.. \_softwarearchitecture:

.. toctree::

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

# Software Architecture

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

The fd.io vpp implementation is a third-generation vector packet

processing implementation specifically related to US Patent 7,961,636,

as well as earlier work. Note that the Apache-2 license specifically

grants non-exclusive patent licenses; we mention this patent as a

point of historical interest.

For performance, the vpp dataplane consists of a directed graph of

forwarding nodes which process multiple packets per invocation. This

schema enables a variety of micro-processor optimizations: pipelining

and prefetching to cover dependent read latency, inherent I-cache

phase behavior, vector instructions. Aside from hardware input and hardware output nodes,

the entire forwarding graph is portable code.

Depending on the scenario at hand, we often spin up multiple worker

threads which process ingress-hashes packets from multiple queues using

identical forwarding graph replicas.

## Implementation Taxonomy

=======================

The vpp dataplane consists of four distinct layers:

\* An **infrastructure** layer comprising vppinfra, vlib, svm, and binary api libraries. See .../src/{**vppinfra**, **vlib**, **vlibapi**, **vlibmemory**, **svm** }

\* A generic network **stack** layer: vnet. See .../src/vnet

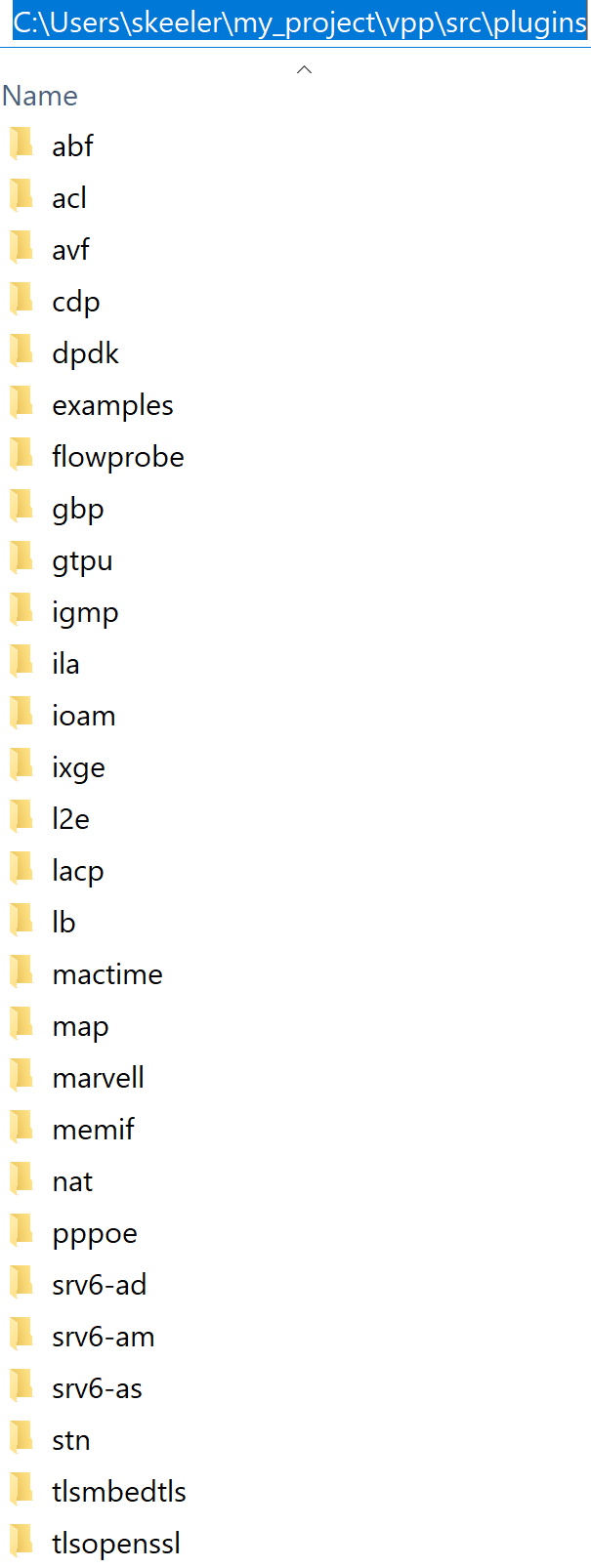
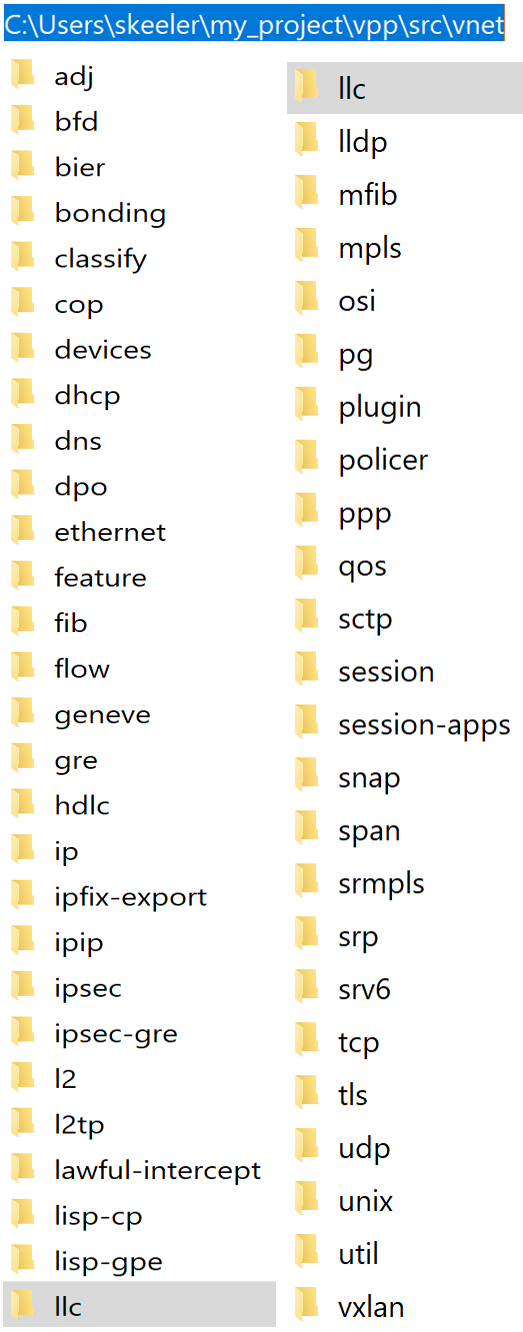
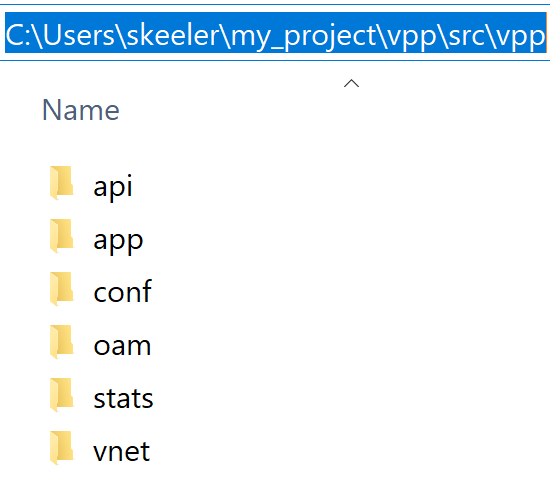
\* An **application shell**: vpp. See .../src/vpp

\* An increasingly rich set of **data-plane plugins**: see .../src/plugins

It's important to understand each of these layers in a certain amount

of detail. Much of the implementation is best dealt with at the API

level and otherwise left alone.

**< data-plane plugins**

**< Network STACK layer**:

**Application shell**:

# Infrastructure

\*\*\*\*\*\*\*\*\*\*\*\*\*\*

The files for this function are located in the **…/src/vnet** folder. The Infrastructure layer contains the following sections:

\* vppinfra

\* vlib

\* vlibapi

\* vlibmemory

\* svm

## Vppinfra

========

Vppinfra section is a collection of basic c-library services, quite sufficient

to build standalone programs to run directly on bare metal. It also

provides high-performance dynamic arrays, hashes, bitmaps,

high-precision real-time clock support, fine-grained event-logging,

and data structure serialization.

One fair comment / fair warning about vppinfra: you can't always tell

a macro from an inline function from an ordinary function simply by

name. Macros are used to avoid function calls in the typical case, and

to cause (intentional) side-effects.

Vppinfra has been around for almost 20 years and tends not to change frequently.

### Vectors

+++++++

Vppinfra vectors are ubiquitous dynamically resized arrays with by

user defined "headers". Many vpppinfra data structures (e.g. hash,

heap, pool) are vectors with various different headers.

The memory layout looks like this::

User header (optional, uword aligned)

Alignment padding (if needed)

Vector length in elements

User's pointer -> Vector element 0

Vector element 1

...

Vector element N-1

As shown above, the vector APIs deal with pointers to the 0th element

of a vector. Null pointers are valid vectors of length zero.

To avoid thrashing the memory allocator, one often resets the length

of a vector to zero while retaining the memory allocation. Set the

vector length field to zero via the vec\_reset\_length(v) macro. [Use

the macro! It's smart about NULL pointers.]

Typically, the user header is not present. User headers allow for

other data structures to be built atop vppinfra vectors. Users may

specify the alignment for data elements via the vec\_\*\_aligned macros.

Vectors elements can be any C type e.g. (int, double, struct

bar). This is also true for data types built atop vectors (e.g. heap,

pool, etc.). Many macros have \_a variants supporting alignment of

vector data and \_h variants supporting non-zero-length vector

headers. The \_ha variants support both.

Inconsistent usage of header and/or alignment related macro variants

will cause delayed, confusing failures.

Standard programming error: memorize a pointer to the ith element of a

vector, and then expand the vector. Vectors expand by 3/2, so such

code may appear to work for a period of time. Correct code almost

always memorizes vector \*\*indices\*\* which are invariant across

reallocations.

In typical application images, one supplies a set of global functions

designed to be called from gdb. Here are a few examples:

\* vl(v) - prints vec\_len(v)

\* pe(p) - prints pool\_elts(p)

\* pifi(p, index) - prints pool\_is\_free\_index(p, index)

\* debug\_hex\_bytes (p, nbytes) - hex memory dump nbytes starting at p

Use the "show gdb" debug CLI command to print the current set.

### Bitmaps

+++++++

Vppinfra bitmaps are dynamic, built using the vppinfra vector

APIs. Quite handy for a variety jobs.

### Pools

+++++

Vppinfra pools combine vectors and bitmaps to rapidly allocate and

free fixed-size data structures with independent lifetimes. Pools are

perfect for allocating per-session structures.

### Hashes

++++++

Vppinfra provides several hash flavors. Data plane problems involving

packet classification / session lookup often use

.../src/vppinfra/bihash\_template.[ch] bounded-index extensible

hashes. These templates are instantiated multiple times, to

efficiently service different fixed-key sizes.

Bihashes are thread-safe. Read-locking is not required. A simple

spin-lock ensures that only one thread writes an entry at a time.

The original vppinfra hash implementation in .../src/vppinfra/hash.[ch] are simple to use, and are often used in control-plane code which needs exact-string-matching.

In either case, one almost always looks up a key in a hash table to

obtain an index in a related vector or pool. The APIs are simple

enough, but one must take care when using the unmanaged

arbitrary-sized key variant. Hash\_set\_mem (hash\_table, key\_pointer,

value) memorizes key\_pointer. It is usually a bad mistake to pass the

address of a vector element as the second argument to hash\_set\_mem. It

is perfectly fine to memorize constant string addresses in the text

segment.

### Format

++++++

Vppinfra format is roughly equivalent to printf.

Format has a few properties worth mentioning. Format's first argument

is a (u8 \\*) vector to which it appends the result of the current

format operation. Chaining calls is very easy::

u8 \* result;

result = format (0, "junk = %d, ", junk);

result = format (result, "more junk = %d\n", more\_junk);

As previously noted, NULL pointers are perfectly proper 0-length

vectors. Format returns a (u8 \\*) vector, \*\*not\*\* a C-string. If you

wish to print a (u8 \\*) vector, use the "%v" format string. If you need

a (u8 \\*) vector which is also a proper C-string, either of these

schemes may be used::

vec\_add1 (result, 0)

or

result = format (result, "<whatever>%c", 0);

Remember to vec\_free() the result if appropriate. Be careful not to

pass format an uninitialized u8 \\*.

Format implements a particularly handy user-format scheme via the "%U"

format specification. For example::

u8 \* format\_junk (u8 \* s, va\_list \*va)

{

junk = va\_arg (va, u32);

s = format (s, "%s", junk);

return s;

}

result = format (0, "junk = %U, format\_junk, "This is some junk");

format\_junk() can invoke other user-format functions if desired. The

programmer shoulders responsibility for argument type-checking. It is

typical for user format functions to blow up if the va\_arg(va, <type>)

macros don't match the caller's idea of reality.

### Unformat

++++++++

Vppinfra unformat is vaguely related to scanf, but considerably more general.

A typical use case involves initializing an unformat\_input\_t from

either a C-string or a (u8 \\*) vector, then parsing via unformat() as

follows::

unformat\_input\_t input;

unformat\_init\_string (&input, "<some-C-string>");

/\* or \*/

unformat\_init\_vector (&input, <u8-vector>);

Then loop parsing individual elements::

while (unformat\_check\_input (&input) != UNFORMAT\_END\_OF\_INPUT)

{

if (unformat (&input, "value1 %d", &value1))

;/\* unformat sets value1 \*/

else if (unformat (&input, "value2 %d", &value2)

;/\* unformat sets value2 \*/

else

return clib\_error\_return (0, "unknown input '%U'", format\_unformat\_error,

input);

}

As with format, unformat implements a user-unformat function

capability via a "%U" user unformat function scheme.

### Vppinfra errors and warnings

+++++++++++++++++++++++++++

Many functions within the vpp dataplane have return-values of type

clib\_error\_t \\*. Clib\_error\_t'ss are arbitrary strings with a

bit of metadata [fatal, warning] and are easy to announce. Returning

a NULL clib\_error\_t \\* indicates "A-OK, no error."

Clib\_warning(<format-args>) is a handy way to add debugging output;

clib warnings prepend function:line info to unambiguously locate the

message source. Clib\_unix\_warning() adds perror()-style Linux

system-call information. In production images, clib\_warnings result in

syslog entries.

### Serialization

+++++++++++++

Vppinfra serialization support allows the programmer to easily serialize and unserialize complex data structures.

The underlying primitive serialize/unserialize functions use network

byte-order, so there are no structural issues serializing on a

little-endian host and unserializing on a big-endian host.

### Event-logger, graphical event log viewer

++++++++++++++++++++++++++++++

The vppinfra event logger provides very lightweight (sub-100ns)

precisely time-stamped event-logging services. See

.../src/vppinfra/{elog.c, elog.h}

Serialization support makes it easy to save and ultimately to combine

a set of event logs. In a distributed system running NTP over a local

LAN, we find that event logs collected from multiple system elements

can be combined with a temporal uncertainty no worse than 50us.

A typical event definition and logging call looks like this::

ELOG\_TYPE\_DECLARE (e) =

{

.format = "tx-msg: stream %d local seq %d attempt %d",

.format\_args = "i4i4i4",

};

struct { u32 stream\_id, local\_sequence, retry\_count; } \* ed;

ed = ELOG\_DATA (m->elog\_main, e);

ed->stream\_id = stream\_id;

ed->local\_sequence = local\_sequence;

ed->retry\_count = retry\_count;

The ELOG\_DATA macro returns a pointer to 20 bytes worth of arbitrary

event data, to be formatted (offline, not at runtime) as described by

format\_args. Aside from obvious integer formats, the CLIB event logger

provides a couple of interesting additions. The "t4" format

pretty-prints enumerated values::

ELOG\_TYPE\_DECLARE (e) =

{

.format = "get\_or\_create: %s",

.format\_args = "t4",

.n\_enum\_strings = 2,

.enum\_strings = { "old", "new", },

};

The "t" format specifier indicates that the corresponding datum is an

index in the event's set of enumerated strings, as shown in the

previous event type definition.

The “T” format specifier indicates that the corresponding datum is an

index in the event log’s string heap. This allows the programmer to

emit arbitrary formatted strings. One often combines this facility

with a hash table to keep the event-log string heap from growing

arbitrarily large.

Noting the 20-octet limit per-log-entry data field, the event log

formatter supports arbitrary combinations of these data types. As in:

the ".format" field may contain one or more instances of the

following:

\* i1 - 8-bit unsigned integer

\* i2 - 16-bit unsigned integer

\* i4 - 32-bit unsigned integer

\* i8 - 64-bit unsigned integer

\* f4 - float

\* f8 - double

\* s - NULL-terminated string - be careful

\* sN - N-byte character array

\* t1,2,4 - per-event enumeration ID

\* T4 - Event-log string table offset

The vpp engine event log is thread-safe, and is shared by all

threads. Take care not to serialize the computation. Although the

event-logger is about as fast as practicable, it's not appropriate for

per-packet use in hard-core data plane code. It's most appropriate for

capturing rare events - link up-down events, specific control-plane

events and so forth.

The vpp engine has several debug CLI commands for manipulating its event log::

vpp# event-logger clear

vpp# event-logger save <filename> # for security, writes into /tmp/<filename>.

# <filename> must not contain '.' or '/' characters

vpp# show event-logger [all] [<nnn>] # display the event log

# by default, the last 250 entries

The event log defaults to 128K entries. The command-line argument

"... vlib { elog-events <nnn> }" configures the size of the event log.

As described above, the vpp engine event log is thread-safe and

shared. To avoid confusing non-appearance of events logged by worker

threads, make sure to code &vlib\_global\_main.elog\_main - instead of

&vm->elog\_main. The latter form is correct in the main thread, but

will almost certainly produce bad results in worker threads.

## Vlib, Vlibapi, Vlibmemory

===================

Vlib and associated libraries (vlibapi, vlibmemory) provides vector processing support including graph-node

scheduling, reliable multicast support, ultra-lightweight cooperative

multi-tasking threads, a CLI, plug in .DLL support, physical memory and Linux epoll support. Parts of this library embody US Patent 7,961,636.

### G2 graphical event viewer

++++++++++++++++++++

The g2 graphical event viewer can display serialized vppinfra event

logs directly, or via the c2cpel tool.

*.. note::*

***Todo: please convert wiki page and figures***

### Init function discovery

++++++++++++++++

vlib applications register for various [initialization] events by

placing structures and \_\_attribute\_\_((constructor)) functions into the

image. At appropriate times, the vlib framework walks

constructor-generated singly-linked structure lists, calling the

indicated functions. vlib applications create graph nodes, add CLI

functions, start cooperative multi-tasking threads, etc. etc. using

this mechanism.

vlib applications invariably include a number of VLIB\_INIT\_FUNCTION

(my\_init\_function) macros.

Each init / configure / etc. function has the return type clib\_error\_t

\\*. Make sure that the function returns 0 if all is well, otherwise

the framework will announce an error and exit.

vlib applications must link against vppinfra, and often link against

other libraries such as VNET. In the latter case, it may be necessary

to explicitly reference symbol(s) otherwise large portions of the

library may be AWOL at runtime.

### Node Graph Initialization

-------------------------

vlib packet-processing applications invariably define a set of graph

nodes to process packets.

One constructs a vlib\_node\_registration\_t, most often via the

VLIB\_REGISTER\_NODE macro. At runtime, the framework processes the set

of such registrations into a directed graph. It is easy enough to add

nodes to the graph at runtime. The framework does not support removing

nodes.

vlib provides several types of vector-processing graph nodes,

primarily to control framework dispatch behaviors. The type member of

the vlib\_node\_registration\_t functions as follows:

\* VLIB\_NODE\_TYPE\_PRE\_INPUT - run before all other node types

\* VLIB\_NODE\_TYPE\_INPUT - run as often as possible, after pre\_input nodes

\* VLIB\_NODE\_TYPE\_INTERNAL - only when explicitly made runnable by adding pending frames for processing

\* VLIB\_NODE\_TYPE\_PROCESS - only when explicitly made runnable. "Process" nodes are actually cooperative multi-tasking threads. They \*\*must\*\* explicitly suspend after a reasonably short period of time.

For a precise understanding of the graph node dispatcher, please read

.../src/vlib/main.c:vlib\_main\_loop.

### Graph node dispatcher

---------------------

Vlib\_main\_loop() dispatches graph nodes. The basic vector processing

algorithm is diabolically simple, but may not be obvious from even a

long stare at the code. Here's how it works: some input node, or

set of input nodes, produce a vector of work to process. The graph

node dispatcher pushes the work vector through the directed graph,

subdividing it as needed, until the original work vector has been

completely processed. At that point, the process recurs.

This scheme yields a stable equilibrium in frame size, by

construction. Here's why: as the frame size increases, the

per-frame-element processing time decreases. There are several related

forces at work; the simplest to describe is the effect of vector

processing on the CPU L1 I-cache. The first frame element [packet]

processed by a given node warms up the node dispatch function in the

L1 I-cache. All subsequent frame elements profit. As we increase the

number of frame elements, the cost per element goes down.

Under light load, it is a crazy waste of CPU cycles to run the graph

node dispatcher flat-out. So, the graph node dispatcher arranges to

wait for work by sitting in a timed epoll wait if the prevailing frame

size is low. The scheme has a certain amount of hysteresis to avoid

constantly toggling back and forth between interrupt and polling

mode. Although the graph dispatcher supports interrupt and polling

modes, our current default device drivers do not.

The graph node scheduler uses a hierarchical timer wheel to reschedule

process nodes upon timer expiration.

### Process / thread model

----------------------

vlib provides an ultra-lightweight cooperative multi-tasking thread

model. The graph node scheduler invokes these processes in much the

same way as traditional vector-processing run-to-completion graph

nodes; plus-or-minus a setjmp/longjmp pair required to switch

stacks. Simply set the vlib\_node\_registration\_t type field to

vlib\_NODE\_TYPE\_PROCESS. Yes, process is a misnomer. These are

cooperative multi-tasking threads.

As of this writing, the default stack size is 2<<15; 32kb. Initialize

the node registration's process\_log2\_n\_stack\_bytes member as

needed. The graph node dispatcher makes some effort to detect stack

overrun, e.g. by mapping a no-access page below each thread stack.

Process node dispatch functions are expected to be "while(1) { }" loops

which suspend when not otherwise occupied, and which must not run for

unreasonably long periods of time.

"Unreasonably long" is an application-dependent concept. Over the

years, we have constructed frame-size sensitive control-plane nodes

which will use a much higher fraction of the available CPU bandwidth

when the frame size is low. The classic example: modifying forwarding

tables. So long as the table-builder leaves the forwarding tables in a

valid state, one can suspend the table builder to avoid dropping

packets as a result of control-plane activity.

Process nodes can suspend for fixed amounts of time, or until another

entity signals an event, or both. See the next section for a

description of the vlib process event mechanism.

When running in vlib process context, one must pay strict attention to

loop invariant issues. If one walks a data structure and calls a

function which may suspend, one had best know by construction that it

cannot change. Often, it's best to simply make a snapshot copy of a

data structure, walk the copy at leisure, then free the copy.

### Process events???

--------------

The vlib process event mechanism API is extremely lightweight and easy

to use. Here is a typical example::

vlib\_main\_t \*vm = &vlib\_global\_main;

uword event\_type, \* event\_data = 0;

while (1)

{

vlib\_process\_wait\_for\_event\_or\_clock (vm, 5.0 /\* seconds \*/);

event\_type = vlib\_process\_get\_events (vm, &event\_data);

switch (event\_type) {

case EVENT1:

handle\_event1s (event\_data);

break;

case EVENT2:

handle\_event2s (event\_data);

break;

case ~0: /\* 5-second idle/periodic \*/

handle\_idle ();

break;

default: /\* bug! \*/

ASSERT (0);

}

vec\_reset\_length(event\_data);

}

In this example, the VLIB process node waits for an event to occur, or

for 5 seconds to elapse. The code demuxes on the event type, calling

the appropriate handler function. Each call to vlib\_process\_get\_events

returns a vector of per-event-type data passed to successive

vlib\_process\_signal\_event calls; vec\_len (event\_data) >= 1.

It is an error to process only event\_data[0].

Resetting the event\_data vector-length to 0 [instead of calling

vec\_free] means that the event scheme doesn't burn cycles continuously

allocating and freeing the event data vector. This is a common

vppinfra / vlib coding pattern, well worth using when appropriate.

Signaling an event is easy, for example::

vlib\_process\_signal\_event (vm, process\_node\_index, EVENT1,

(uword)arbitrary\_event1\_data); /\* and so forth \*/

One can either know the process node index by construction - dig it

out of the appropriate vlib\_node\_registration\_t - or by finding the

vlib\_node\_t with vlib\_get\_node\_by\_name(...).

### Buffers

-------

vlib buffering solves the usual set of packet-processing problems,

albeit at high performance. Key in terms of performance: one

ordinarily allocates / frees N buffers at a time rather than one at a

time. Except when operating directly on a specific buffer, one deals

with buffers by index, not by pointer.

Packet-processing frames are effectively u32[], not

vlib\_buffer\_t[].

Packets comprise one or more vlib buffers, chained together as

required. Multiple particle sizes are supported; hardware input nodes

simply ask for the required size(s). Coalescing support is

available. For obvious reasons one is discouraged from writing one's

own wild and wacky buffer chain traversal code.

vlib buffer headers are allocated immediately prior to the buffer data

area. In typical packet processing this saves a dependent read wait:

given a buffer's address, one can prefetch the buffer header

[metadata] at the same time as the first cache line of buffer data.

Buffer header metadata (vlib\_buffer\_t) includes the usual rewrite

expansion space, a current\_data offset, RX and TX interface indices,

packet trace information, and a opaque areas.

The opaque data is intended to control packet processing in arbitrary

subgraph-dependent ways. The programmer shoulders responsibility for

data lifetime analysis, type-checking, etc.

Buffers have reference-counts in support of e.g. multicast

replication.

### Shared-memory message API

-------------------------

Local control-plane and application processes interact with the vpp

dataplane via asynchronous message-passing in shared memory over

unidirectional queues. The same application APIs are available via

sockets.

Capturing API traces and replaying them in a

simulation environment requires a disciplined approach

to the problem. This seems like a make-work task, but it is not. When

something goes wrong in the control-plane after 300,000 or 3,000,000

operations, high-speed replay of the events leading up to the accident

is a huge win.

The shared-memory message API message allocator vl\_api\_msg\_alloc uses

a particularly cute trick. Since messages are processed in order, we

try to allocate message buffering from a set of fixed-size,

preallocated rings. Each ring item has a "busy" bit. Freeing one of

the preallocated message buffers merely requires the message consumer

to clear the busy bit. No locking required.

## SVM

SVM performs the shared VM allocation, defines the mmap page size, and works with the mapping.

# Network Stack Layer (src/vnet)

--------

The files for this function are located in the **…/src/vnet** folder.

## Vnet

====

The vnet library provides vectorized layer-2 and 3 networking graph

nodes, a packet generator, and a packet tracer.

In terms of building a packet processing application, vnet provides a

platform-independent subgraph to which one connects a couple of

device-driver nodes.

Typical RX connections include "ethernet-input" [full software

classification, feeds ipv4-input, ipv6-input, arp-input etc.] and

"ipv4-input-no-checksum" [if hardware can classify, perform ipv4

header checksum].

### Effective graph dispatch function coding

----------------------------------------

Over the 15 years, two distinct styles have emerged: a

single/dual/quad loop coding model and a fully-pipelined coding

model. We seldom use the fully-pipelined coding model, so we won't

describe it in any detail.

### Single/dual loops

-----------------

The single/dual/quad loop model is the only way to conveniently solve

problems where the number of items to process is not known in advance:

typical hardware RX-ring processing. This coding style is also very

effective when a given node will not need to cover a complex set of

dependent reads.

# Application Shell (src/vpp)

--------

The files for this function are located in the …/ src/VPP folder.

# Plug-ins (src/plugins)

--------

vlib implements a straightforward plug-in DLL

mechanism. VLIB client applications specify a directory to search for

plug-in .DLLs, and a name filter to apply (if desired). VLIB needs to

load plug-ins very early.

Once loaded, the plug-in DLL mechanism uses dlsym to find and verify

a vlib\_plugin\_registration data structure in the newly-loaded

plug-in.

### Debug CLI

---------

Adding debug CLI commands to VLIB applications is very simple.

Here is a complete example::

static clib\_error\_t \*

show\_ip\_tuple\_match (vlib\_main\_t \* vm,

unformat\_input\_t \* input,

vlib\_cli\_command\_t \* cmd)

{

vlib\_cli\_output (vm, "%U\n", format\_ip\_tuple\_match\_tables, &routing\_main);

return 0;

}

static VLIB\_CLI\_COMMAND (show\_ip\_tuple\_command) = {

.path = "show ip tuple match",

.short\_help = "Show ip 5-tuple match-and-broadcast tables",

.function = show\_ip\_tuple\_match,

};

This example implements the "show ip tuple match" debug cli

command. In ordinary usage, the vlib cli is available via the "vppctl"

applicationn, which sends traffic to a named pipe. One can configure

debug CLI telnet access on a configurable port.

The cli implementation has an output redirection facility which makes

it simple to deliver cli output via shared-memory API messaging,

Particularly for debug or "show tech support" type commands, it would

be wasteful to write vlib application code to pack binary data, write

more code elsewhere to unpack the data and finally print the

answer. If a certain cli command has the potential to hurt packet

processing performance by running for too long, do the work

incrementally in a process node. The client can wait.

### Packet tracer

-------------

Vlib includes a frame element [packet] trace facility, with a simple

vlib cli interface. The cli is straightforward: "trace add

<input-node-name> <count>".

To trace 100 packets on a typical x86\_64 system running the dpdk

plugin: "trace add dpdk-input 100". When using the packet generator:

"trace add pg-input 100"

Each graph node has the opportunity to capture its own trace data. It

is almost always a good idea to do so. The trace capture APIs are

simple.

The packet capture APIs snapshoot binary data, to minimize processing

at capture time. Each participating graph node initialization provides

a vppinfra format-style user function to pretty-print data when required

by the VLIB "show trace" command.

Set the VLIB node registration ".format\_trace" member to the name of the per-graph node format function.

Here's a simple example::

u8 \* my\_node\_format\_trace (u8 \* s, va\_list \* args)

{

vlib\_main\_t \* vm = va\_arg (\*args, vlib\_main\_t \*);

vlib\_node\_t \* node = va\_arg (\*args, vlib\_node\_t \*);

my\_node\_trace\_t \* t = va\_arg (\*args, my\_trace\_t \*);

s = format (s, "My trace data was: %d", t-><whatever>);

return s;

}

The trace framework hands the per-node format function the data it

captured as the packet whizzed by. The format function pretty-prints

the data as desired.