel.
    - Recursive: The path is described only via the next-hop and table-id.
    - De-aggregate: The path is described only via the special all zeros address and a table-id. This implies a subsequent lookup in the table should be performed.
- The *fib_path_list_t* represents the list of paths from which to choose one when forwarding. The path-list is a shared object, i.e. it is the parent to multiple *fib_entry_t* children. In order to share any object type it is necessary for a child to search for an existing object matching its requirements. For this there must be a data-base. The key to the path-list data-base is a combined description of all of the paths it contains[2]. Searching the path-list database is required with each route addition, so it is populated only with path-lists for which sharing will bring convergence benefits (see Section: Fast Convergence).

Figure 2 shows an example of a route with two attached-next-hop paths. Each of these paths will 'resolve' by finding the adjacency that matches the path's attributes, which are the same as the key for the adjacency data-base[3]. The 'forwarding information' (FI) is the set of adjacencies that are available for load-balancing the traffic in the data-plane. A path 'contributes' an adjacency to the route's forwarding information, the path-list contributes the full forwarding information for IP packets.

**Error! Reference source not found.** shows the object instances and their relationships created in order to resolve the routes also shown. The graph nature of these relationships is evident; children are displayed at the top of the diagram, their parents below them. Forward walks are thus from top to bottom, back walks bottom to top. The diagram shows the objects that are shared, the path-list and adjacency. Sharing objects is critical to fast convergence (see section Fast Convergence).

[2] Optimisations

3 Note it is valid for either interface to be bound to a different table than table 1.

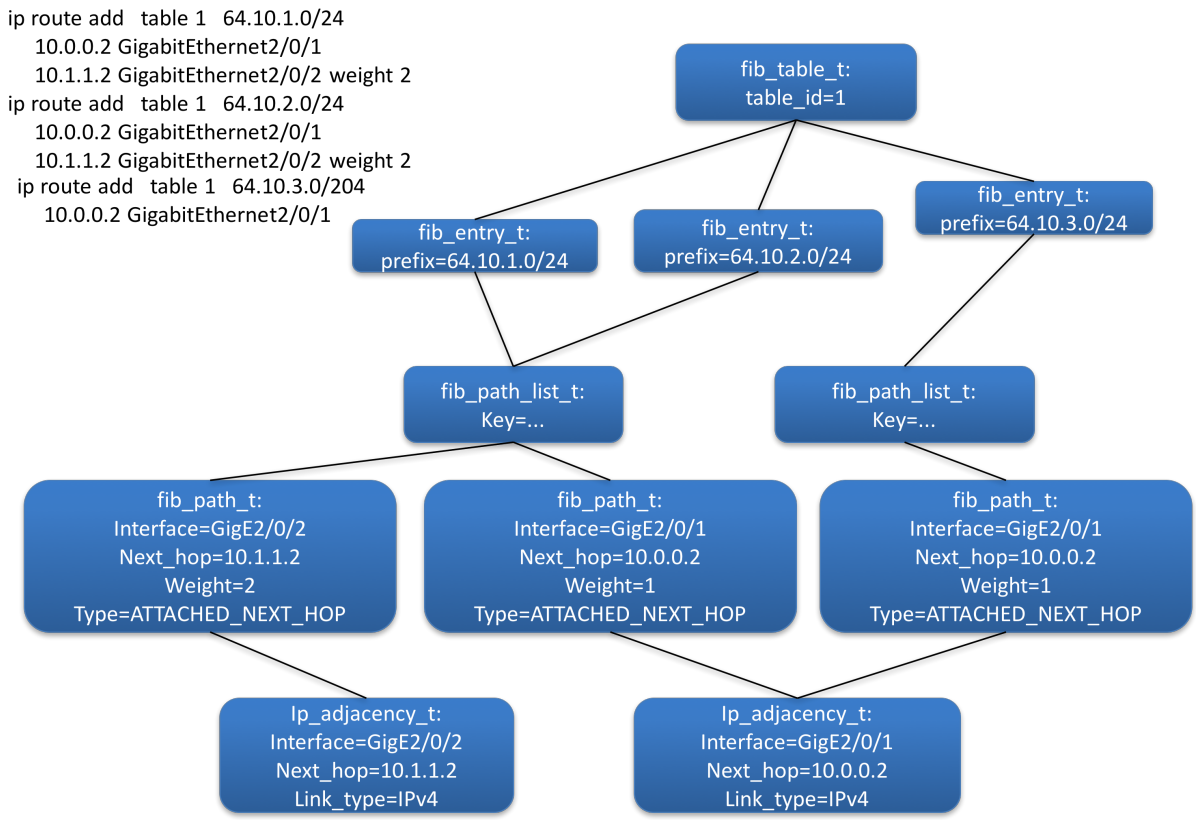


**Figure 3: Route object diagram**

### *FIB sources*

There are various entities in the system that can add routes to the FIB tables. Each of these entities is termed a 'source'. When the same prefix is added by different sources the FIB must arbitrate between them to determine which source will contribute the forwarding information. Since each source determines the forwarding information using different best path and loop prevention algorithms, it is not correct for the forwarding information of multiple sources to be combined. Instead the FIB must choose to use the forwarding information from only one source. This choice is based on a static priority assignment[4]. The FIB must maintain the information each source has added so it can be restored should that source become the best source. VPP has two 'control-plane' sources; the API and the CLI – the API has the higher priority. Each source's data is represented by a *fib_entry_src_t* object – of which a *fib_entry_t* maintains a sorted vector.

A prefix is 'connected' when it is applied to a router's interface. The following configuration:

**set interface address 192.168.1.1/24 GigE0**

results in the addition of two FIB entries; 192.168.1.0/24 which is connected and attached, and 192.168.1.1/32 which is connected and local (a.k.a receive or for-us). Both prefixes are 'interface' sourced. The interface source has a high priority, so the accidental or nefarious addition of identical prefixes does not prevent the router from correctly forwarding. Packets matching a connected prefix will generate an ARP request for the packet's destination address, this process is known as a 'glean'.

[4] The engaged reader can see the full priority list in vnet/vnet/fib/fib_entry.h.

An ‘attached’ prefix also results in a glean, but the router does not have its own address in that sub-net. The following configuration will result in an attached route, which resolves via an attached path;

**ip route add table X 10.10.10.0/24 via gre0**

as mentioned before, these are only appropriate for point-to-point links. An attached-host prefix is covered by either an attached prefix (note that connected prefixes are also attached). If table X is not the table to which gre0 is bound, then this is the case of an attached export (see section Attached Export)

### *Adjacency source FIB entries*

Whenever an ARP entry is created it will source a *fib_entry_t*. In this case the route is of the form:

**ip route add table X 10.0.0.1/32 via 10.0.0.1 GigEth0/0/0**

It is a host prefix with a path whose next-hop address is the same. This route highlights the distinction between the route’s prefix – a description of the traffic to match - and the path – a description of where to send the matched traffic. Table X is the same table to which the interface is bound. FIB entries that are sourced by adjacencies are termed adj-fibs. The priority of the adjacency source is lower than the API source, so the following configuration:

**set interface address 192.168.1.1/24 GigE0**

**ip arp 192.168.1.2 GigE0 dead.dead.dead**

**ip route add 192.168.1.2 via 10.10.10.10 GigE1**

will forward traffic for 192.168.1.2 via GigE1. That is the route added by the control plane is favoured over the adjacency discovered by ARP. The control plane, with its associated authentication, is considered the authoritative source. To counter the nefarious addition of adj-fibs, through the nefarious injection of adjacencies, the FIB is also required to ensure that only adj-fibs whose less specific covering prefix is attached are installed in forwarding. This requires the use of 'cover tracking', where a route maintains a dependency relationship with the route that is its less specific cover. When this cover changes (i.e. there is a new covering route) or the forwarding information of the cover is updated, then the covered route is notified. Adj-fibs that fail this cover check are not installed in the *fib_table_t*’s forwarding table, there are only present in the non-forwarding table.

Overlapping sub-nets are not supported, so no adj-fib has multiple paths. The control plane is expected to remove a prefix configured for an interface before the interface changes VRF. So while the following configuration is accepted:

**set interface address 192.168.1.1/32 GigE0**

**ip arp 192.168.1.2 GigE0 dead.dead.dead**

**set interface ip table GigE0 2**

it does not result in the desired behaviour, where the adj-fib and connecteds are moved to table 2.

### *Recursive Routes*

Figure 4 shows the data structures used to describe a recursive route. The representation is almost identical to attached next-hop paths. The difference being that the *fib_path_t* has a parent that is another *fib_entry_t*, termed the 'via-entry'.

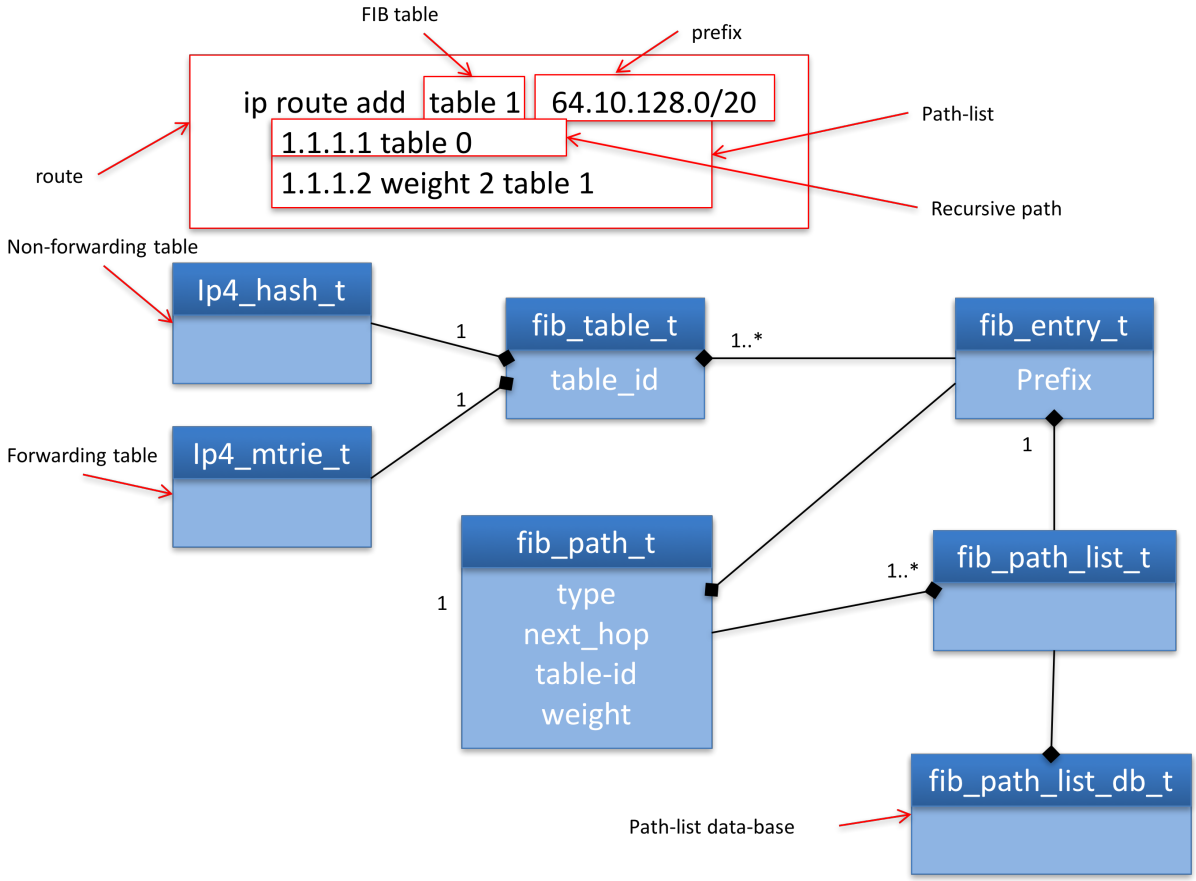


**Figure 4: Recursive route class diagram.**

In order to forward traffic to 64.10.128.0/20 the FIB must first determine how to forward traffic to 1.1.1.1/32. This is recursive resolution. Recursive resolution, which is essentially a cache of the data-plane result, emulates a longest prefix match for the 'via-address' 1.1.1.1 in the 'via-table' table 0[5].

Recursive resolution (RR) will source a host-prefix entry in the via-table for the via-address. The RR source is a low priority source. In the unlikely[6] event that the RR source is the best source, then it must derive forwarding information from its covering prefix. There are two cases to consider:

- The cover is connected[7]. The via-address is then an attached host and the RR source can resolve directly via the adjacency with the key {via-address, interface-of-connected-cover}
- The cover is not connected[8]. The RR source can directly inherit the forwarding information from its cover.

[5] Note it is only possible to add routes via an address (i.e. a /32 or /128) not via a shorter mask prefix. There is no use case for the latter

[6] For iBGP the via-address is the loopback address od the peer PE, for eBGP it is the adj-fib for the CE.

[7] As is the case for eBGP

[8] As is the case for iBGP

This dependency on the covering prefix means the RR source will track its cover. The covering prefix will 'change' when;

- A more specific prefix is inserted. For this reason whenever an entry is inserted into a FIB table its cover must be found so that its covered dependents can be informed.
- The existing cover is removed. The covered prefixes must form a new relationship with the next less specific.

The cover will be 'updated' when the route for the covering prefix is modified. The cover tracking mechanism will provide the RR sourced entry with a notification in the event of a change or update of the cover, and the source can take the necessary action.

The RR sourced FIB entry becomes the parent of the *fib_path_t* and will contribute its forwarding information to that path, so that the child's FIB entry can construct its own forwarding information.

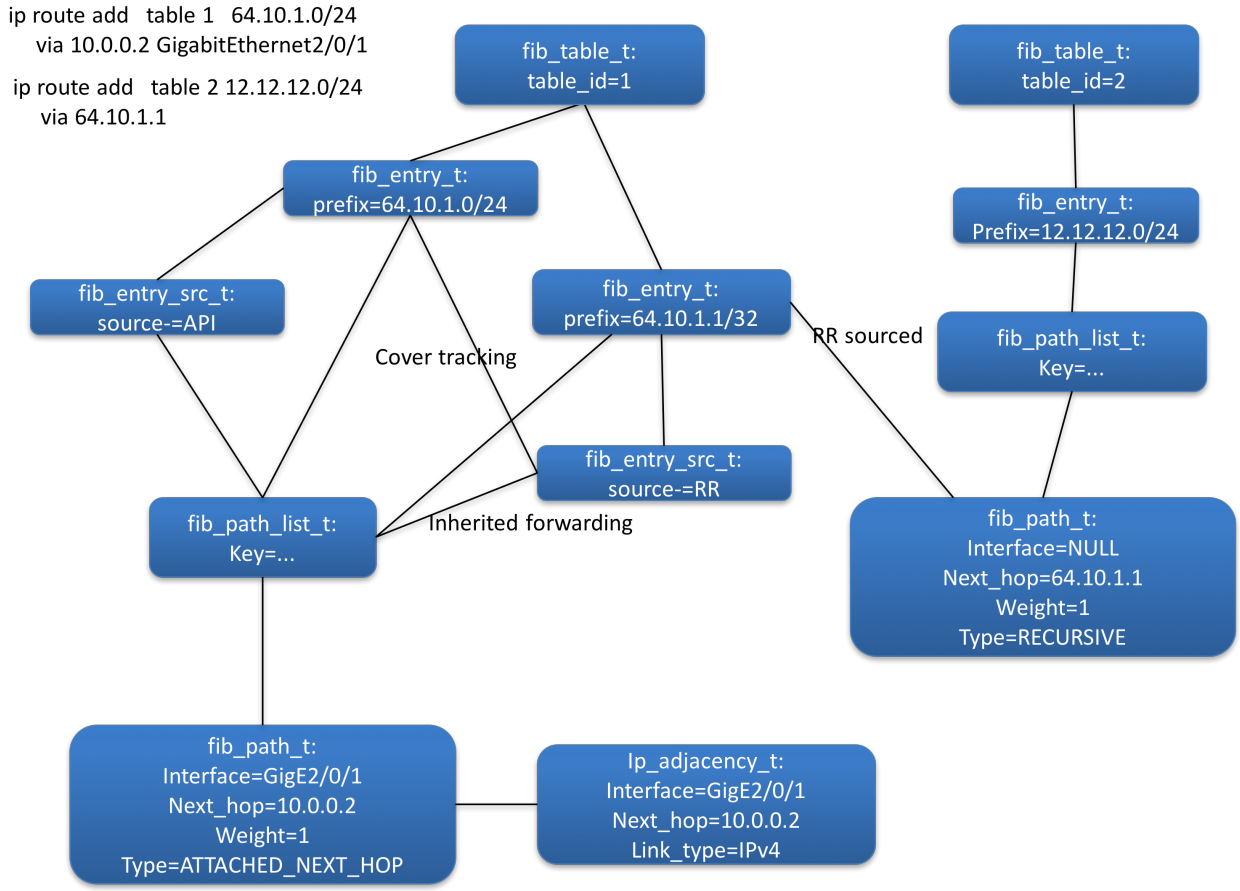


**Figure 5: Recursive Routes object diagram**

Figure 5 shows the object instances created to represent the recursive route and its resolving route also shown.

If the source adding recursive routes does not itself perform recursive resolution[9] then it is possible that the source may inadvertently programme a recursion loop. An example of a recursion loop is the following configuration:

**ip route add 5.5.5.5/32 via 6.6.6.6**

**ip route add 6.6.6.6/32 via 7.7.7.7**

**ip route add 7.7.7.7/32 via 5.5.5.5**

This shows a loop over three levels, but any number is possible. FIB will detect recursion loops by forward walking the graph when a *fib_entry_t* forms a child-parent relationship with a *fib_path_list_t*. The walk checks to see if the same object instances are encountered. When a recursion loop is formed the control plane[10] graph becomes cyclic, thus allowing the child-parent dependencies to form. This is necessary so that when the loop breaks, the affected children and be updated.

### *Output labels*

A route may have associated out MPLS labels[11]. These are labels that are expected to be imposed on a packet as it is forwarded. It is important to note that an MPLS label is per-route and per-path, therefore, even though routes share paths the do not necessarily have the same label for that path[12]. A label is therefore uniquely associated to a *fib_entry_t* and associated with one of the *fib_path_t* to which it forwards.

MPLS labels are modelled via the generic concept of a 'path-extension'. A *fib_entry_t* therefore has a vector of zero to many *fib_path_ext_t* objects to represent the labels with which it is configured.

### Attached Export

Extranets make prefixes in VRF A also reachable from VRF B. VRF A is the export VRF, B the import. Consider this route in the export VRF;

**ip route add table 2 1.1.1.0/24 via 10.10.10.0 Gige0/0/0**

there are two ways one might consider representing this route in the import VRF:

1) **ip route add table 3 1.1.1.0/24 via 10.10.10.0 Gige0/0/0**
2) **ip route add table 3 1.1.1.0/24 via lookup-in-table 2**

where option 2) is an example of a de-aggregate route where a second lookup is performed in table 2, the export VRF. Option 2) is clearly less efficient, since the cost of the second lookup is high. Option 1) is therefore preferred. However, connected and attached prefixes, and specifically the adj-fibs that they cover, require special attention. The control plane is aware of the connected and attached prefixes that are required to be exported, but it is unaware of the adj-fibs. It is therefore the responsibility of FIB to ensure that whenever an attached prefix is exported, so are the adj-fibs and local prefixes that it covers, and only the adj-fibs and locals, not any covered more specific (sourced e.g. by API). The imported FIB entries are sourced as 'attached-export', this is a low priority

[9] If that source is relying on FIB to perform recursive resolution, then there is no reason it should do so itself.
[10] The derived data-plane graph MUST never be cyclic.
[11] Advertised, e.g. by LDP, SR or BGP.
[12] The only case where the labels will be the same is BGP VPNv4 label allocation per-VRF.

source, so if those prefixes already exist in the import VRF, sourced by the API, then they will continue to forward with that information.

Figure 6 shows the data structures used to perform attached export.

- *fib_import_t*. A representation of the need to import covered prefixes. An instance is associated with the FIB entry in the import VRF. The need to import prefixes is recognised when an attached route is added to a table that is different to the table of the interface to which it t is attached. The creation of a *fib_import_t* will trigger the creation of a *fib_export_t*.
- *fib_export_t*. A representation of the need to export prefixes. An instance is associated with the attached entry in the export VRF. A *fib_export_t* can have many associated *fib_import_t* objects representing multiple VRFs into which the prefix is exported.

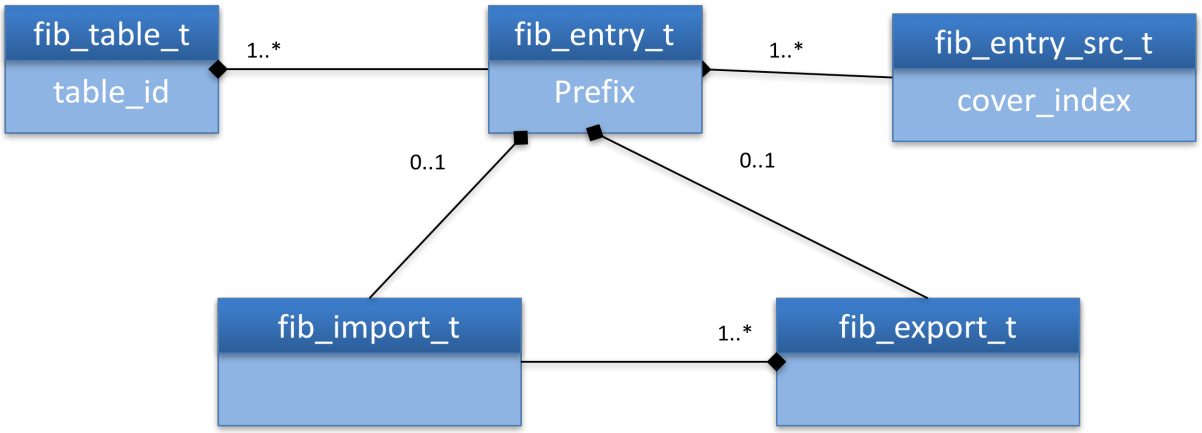


**Figure 6: Attached Export Class diagram.**

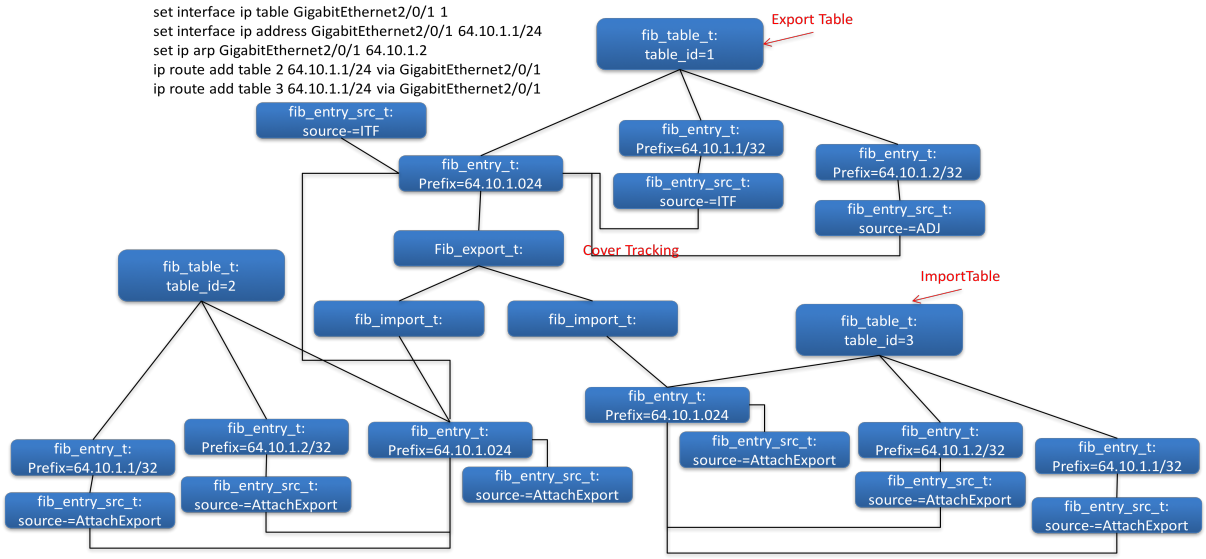


**Figure 7: Attached Export object diagram**

Figure 7 shows an object instance diagram for the export of a connected from table 1 to two other tables. The /32 adj-fib and local prefix in the export VRF are exported into the import VRFs, where they are sourced as 'Attached-export' and inherit the forwarding information from the exported

entry. The attached prefix in the import VRF also performs cover tracking with the connected prefix in the export VRF so that it can react to updates to that prefix that will require the removal the imported covered prefixes.

### Graph Walks

All FIB object types are allocated from a VPP memory pool[13]. The objects are thus susceptible to memory re-allocation, therefore the use of a bare 'C' pointer to refer to a child or parent is not possible. Instead there is the concept of a *fib_node_ptr_t* which is a tuple of *type,index*. The type indicates what type of object it is (and hence which pool to use) and the index is the index in that pool. This allows for the safe retrieval of any object type.

When a child resolves via a parent it does so knowing the type of that parent. The child to parent relationship is thus fully known to the child, and hence a forward walk of the graph (from child to parent) is trivial. However, a parent does not choose its children, it does not even choose the type. All object types that form part of the FIB control plane graph all inherit from a single base class[14]; *fib_node_t*. A *fib_node_t* indentifies the object's index and its associated virtual function table provides the parent a mechanism to 'visit' that object during the walk. The reason for a back-walk is to inform all children that the state of the parent has changed in some way, and that the child may itself need to update.

To support the many to one, child to parent, relationship a parent must maintain a list of its children. The requirements of this list are;

- O(1) insertion and delete time. Several child-parent relationships are made/broken during route addition/deletion.
- Ordering. High priority children are at the front, low priority at the back (see section Fast Convergence)
- Insertion at arbitrary locations.

To realise these requirements the child-list is a doubly linked-list, where each element contains a *fib_node_ptr_t*. The VPP pool memory model applies to the list elements, so they are also identified by an index. When a child is added to a list it is returned the index of the element. Using this index the element can be removed in constant time. The list supports 'push-front' and 'push-back' semantics for ordering. To walk the children of a parent is then to iterate of this list.

A back-walk of the graph is a depth first search where all children in all levels of the hierarchy are visited. Such walks can therefore encounter all object instances in the FIB control plane graph, numbering in the millions. A FIB control-plane graph is cyclic in the presence of a recursion loop, so the walk implementation has mechanisms to detect this and exit early.

A back-walk can be either synchronous or asynchronous. A synchronous walk will visit the entire section of the graph before control is returned to the caller, an asynchronous walk will queue the walk to a background process, to run at a later time, and immediately return to the caller. To implement asynchronous walks a *fib_walk_t* object it added to the front of the parent's child list. As children are visited the *fib_walk_t* object advances through the list. Since it is inserted in the list,

[13] Fast memory allocation is crucial to fast route update times.
[14] VPP may be written in C and not C++ but inheritance is still possible.

when the walk suspends and resumes, it can continue at the correct location. It is also safe with respect to the deletion of children from the list. New children are added to the head of the list, and so will not encounter the walk, but since they are new, they already have the up to date state of the parent.

A VLIB process 'fib-walk' runs to perform the asynchronous walks. VLIB has no priority scheduling between respective processes, so the fib-walk process does work in small increments so it does not block the main route download process. Since the main download process effectively has priority numerous asynchronous back-walks can be started on the same parent instance before the fib-walk process can run. FIB is a 'final state' application. If a parent changes n times, it is not necessary for the children to also update n times, instead it is only necessary that this child updates to the latest, or final, state. Consequently when multiple walks on a parent (and hence potential updates to a child) are queued, these walks can be merged into a single walk.

Choosing between a synchronous and an asynchronous walk is therefore a trade-off between time it takes to propagate a change in the parent to all of its children, versus the time it takes to act on a single route update. For example, if a route update where to affect millions of child recursive routes, then the rate at which such updates could be processed would be dependent on the number of child recursive route – which would not be good. At the time of writing FIB2.0 uses synchronous walk in all locations except when walking the children of a path-list, and it has more than 32[15] children. This avoids the case mentioned above.

## Data-Plane

The data-plane data model is a directed, acyclic[16] graph of heterogeneous objects. A packet will forward walk the graph as it is switched. Each object describes the actions to perform on the packet. Each object type has an associated VLIB graph node. For a packet to forward walk the graph is therefore to move from one VLIB node to the next, with each performing the required actions. This is the heart of the VPP model.

The data-plane graph is composed of generic data-path objects (DPOs). A parent DPO is identified by the tuple*:{type,index,next_node}*. The *next_node* parameter is the index of the VLIB node to which the packets should be sent next, this is present to maximise performance - it is important to ensure that the parent does not need to be read[17] whilst processing the child. Specialisations[18] of the DPO perform distinct actions. The most common DPOs and briefly what they represent are:

- Load-balance: a choice in an ECMP set.
- Adjacency: apply a rewrite and forward through an interface
- MPLS-label: impose an MPLS label.
- Lookup: perform another lookup in a different table.

The data-plane graph is derived from the control-plane graph by the objects therein 'contributing' a DPO to the data-plane graph. Objects in the data-plane contain only the information needed to switch a packet, they are therefore simpler, and in memory terms smaller, with the aim to fit one

[15] The value is arbitrary and yet to be tuned.
[16] Driected implies it cannot be back-walked. It is acyclic even in the presence of a recursion loop.
[17] Loaded into cache, and hence potentially incurring a d-cache miss.
[18] The engaged reader is directed to vnet/vnet/dpo/*

DPO on a single cache-line. The derivation from the control plane means that the data-plane graph contains only object whose current state can forward packets. For example, the difference between a *fib_path_list_t* and a *load_balance_*t is that the former expresses the control-plane's desired state, the latter the data-plane available state. If some paths in the path-list are unresolved or down, then the load-balance will not include them in the forwarding choice.

Figure 8 shows a simplified view of the control-plane graph indicating those objects that contribute DPOs. Also shown are the VLIB node graphs at which the DPO is used.

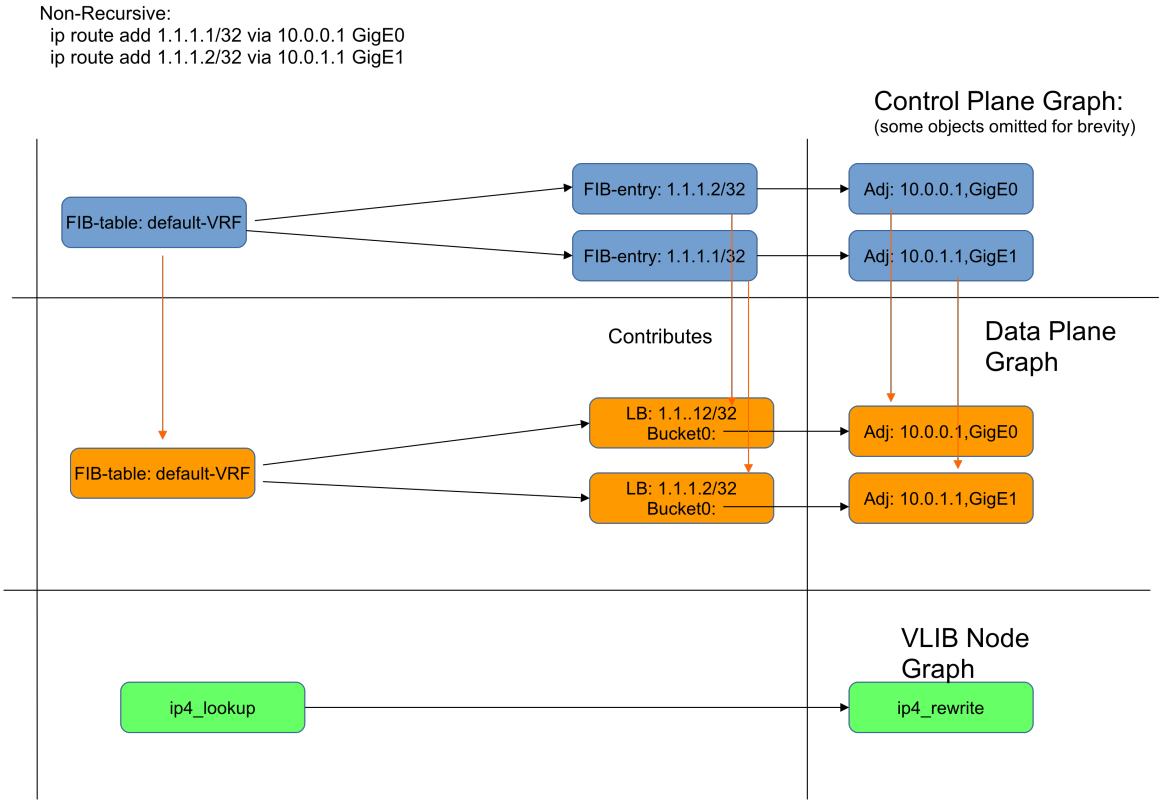


**Figure 8: DPO contributions for a non-recursive route**

Each *fib_entry_t* contributes it own *load_balance_t,* for three reasons*;*

- The result of a lookup in a IPv[46] table is a single 32 bit unsigned integer. This is an index into a memory pool. Consequently the object type must be the same for each result. Some routes will need a load-balance and some will not, but to insert another object in the graph to represent this choice is a waste of cycles, so the load-balance object is always the result. If the route does not have ECMP, then the load-balance has only one choice.
- In order to collect per-route counters, the lookup result must in some way uniquely identify the *fib_entry_t*. A shared load-balance (contributed by the path-list) would not allow this.
- In the case the *fib_entry_t* has MPLS out labels, and hence a *fib_path_ext_t*, then the load-balance must be per-prefix, since the MPLS labels that are its parents are themselves per-*fib_entry_t*.

Figure 9 shows the load-balance objects contributed for a recursive route.

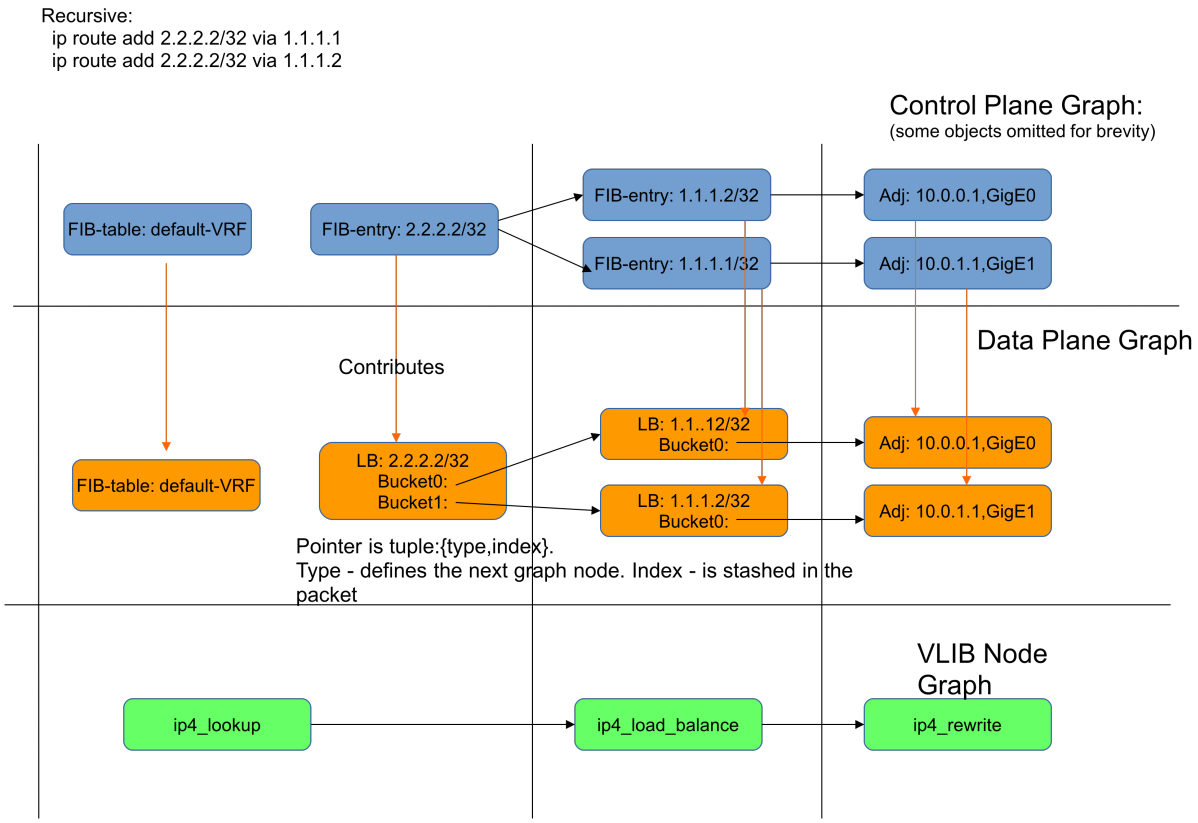


**Figure 9: DPO contribution for a recursive route.**

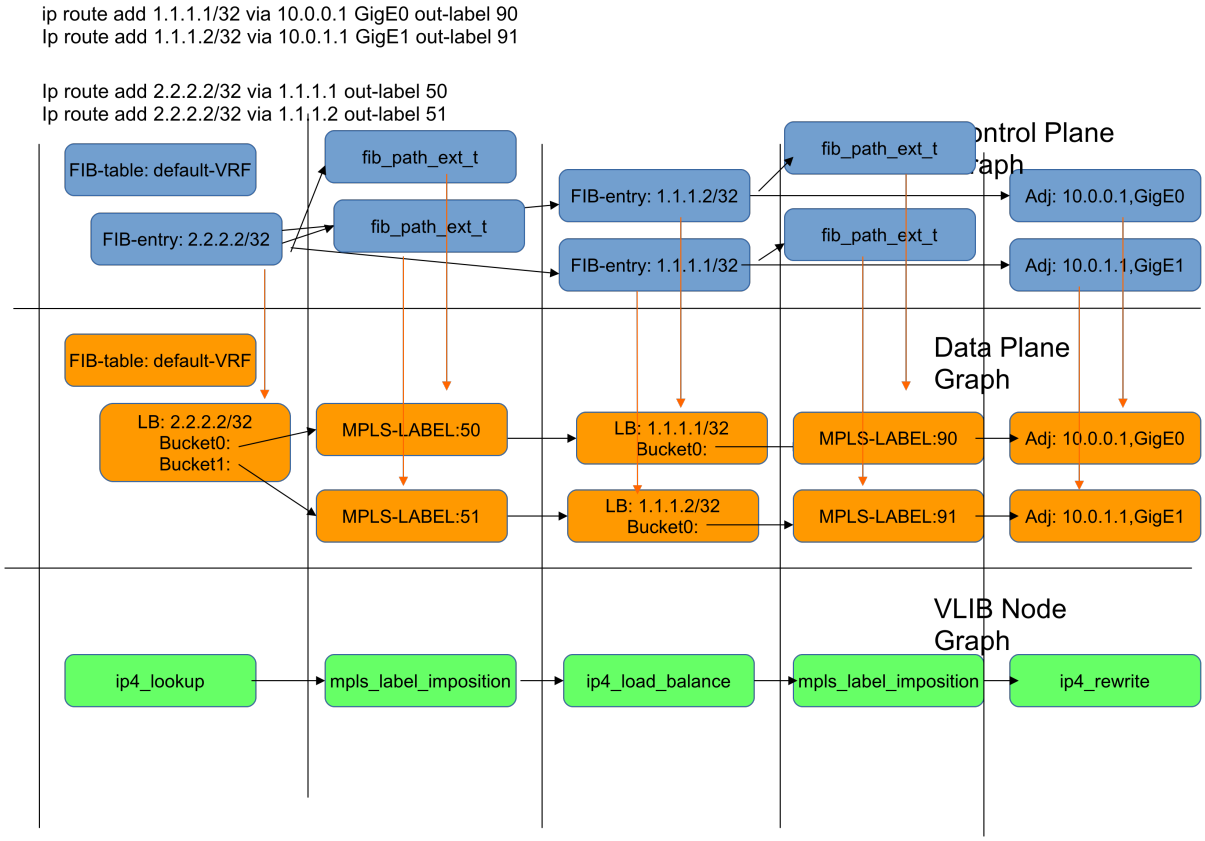


**Figure 10: DPO Contributions from labelled recursive routes.**

Figure 10 shows the derived data-plane graph for a labelled recursive route. There can be as many MPLS-label DPO instances as there are routes multiplied by the number of paths per-route. For this reason the mpls-label DPO should be as small as possible[19].

The data-plane graph is constructed by 'stacking' one instance of a DPO on another to form the child-parent relationship. When this stacking occurs, the necessary VLIB graph arcs are automatically constructed from the respected DPO type's registered graph nodes.

The diagrams above show that for any given route the full data-plane graph is known before any packet arrives. If that graph is composed of n objects, then the packet will visit n nodes and thus incur a forwarding cost of approximately n times the graph node cost. This could be reduced if the graph were 'collapsed' into a single DPO and associated node. However, collapsing a graph removes the indirection objects that provide fast convergence (see section Fast Convergence). To collapse is then a trade-off between faster forwarding and fast convergence; VPP favours the latter.

This DPO model effectively exists today but is informally defined. Presently the only object that is in the data-plane is the *ip_adjacency_t,* however, features (like ILA, OAM hop-by-hop, SR, MAP, etc) sub-type the adjacency. The member *lookup_next_index* is equivalent to defining a new sub-type. Adding to the existing union, or casting sub-type specific data into the *opaque* member, or even over the rewrite string (e.g. the new port range checker), is equivalent defining a new C-struct type. Fortunately, at this time, all these sub-types are smaller in memory than the ip_adjacency_t. It is now possible to dynamically register new adjacency sub-types with ip_register_adjacency() and provide a custom format function.

In my opinion a strongly defined object model will be easier for contributors to understand, and more robust to implement.

## Tunnels

Tunnels share a similar property to recursive routes in that after applying the tunnel encapsulation, a new packet must be forwarded, i.e. forwarding is recursive. However, as with recursive routes the tunnel's destination is known beforehand, so the recursive switch can be avoided if the packet can follow the already constructed data-plane graph for the tunnel's destination. This process of joining to DP graphs together is termed 'stacking'.

[19] i.e. we should not re-use the adjacency structure.

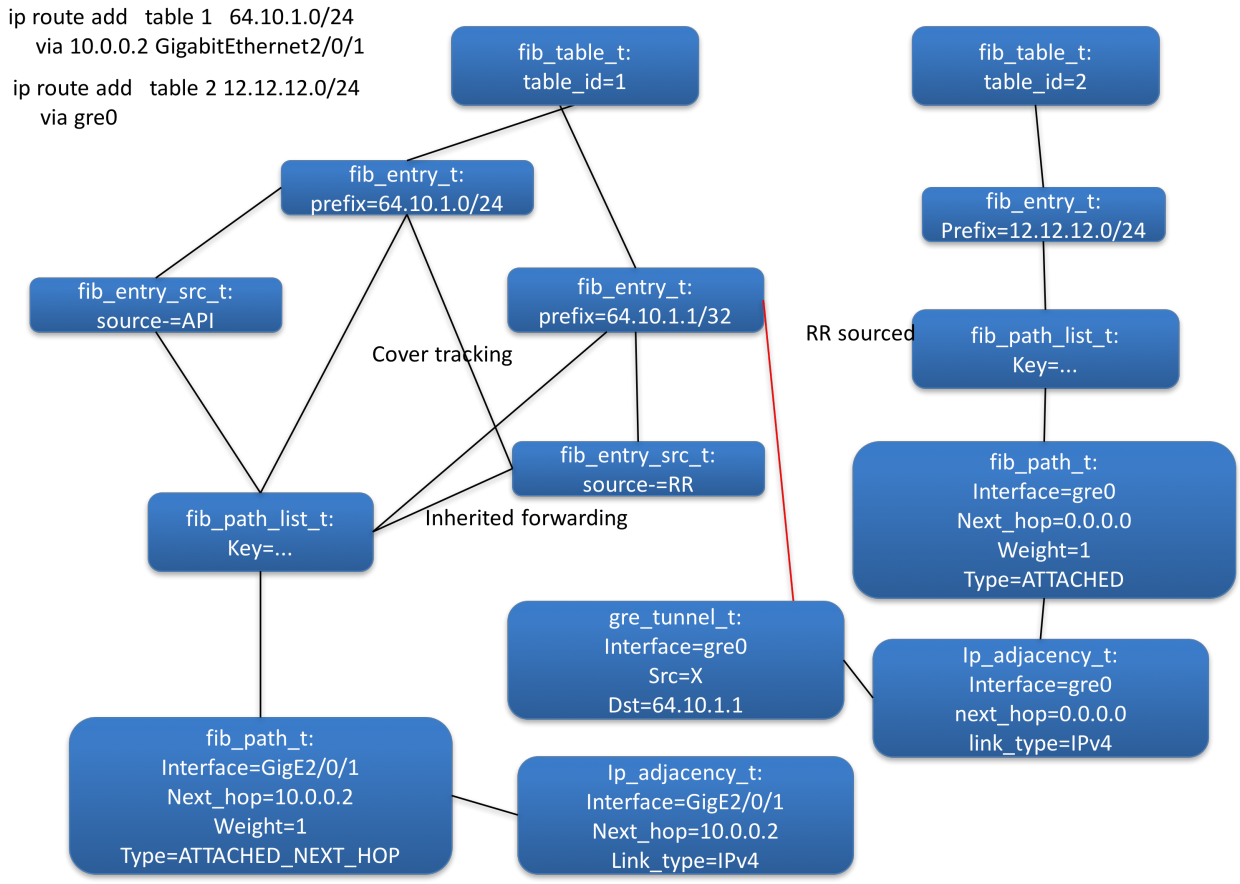


**Figure 11: Tunnel control plane object diagram**

Figure 11 shows the control plane object graph for a route via a tunnel. The two sub-graphs for the route via the tunnel and the route for the tunnel's destination are shown to the right and left respectively. The red line shows the relationship form by stacking the two sub-graphs. The adjacency on the tunnel interface is termed a 'mid-chain' this it is now present in the middle of the graph/chain rather than its usual terminal location.

The mid-chain adjacency is contributed by the *gre_tunnel_t* , which also becomes part of the FIB control-plane graph. Consequently it will be visited by a back-walk when the forwarding information for the tunnel's destination changes. This will trigger it to restack the mid-chain adjacency on the new *load_balance_t* contributed by the parent *fib_entry_t*.

If the back-walk indicates that there is no route to the tunnel, or that the route does not meet resolution constraints, then the tunnel can be marked as down, and fast convergence can be triggered in the same way as for physical interfaces (see section ...).

## MPLS FIB

There is a tight coupling between IP and MPLS forwarding. MPLS forwarding equivalence classes (FECs) are often an IP prefix – that is to say that traffic matching a given IP prefix is routed into a

MPLS label switch path (LSP). It is thus necessary to be able to associated a given prefix/route with an [out-going] MPLS label that will be imposed when the packet is forwarded. This is configured as:

**ip route add 1.1.1.1/32 via 10.10.10.10 GigE0/0/0 out-label 33**

packets matching 1.1.1.1/32 will be forwarded out GigE0/0/0 and have MPLS label 33 imposed. More than one out-going label can be specified. Out-going MPLS labels can be applied to recursive and non-recursive routes, e.g;

**ip route add 2.2.2.0/24 via 1.1.1.1 out-label 34**

packets matching 2.2.2.0/24 will thus have two MPLS labels imposed; 34 and 33. This is the realisation of, e,g, an MPLS BGP VPNv4.

The router receiving the MPLS encapsulated packets needs to be programmed with actions associated which each label value – this is the role of the MPLS FIB. The MPLS FIB Is a table, whose key is the MPLS label value and end-of-stack (EOS) bit, which stores the action to perform on packets with matching encapsulation. Currently supported actions are:

1) Pop the label and perform an IPv[46] lookup in a specified table
2) Pop the label and forward via a specified next-hop (this is penultimate-hop-pop, PHP)
3) Swap the label and forward via a specified next-hop.

These can be programmed respectively by:

1) **mpls local-label 33 ip4-lookup-in-table X**
2) **mpls local-label 33 via 10.10.10.10 GigE0/0/0**
3) **mpls local-label 33 via 10.10.10.10 GigE0/0/0 out-label 66**

the latter is an example of an MPLS cross connect. Any description of a next-hop, recursive, non-recursive, labelled, non-labelled, etc, that is valid for an IP prefix, is also valid for an MPLS local-label.

## Implementation

The MPLS FIB is implemented using exactly the same data structures as the IP FIB.

The only difference is the implementation of the table. Whereas for IPv4 this is an mtrie and for IPv6 a hash table, for MPLS it is a flat array indexed by a 21 bit key (label & EOS bit). This implementation is chosen to favour packet forwarding speed.

## Tunnels

VPP no longer supports MPLS tunnels that are coupled to a particular transport, i.e. MPLSoGRE or MPLSoEth. Such tight coupling is not beneficial. Instead VPP supports;

1) MPLS LSPs associated with IP prefixes and MPLS local-labels (as described above) which are transport independent (i.e. the IP route could be reachable over a GRE tunnel, or any other interface type).
2) A generic uni-directional MPLS tunnel interface that is transport independent.

An MPLS tunnel is effectively an LSP with an associated interface. The LSP can be described by any next-hop type (recursive, non-recursive etc), e.g.:

**mpls tunnel add via 10.10.10.10 GigE0/0/0 out-label 66**

IP routes and/or MPLS x-connects can be routed via the interface, e.g.

**ip route add 2.2.2.0/24 via mpls-tunnel0**

packets matching the route for 2.2.2.0/24 would thus have label 66 imposed since it is transmitted via the tunnel.

These MPLS tunnels can be used to realise MPLS RSVP-TE tunnels.

## Fast Convergence