

IP Instances Design Document

PSARC 2006/366

2006-12-21

Erik Nordmark

1 Background material

Without reading the background material this document doesn't make much sense!

See si-interfaces.pdf for instance in

<http://www.opensolaris.org/os/project/crossbow/Docs/si-interfaces.pdf>

Table of Contents

1Background material.....	1
2Code changes.....	1
3Internal TCP/IP changes.....	2
4Design Methodology.....	2
5Details for different modules and subsystems.....	3
5.1ARP.....	3
5.2ICMP.....	4
5.3UDP.....	5
5.4IP.....	5
5.5TCP.....	6
5.6SCTP.....	6
5.7IP Filter.....	6
5.8IPsec.....	7
5.9STREAM head.....	13
6netstack framework design.....	14
7zone admin design changes.....	14
7.1zone_create needing a flags argument for “exclusive”.....	14
7.2zone_ifname pieces.....	14
7.3zonecfg, zoneadm changes.....	15
8Threat Analysis.....	15
9Design Alternatives.....	15
10Appendix A: example of TCP/IP module changes.....	16
10.1Defining a icmp_stack_t.....	16
10.2Having the init/fini routines use netstack_register/unregister.....	16
10.3Introduce rawip_stack_init() and rawip_stack_fini().....	16
10.4Adding icmp_stack_t pointer.....	17
10.5icmp_open using netstack_find_by_cred().....	17
10.6Using kstat_create_netstack and kstat_destroy_netstack.....	18

11 Appendix B: large foo_stack_t structures.....	18
11.1 ip_stack_t.....	19
11.2 tcp_stack_t.....	24
11.3 sctp_stack_t.....	26
11.4 ipf_stack_t.....	28

2 Code changes

The code changes can be accessed at

<file:///net/nptbld-x.sfbay/disk1/nordmark/si-review/webrev.orig/categories.html>

and at

<http://cr.grommit.com/~nordmark/ip-instances/webrev.orig/categories.html>

3 Internal TCP/IP changes

The general structure of the changes to all the TCP/IP kernel modules consist of

- Defining a foo_stack_t that contains all the previously global data that is modified
- Having the init and fini routines call netstack_register() and netstack_unregister(), respectively
- Writing foo_stack_init() and foo_stack_fini() routines which allocate and initiate (and free) an instance of a foo_stack_t
- Making the per-instance (normally per-queue i.e. referenced by q_ptr) structure have a pointer to the foo_stack_t
- Making the foo_open() routine use netstack_find_by_cred() to get the pointer to the netstack_t and as a result a pointer to the foo_stack_t, so that this can be saved in the per-instance structure
- For modules that use kstat_create, modify them to use kstat_create_netstack() and kstat_delete_netstack() which ensures that the statistics are visible in the zones that use that IP instance.

All the modules follow the same pattern. The specifics for the different modules are covered below.

An example of the above changes for icmp.c are shown in Appendix A.

4 Design Methodology

The idea is to ensure complete separation by no longer having any writable data structures in the TCP/IP set of modules.

The set of global data structures in these modules was determined using the symbol table; looking at all the OBJT symbols.

Manual inspection was then used to determine which ones are never modified (example, struct streamtab which is a statically allocation structure).

The replacement of the strings that represent the previously global data (whether corresponding to symbols themselves, or being #defines for ndd variables) was done semi-automatically (sed(1) is your friend!). The replacements are of the form

```
FROM      ip_g_forward      TO      ipst->ips_ip_g_forward
```

FROM ip_mib TO ipst->ips_ip_mib

5 Details for different modules and subsystems

5.1 Netstack framework and reference counting

In order to insulate the various IP kernel modules from the relationship between zones and IP instances, we have added the netstack framework. The IP kernel modules register with this framework and as a result get callbacks when IP instances are added or removed.

The netstack framework in turn uses the `zone_key_create()` callbacks to be told when zones are added or removed.

Each `netstack_t` tracks how many zones use it, and also how many explicit references there are on the `netstack_t`. The `netstack_hold()/netstack_rele()` functions are used for explicit handling of the reference counts. And the various lookup routines (`netstack_find_by_xx()`) return with a reference on the `netstack_t`.

The IP kernel modules typically have a reference on the `netstack_t` for each open stream, but not for finer grain structures. For example the IP modules have a reference for each `conn_t` and each `ill_t`, since these correspond to an open stream. But there are other data structures inside IP, such as `ire_t`, that have pointers to the `netstack_t` (either directly, or in this example by having a pointer to the `ip_stack_t`), that do not hold references on the `netstack_t`. These data structures all need to be discarded before the `ill_t` etc can be discarded, hence the existing mechanism for doing that ensures that they are gone before the `netstack_t` is removed.

There is asynchronous behavior in the IP kernel modules when it comes to discarding state. For instance, TCP wants to hold on to a `tcp_t` in `TIME_WAIT` state for some time. We allow that behavior even when the reason for closing the TCP connection is because a zone is halted and as a result the IP instance is being removed. This is handled without having the `zone_destroy` wait for 2 minutes! The `zsd_destroy()` callback merely makes the `netstack_t` unavailable for lookups and does one `netstack_rele()`. Then when the final `netstack_rele()` takes place, `netstack_stack_inactive()` is called which ends up calling all the destroy callbacks for the IP instances.

In principle the management of the `netstack_hold/rele` is done by `ipcl_conn_create()` and `ipcl_conn_destroy()`. But due to TCPs and SCTPs handling of eager connections, there is some explicit logic to handle those cases in those modules.

5.2 Use of zoneid inside the TCP/IP modules

In the shared-IP instance the zoneid is used as a discriminant. For instance, source address selection, the fanout of inbound packets to connections, as well as some routing lookups are a function of the zoneid.

However, for an exclusive-IP instance, there is no need for such discrimination since we want IP to function as if it was in the global zone. For this reason we make the zoneid (in the `conn_t`, `ire_t`, `ipif_t`, etc) be the global zoneid in the case of an exclusive-IP.

Note that for TX (in `tnet.c` and `tn_ipopt.c`) we need to handle two different concepts:

- the local separation, which consists of a `netstack_t` (or `ip_stack_t`), and a zoneid to tell different zones apart for the shared-IP instance
- expressions of the label for the remote end, which is represented in the form of a zoneid.

We use different variable names (`zoneid` vs. `ip_zoneid`) to keep those apart.

5.3 ARP

The module is a straight application of then methodology above.

The ar_stack_t is defined in

file:///net/nptbld-x.sfbay/disk1/nordmark/si-only/webrev/usr/src/uts/common/inet/arp_impl.h.html

and has

```
166  /*
167   * ARP stack instances
168   */
169  struct arp_stack {
170      netstack_t      *as_netstack;    /* Common netstack */
171      void            *as_head;        /* AR Instance Data List Head */
172      caddr_t         as_nd;          /* AR Named Dispatch Head */
173      struct arl_s    *as_arl_head;    /* ARL List Head */
174      arpparam_t      *as_param_arr;   /* ndd variable table */
175
176      /* ARP Cache Entry Hash Table */
177      ace_t           *as_ce_hash_tbl[ARP_HASH_SIZE];
178      ace_t           *as_ce_mask_entries; /* proto_mask not all ones */
179
180      int             as_snmp_hash_size;
181      /*
182       * With the introduction of netinfo (neti kernel module),
183       * it is now possible to access data structures in the ARP module
184       * without the code being executed in the context of the IP module,
185       * thus there is no locking being enforced through the use of STREAMS.
186       */
187      krwlock_t       as_arl_g_lock;
188      arl_t           *as_arl_g_head; /* ARL List Head */
189
190
191      uint32_t        as_arp_index_counter;
192      uint32_t        as_arp_counter_wrapped;
193
194      /* arp_net.c */
195      hook_family_t   as_arproot;
196
197      /*
198       * Hooks for ARP
199       */
200      hook_event_t    as_arp_physical_in_event;
201      hook_event_t    as_arp_physical_out_event;
202      hook_event_t    as_arp_nic_events;
203
204      hook_event_token_t as_arp_physical_in;
205      hook_event_token_t as_arp_physical_out;
206      hook_event_token_t as_arpnicevents;
207
208      net_data_t      as_arp;
209 };
210 typedef struct arp_stack arp_stack_t;
```

5.4 ICMP

ICMP is also a straight application of the methodology.

The `icmp_stack_t` is defined in

file:///net/nptbld-x.sfbay/disk1/nordmark/si-only/webrev/usr/src/uts/common/inet/rawip_impl.h.html

and has

```
56 /*
57  * ICMP stack instances
58  */
59 struct icmp_stack {
60     netstack_t      *is_netstack;    /* Common netstack */
61     void            *is_head;        /* Head for list of open icmps */
62     IDP             is_nd;           /* Points to table of ICMP ND variables. */
63     icmpparam_t     *is_param_arr;    /* ndd variable table */
64     kstat_t          *is_ksp;         /* kstats */
65     mib2_rawip_t    is_rawip_mib;    /* SNMP fixed size info */
66 };
67 typedef struct icmp_stack icmp_stack_t;
```

5.5 UDP

UDP is more or less a straight application. The only difference compared to arp and icmp is that udp uses `conn_t`, and as a result it uses `ipcl_conn_create()` and `ipcl_conn_destroy()`. Those routines take care of doing `netstack_hold` and `netstack_rele`, respectively. Since `udp_open` gets an implicit `netstack_hold` as part of calling `netstack_find_by_cred()`, `udp_open` needs to compensate for the hold in `ipcl_conn_create()`.

The `udp_stack_t` is defined in

file:///net/nptbld-x.sfbay/disk1/nordmark/si-only/webrev/usr/src/uts/common/inet/udp_impl.h.html

and has

```
130 /*
131  * UDP stack instances
132  */
133 struct udp_stack {
134     netstack_t      *us_netstack;    /* Common netstack */
135
136     uint_t          us_bind_fanout_size;
137     udp_fanout_t    *us_bind_fanout;
138
139     int             us_num_epriv_ports;
140     in_port_t       us_epriv_ports[UDP_NUM_EPRIV_PORTS];
141
142     /* Hint not protected by any lock */
143     in_port_t       us_next_port_to_try;
144
145     IDP             us_nd;    /* Points to table of UDP ND variables. */
146     udpparam_t      *us_param_arr; /* ndd variable table */
147
148     kstat_t         *us_mibkp;    /* kstats exporting mib data */
149     kstat_t         *us_kstat;
150     udp_stat_t      us_statistics;
151
152     mib2_udp_t      us_udp_mib;    /* SNMP fixed size info */
153
154 /*
155  * This controls the rate some ndd info report functions can be used
156  * by non-privileged users. It stores the last time such info is
157  * requested. When those report functions are called again, this
158  * is checked with the current time and compare with the ndd param
159  * udp_ndd_get_info_interval.
160  */
161     clock_t         us_last_ndd_get_info_time;
162
163 /*
164  * The smallest anonymous port in the privileged port range which UDP
165  * looks for free port. Use in the option UDP_ANONPRIVBIND.
166  */
167     in_port_t       us_min_anonpriv_port;
168
169 };
170 typedef struct udp_stack udp_stack_t
```

5.6 IP

The `ip_stack_t` is defined in

file:///net/nptbld-x.sfbay/disk1/nordmark/si-only/webrev/usr/src/uts/common/inet/ip_stack.h.html

and is too large to fit on this page. See Appendix B.

5.7 TCP

The `tcp_time_wait` related state is per-queue and not per IP instance. This is because the queues are not per IP instance – they are related to the number of CPUs on the system.

The TCP default queue for the global zone (shared IP) is created for `strplumb.c` as today. For the other IP instances the TCP default queue is created when first TCP endpoint (e.g., socket) is opened. It can not be created from `tcp_stack_init()` since at that time the zone isn't fully initialized in the kernel thus things like `zone_get_kcred()` will fail.

TCP has an added reference count to track how many `tcp_t`'s might use the global queue; in effect this has to count every `tcp_t` since a `tcp_t` can become detached and need the tcp global queue later. Prior to IP Instances there was no need to be able to remove the tcp global queue. But with IP Instances this happens as part of removing an IP instance (as a result of the exclusive-IP zone being halted). The reference count tells us when the tcp queue can be closed down.

The TCP `time_wait` connections are kept on a list per queue. Since there are not separate queues per separate IP instances, the specific statistics for the TCP `time_wait` handling has to be made global. The queues also have a free list (`tcp_free_list`). When a `conn_t/tcp_t` is placed on the free list it is disassociated from the IP instance. This is necessary in order for an instance be able to go away in a timely manner; having free list entries refer to the IP instance would make it unpredictable when the IP instance can be freed up.

The `tcp_stack_t` is defined in:

file:///net/nptbld-x.sfbay/disk1/nordmark/si-only/webrev/usr/src/uts/common/inet/tcp_stack.h.html

and is too large to fit on this page. See appendix B.

5.8 SCTP

The sctp default queue is handled the same way as the TCP default queue.

The `sctp_stack_t` is defined in:

file:///net/nptbld-x.sfbay/disk1/nordmark/si-only/webrev/usr/src/uts/common/inet/sctp/sctp_stack.h.html

and is too large to fit on this page. See appendix B.

5.9 tun module

The tunneling module has its separate stack structure.

```
265  /*
266   * tunnel stack instances
267   */
268  struct tun_stack {
269      netstack_t      *tuns_netstack; /* Common netstack */
270
271      /*
272       * protects global data structures such as tun_ppa_list
273       * also protects tun_t at ts_next and *ts_atp
274       * should be acquired before ts_lock
275       */
```

```

276         kmutex_t         tuns_global_lock;
277         tun_stats_t      *tuns_ppa_list[TUN_PPA_SZ];
278         tun_t            *tuns_byaddr_list[TUN_T_SZ];
279
280         ipaddr_t         tuns_relay_rtr_addr_v4;
281     };
282     typedef struct tun_stack tun_stack_t;

```

5.10 neti

The Packet Filter Hooks project introduced the netinfo provider.

The `neti_stack_t` is defined in

<file:///net/nptbld-x.sfbay/disk1/nordmark/si-only/webrev/usr/src/uts/common/sys/neti.h.html>

and has

```

164     /*
165     * neti stack instances
166     */
167     struct neti_stack {
168         krwlock_t nts_netlock;
169
170         /* list of net_data_t */
171         LIST_HEAD(netd_listhead, net_data) nts_netd_head;
172         netstack_t *nts_netstack;
173     };
174     typedef struct neti_stack neti_stack_t;

```

5.11 hook

The Packet Filter Hooks project introduced the tracking of hooks in the TCP/IP stack.

The `hook_stack_t` is defined in

<file:///net/nptbld-x.sfbay/disk1/nordmark/si-only/webrev/usr/src/uts/common/sys/hook.h.html>

and has

```

120     /*
121     * hook stack instances
122     */
123     struct hook_stack {
124         cvwaitlock_t hks_familylock;           /* global lock */
125         hook_family_int_head_t hks_familylist; /* family list head */
126         netstack_t *hk_netstack;
127     };

```

5.12 IP Filter

With the packet filtering hooks integrated and `pfil` module is removed, there is just the `ipf_stack_t` for IP Filter. (Previous versions of IP Instances had a `pfil_stack_t` as well.)

One thing that is special for the `ipf` code is that it can be compiled in the kernel to form a kernel module, and also as a user-level part of the `ipftest` binary. For the latter purpose there is a user-level allocation routine which allocates a single `ipf_stack_t`. That way all the `ipf` code base can have `ipf_stack_t` pointers as function arguments whether compiled for kernel or user-level usage. (The alternative would have been to have tons of `#ifdef KERNEL` in the code.)

The ipf_stack_t is defined in:

file:///net/nptbld-x.sfbay/disk1/nordmark/si-only/webrev/usr/src/common/ipf/ipf_stack.h.html

and is too large to fit on this page. See appendix B.

5.13 IPsec

IPsec consists of several kernel modules that are loaded at different points in time. Part of them are loaded on demand using special `ipsec_loader.c` code. `ipsec_loader` can be triggered by any IP instance, i.e. The global zone doesn't have to be the first to use IPsec. `ipsec_loader` triggers the plumbing operation for `keysock` for the shared IP instance. For other instances this is triggered by the first open of `keysock` in that instance.

The current IP instances design provides for the same separate loading, but as a result it has many different stack data structures: `ipsec_stack_t`, `ipsecah_stack_t`, `ipsecesp_stack_t`, `keysock_stack_t`, `spdsock_stack_t`. The reason for such fine granularity is that it isn't clear what pieces can be loaded and unloaded independently.

The `ipsec_in/out` messages have an additional `netstack_t` pointer as well as a `netstackid_t` integer. The pointer is used inside all of the IP/IPsec code, since it is all synchronous. The exception is when IPsec calls `kEF` and `kEf` queues the operation, since we have no upper bound on how long time it takes `kEF` to process the operation. (FWIW it isn't clear how we in S10 prevent the IPsec modules from being unloaded while there are pending asynch operations in `kEF`). For that reason we have the `netstackid_t`, and when we get the `kEF` callback (in `esp_kcf_callback()` and `ah_kcf_callback()`) we lookup the `netstackid_t` to make sure the `netstack` still exists.

`ipsec_stack_t` is defined in:

file:///net/nptbld-x.sfbay/disk1/nordmark/si-only/webrev/usr/src/uts/common/inet/ipsec_impl.h.html

and looks like this:

```
543 /*
544  * IPSEC stack instances
545  */
546 struct ipsec_stack {
547     netstack_t          *ipsec_netstack;
548
549     /* Packet dropper for IP IPsec processing failures */
550     ipdropper_t        ipsec_dropper;
551
552     /* From spd.c */
553     /*
554      * Policy rule index generator. We assume this won't wrap in the
555      * lifetime of a system. If we make 2^20 policy changes per second,
556      * this will last 2^44 seconds, or roughly 500,000 years, so we don't
557      * have to worry about reusing policy index values.
558      */
559     uint64_t            ipsec_next_policy_index;
560
561     HASH_HEAD(ipsec_action_s) ipsec_action_hash[IPSEC_ACTION_HASH_SIZE];
562     HASH_HEAD(ipsec_sel)      *ipsec_sel_hash;
563     uint32_t               ipsec_spd_hashsize;
564
565     ipsif_t               ipsec_ipsid_buckets[IPSID_HASHSIZE];
566
567     /*
568      * Active & Inactive system policy roots
569      */
570     ipsec_policy_head_t    ipsec_system_policy;
571     ipsec_policy_head_t    ipsec_inactive_policy;
```

```

572
573     /* Packet dropper for generic SPD drops. */
574     ipdropper_t         ipsec_spd_dropper;
575
576 /* ipdrop.c */
577     kstat_t             *ipsec_ip_drop_kstat;
578     struct ip_dropstats *ipsec_ip_drop_types;
579
580 /* spd.c */
581     /*
582     * Have a counter for every possible policy message in
583     * ipsec_policy_failure_msgs
584     */
585     uint32_t            ipsec_policy_failure_count[IPSEC_POLICY_MAX];
586     /* Time since last ipsec policy failure that printed a message. */
587     hrttime_t           ipsec_policy_failure_last;
588
589 /* ip_spd.c */
590     /* stats */
591     kstat_t             *ipsec_ksp;
592     struct ipsec_kstats_s *ipsec_kstats;
593
594 /* sadb.c */
595     /* Packet dropper for generic SADB drops. */
596     ipdropper_t         ipsec_sadb_dropper;
597
598 /* spd.c */
599     boolean_t           ipsec_inbound_v4_policy_present;
600     boolean_t           ipsec_outbound_v4_policy_present;
601     boolean_t           ipsec_inbound_v6_policy_present;
602     boolean_t           ipsec_outbound_v6_policy_present;
603
604 /* spd.c */
605     /*
606     * Because policy needs to know what algorithms are supported, keep the
607     * lists of algorithms here.
608     */
609     kmutex_t            ipsec_alg_lock;
610
611     uint8_t             ipsec_nalgs[IPSEC_NALGTYPES];
612     ipsec_alginfo_t     *ipsec_alglists[IPSEC_NALGTYPES][IPSEC_MAX_ALGS];
613
614     uint8_t             ipsec_sortlist[IPSEC_NALGTYPES][IPSEC_MAX_ALGS];
615
616     int                 ipsec_algs_exec_mode[IPSEC_NALGTYPES];
617 };
618
619 typedef struct ipsec_stack ipsec_stack_t;
620
621

```

ipsecah_stack_t is defined in:

<file:///net/nptbld-x.sfbay/disk1/nordmark/si-only/webrev/usr/src/uts/common/inet/ipsecah.h.html>

and contains:

```
90 /*
91  * IPSECAH stack instances
92  */
93 struct ipsecah_stack {
94     netstack_t          *ipsecah_netstack;      /* Common netstack */
95
96     caddr_t             ipsecah_g_nd;
97     ipsecahparam_t     *ipsecah_params;
98     kmutex_t           ipsecah_param_lock;     /* Protects params */
99
100    sadbp_t             ah_sadb;
101
102    /* Packet dropper for AH drops. */
103    ipdropper_t         ah_dropper;
104
105    kstat_t             *ah_ksp;
106    ah_kstats_t         *ah_kstats;
107
108    /*
109     * Keysock instance of AH.  There can be only one per stack instance.
110     * Use casptr() on this because I don't set it until KEYSOCK_HELLO
111     * comes down.
112     * Paired up with the ah_pfkey_q is the ah_event, which will age SAs.
113     */
114    queue_t             *ah_pfkey_q;
115    timeout_id_t        ah_event;
116
117    mblk_t              *ah_ip_unbind;
118 };
119 typedef struct ipsecah_stack ipsecah_stack_t;
```

The ipsecesp_stack_t is defined in:

<file:///net/nptbld-x.sfbay/disk1/nordmark/si-only/webrev/usr/src/uts/common/inet/ipsecesp.h.html>

and contains:

```
45 /*
46  * IPSECESP stack instances
47  */
48 struct ipsecesp_stack {
49     netstack_t          *ipsecesp_netstack;    /* Common netstack */
50
51     caddr_t             ipsecesp_g_nd;
52     struct ipsecespparam_s *ipsecesp_params;
53     kmutex_t           ipsecesp_param_lock;    /* Protects params */
54
55     /* Packet dropper for ESP drops. */
56     ipdropper_t        esp_dropper;
57
58     kstat_t             *esp_ksp;
59     struct esp_kstats_s *esp_kstats;
60
61     /*
62      * Keysock instance of ESP.  There can be only one per stack instance.
63      * Use casptr() on this because I don't set it until KEYSOCK_HELLO
64      * comes down.
65      * Paired up with the esp_pfkey_q is the esp_event, which will age SAs.
66      */
67     queue_t             *esp_pfkey_q;
68     timeout_id_t        esp_event;
69
70     mblk_t              *esp_ip_unbind;
71
72     sadbp_t             esp_sadb;
73
74 };
75 typedef struct ipsecesp_stack ipsecesp_stack_t;
```

The `keysock_stack_t` is defined in:

<file:///net/nptbld-x.sfbay/disk1/nordmark/si-only/webrev/usr/src/uts/common/inet/keysock.h.html>

and contains:

```
48 /*
49  * KEYSOCK stack instances
50  */
51 struct keysock_stack {
52     netstack_t          *keystack_netstack;    /* Common netstack */
53     /*
54      * keysock_plumbed: zero if plumb not attempted, positive if it
55      * succeeded, negative if it failed.
56      */
57     int                 keystack_plumbed;
58     boolean_t          keystack_plumb_inprogress;
59     caddr_t            keystack_g_nd;
60     struct keysockparam_s *keystack_params;
61
62     kmutex_t           keystack_param_lock;
63                       /* Protects the NDD variables. */
64
65     /* List of open PF_KEY sockets, protected by keysock_list_lock. */
66     kmutex_t           keystack_list_lock;
67     struct keysock_s   *keystack_list;
68
69     /*
70      * Consumers table. If an entry is NULL, keysock maintains
71      * the table.
72      */
73     kmutex_t           keystack_consumers_lock;
74
75 #define KEYSOCK_MAX_CONSUMERS 256
76     struct keysock_consumer_s *keystack_consumers[KEYSOCK_MAX_CONSUMERS];
77
78     /*
79      * State for flush/dump. This would normally be a boolean_t, but
80      * cas32() works best for a known 32-bit quantity.
81      */
82     uint32_t           keystack_flushdump;
83     int                keystack_flushdump_errno;
84
85     /*
86      * This integer counts the number of extended REGISTERed sockets. This
87      * determines if we should send extended REGISTERs.
88      */
89     uint32_t           keystack_num_extended;
90
91     /*
92      * Global sequence space for SADB_ACQUIRE messages of any sort.
93      */
94     uint32_t           keystack_acquire_seq;
95 };
96 typedef struct keysock_stack keysock_stack_t;
```

The `spdssock_stack_t` is defined in:

<file:///net/nptbld-x.sfbay/disk1/nordmark/si-only/webrev/usr/src/uts/common/inet/spdssock.h.html>

and contains:

```
37 /*
38  * SPDSOCK stack instances
39  */
40 struct spd_stack {
41     netstack_t             *spds_netstack; /* Common netstack */
42
43     caddr_t                spds_g_nd;
44     struct spdssockparam_s *spds_params;
45     kmutex_t               spds_param_lock;
46     /* Protects the NDD variables. */
47
48     /*
49      * To save algorithm update messages that are processed only after
50      * IPsec is loaded.
51      */
52     #define SPD_EXT_MAX          9
53     struct spd_ext          *spds_extv_algs[SPD_EXT_MAX + 1];
54     mblk_t                  *spds_mp_algs;
55     boolean_t               spds_algs_pending;
56     struct ipsec_alginfo
57         *spds_algs[IPSEC_NALGTYPES][IPSEC_MAX_ALGS];
58     int                     spds_algs_exec_mode[IPSEC_NALGTYPES];
59     kmutex_t                spds_alg_lock;
60 };
61 typedef struct spd_stack spd_stack_t;
```

5.14 STREAM head

The stream head has a separate `str_stack_t`.

The reason for this is that we need to track things like the `I_PLINK` in order to be able to have the kernel undo all the “ifconfig plumb” operations that was done when the zone booted.

This “unplumbing” is driven from the `zsd` callback when the zone is shutdown so that those streams/queues are closed in IP (and the data structures cleaned up) before we get to the `zsd destroy` callback. Thus the `str_stack_shutdown()` callback is called from the netstack framework.

The `str_stack_t` also contains the autopush information to allow separate autopush configuration for different IP instances.

The project introduces some additional checks related to STREAMS anchors. The thread removing an anchor must have the same zoneid as the thread that created it. In addition, STREAMS anchors as well as `_I_INSERT` and `_I_REMOVE` require `PRIV_SYS_NET_CONFIG` that is, `uid=0` in the non-global zone can not perform those operations (they are considered to be too unsafe.)

str_stack_t is defined in:

<file:///net/nptbld-x.sfbay/disk1/nordmark/si-only/webrev/usr/src/uts/common/sys/strsubr.h.html>

and contains:

```
783 /*
784  * Each stack (zone with the EXCL flag) has one of these
785  */
786 #ifndef NSTRPHASH
787 #define NSTRPHASH      128
788 #endif
789 struct str_stack {
790     netstack_t          *ss_netstack;    /* Common netstack */
791
792     struct autopush *ss_autopush;    /* autopush data array */
793     int                ss_nautopush;  /* number of elements in autopush */
794     struct autopush **ss_strpcache;   /* autopush hash list */
795     int                ss_strpmask;   /* used in hash function */
796     struct autopush *ss_strpfreep;   /* autopush freelist */
797     kmutex_t          ss_sad_lock;
798
799     int                ss_devcnt;     /* number of mux_nodes */
800     struct mux_node *ss_mux_nodes;   /* mux info for cycle checking */
801 };
802 typedef struct str_stack str_stack_t;
```

6 netstack framework design

The netstack framework code is initialized early in the boot and immediately does a zone_key_create() to be notified of zones that come and go.

When exclusive-IP zones come and go the netstack framework calls the corresponding create, shutdown and destroy callbacks that the TCP/IP modules have registered with the netstack framework.

When shared-IP zones come and go the netstack framework updates the mapping from kstat_create_netstack() to kstat_create_zone() by issuing kstat_zone_add() and kstat_zone_remove() as appropriate. (The kstat_create_netstack() is used by the TCP/IP modules to export kstats to the set of zones that use a particular IP instances while isolating the TCP/IP modules from the relationship between IP instances and zones.)

7 zone admin design changes

7.1 zone_create needing a flags argument for “exclusive”

The zone's infrastructure needs to know whether a zone has an exclusive IP or a shared IP. The zones code doesn't use this itself, but it must pass it to the zsd create() callback that the netstack framework code has registered. Since the zsd callbacks occur directly when zone_create() is called, it is not possible to have this flag be set later (e.g., using some new zone property).

7.2 zone_t structure

We are adding two fields to the kernel zone_t:

```
/*
```

```
* zone_iflist is protected by zone_lock
*/
```

```
struct ifnamelist *zone_iflist;
netstack_t      *zone_netstack;
```

The latter is used to efficiently be able to find a netstack_t given a cred_t; something which every tcp_open() needs to do.

7.3 zone_*_ifname system calls

As part of readying an exclusive-IP zone the kernel is told which network interface names (e.g., “bge1”) that the zone is allowed to use. This is done using the new zone_ifname_add()/zone_ifname_delete() system calls.

The list of ifnames hangs off the zone_t structure in the kernel.

It is used by user-level in a few ways:

- zoneadm verify checks that the network interfaces are not configured in the global zone, and not used by some other zone that is already in a state greater than “ready”.
- ifconfig -a plumb in an exclusive-IP zone retrieves the list of network interface names from zone_ifname.
- zoneadm list -l; it prints the list of network interface names that is given to a zone. TBD: This is in the process of being replaced by dladm.
- zoneadm verify and ifconfig plumb in the global zone can check whether some other zone has been assigned a network interface.

The list of ifnames could also be used by the kernel if that need arises:

- Be able to provide access control for DLPI style 2 devices if this need arises as we complete the project. (Currently we only support DLPI style 1 devices that have a separate /dev/ entry per instance. Project Clearview will provide /dev/ entries for all networking devices.)

The complete set of ifname related new system calls are:

```
extern int zone_add_ifname(zoneid_t, char *);
extern int zone_remove_ifname(zoneid_t, char *);
extern int zone_check_ifname(zoneid_t *, char *);
extern int zone_get_ifnum(zoneid_t, int *);
extern int zone_get_iflist(zoneid_t, char *);
```

7.4 zonecfg, zoneadm changes

In addition to the changes driven by the above descriptions there are no additional changes. Thus in summary

- zonecfg handling the new ip-type property. If the property is not specified in the xml file, it defaults to “shared” for backwards compatibility
- zonecfg verify checking that for an exclusive-IP zone does not have an address property for the net resource. Thus just as before, the verification happens when the “verify” subcommand is run either explicitly or implicitly as part of exiting zonecfg.
- zoneadm verify checking that for an exclusive-IP zone the network interface name exists and is not

assigned to any running zone

- zoneadm/zoneadmd giving PRIV_SYS_IP_CONFIG and PRIV_SYS_NET_RAW to an exclusive-IP zone
- zoneadm/zoneadmd giving a larger set of devices to an exclusive-IP zone. This consists of the various IP devices (/dev/ip etc), as well as the /dev/ entries for the network interface names that are given to the zone (e.g., /dev/bge3).

8 SMF methods

The net-physical and similar SMF method scripts depend on functions in smf_include.sh called smf_is_globalzone() and smf_is_nonglobalzone() to determine what subset of network configuration they should perform.

We introduce two new functions in smf_include.sh:

```
smf_configure_ip()
```

```
smf_dontconfigure_ip()
```

And make use of those instead in net-physical etc.

9 Threat Analysis

Zone as design in Solaris 10 prevents a non-global zone from doing some things towards the network, which adds some security benefits to zone beyond that of a separate server connected to the network. IP Instances is much like a separate sever in this respect.

The intent is to document these differences so that the users can understand the differences. If we look at different threats at different layers in the stack we approximately end up with a list like:

1. A host can potentially send bad Ethernet frames (bad length, CRC, etc) to make the Ethernet bridges have problems.
2. A host can send Ethernet frames with somebody else's MAC addresses as the source address. This makes learning bridges think that host has moved to a different bridge port, thus will disrupt routing for that host. Note that the learning tables are separate per VLAN in the bridges, thus such an attack is limited to a particular VLAN.
3. A host can send spoofed bridge PDUs (BPDU), which can make the spanning tree protocol between the bridges all confused. This can disrupt the whole Ethernet. It is also limited based on VLAN.
4. A host can spoof the ARP protocol for IPv4 (or Neighbor Discovery in the case of IPv6) claiming to have somebody else's IP address. That will disrupt service for that IP address.
5. A host can spoof ICMP redirects causing other hosts on the datalink to send packets to the wrong place.
6. A host can spoof RIP packets causing other hosts that listen to RIP to send packets to the wrong place. Routers might be configured to not listen to RIP. If they are not, then this attack could have even broader effect.

Note that #5 and #6 are possible even from a shared-IP zone, since it is ok to send any ICMP packets from such a zone, and RIP uses UDP which is also allowed.

But #2 through #4 are possible from an exclusive-IP zone (as well as from a separate host connected to the

same LAN or VLAN), hence are different. #1 is enforced by the Ethernet driver and hardware, and short of replacing the kernel device driver this isn't likely to be possible.

10 Design Alternatives

During the life of this project various alternative designs and approaches have been considered. This section documents some key ones.

10.1 *Only separate IP routing?*

For the customers that want to keep different (V)LANs isolated as they connect them to different zones in the same OS instance, couldn't we just implement separate routing tables in IP?

This was considered, but the design would be quite tricky given

- The complex interrelationships between the IP routing table and other data structures in and around IP such as
 - The ARP table
 - The interfaces (ill/ipif)
 - Multicast membership
 - IPsec being related to network interfaces and routes
- The large amount of code that would need *subtle* changes; there are on the order of 100 callers of the IRE lookup routines in the IP code base. One would have to inspect all of them.
- Once done we wouldn't be able to prove that there is no leakage; the only possible proof would be to do a complete code inspection.
- Maintenance would be difficult. In particular, how could we prevent that a subtle change somewhere in IP would accidentally create “leakage” between the different zones/VLAN?
- Based on experience this seems hard. We have a tiny step in this direction in Solaris 10 (verifying whether a default router is reachable “from a zone”), and even that simple code doesn't actually work as intended.

The IP instances approach avoids those complications by being brute force. The changes do not introduce semantic changes in the IP modules; the changes are driven by having removed the global symbols which means that the functions that need to access that data need to have a `foo_stack_t` pointer (either implicitly as part of an `ill_t`, `conn_t`, `icmp_t` etc, or explicitly as an argument.)

10.2 *Minimize diffs using #defines?*

Early in the project there was a concern about the number of lines of code that would need to be changed in order to do:

```
FROM    ip_g_forward      TO    ipst->ips_ip_g_forward
FROM    ip_mib             TO    ipst->ips_ip_mib
```

for all global state.

A possible alternative would have been to add defines such as

```
#define    ip_mib          ipst->ips_ip_mib
```

But this was rejected because of the obfuscation that would result; having a macro that silently uses a

variable (ipst in this case) would be very odd.

And in fact using tools like sed(1) it was not hard to apply this type of changes to the source base.

10.3 Make the netstack framework available for 3rd parties?

There is 3rd party software for Solaris that sit between IP and the network device drivers. There might be a desire to have such software be able to work with multiple IP instances, as opposed to being limited to working in the global zone.

Wouldn't making the netstack framework available help the 3rd parties do this?

Well, if I wanted to make e.g., 3rd party firewall software multi-instance with delegated management, I'd probably structure it to have separate instances per DLPI interface. That way the solution might apply to something other than Solaris.

If there is a need to make the delegated management fit in with zones, for instance to fit in with IP instances, then it would be easier for to just make it zone aware (record the zoneid in their open routine etc) than tying it directly with the netstack framework.

10.4 Avoid #defines for NETSTACK_IP?

The current netstack framework assumes that any additional kernel module that needs to use the framework can add a #define to netstack.h and recompile the Solaris kernel. Wouldn't it be better to have a mechanism like zone_key_create() where this is dynamic?

With such a mechanism there would need to be some dynamic data structure (e.g., linked list, hash table) of all the foo_stack_t that exist for a given netstackid. That would make it expensive to cross from e.g. a tcp_stack_t to a ip_stack_t. And there are plenty of places in the code where we need to cross from one module's stack structure to another e.g. when tcp does an ire_route_lookup, or IP passes a packet to IPsec. The current arrangement makes this quite efficient - just two pointer references

```
tcps->tcps_netstack->netstack_ip
```

Furthermore, kernel software that needs to be zone aware can be zone aware without being aware whether a particular zone has an exclusive or a shared IP. Only the kernel modules that need to do direct function calls into IP need to be aware of the netstacks.

11 Appendix A: example of TCP/IP module changes

The icmp.c (which implements RAW sockets) has the following modifications following the pattern outlined earlier in the document.

11.1 Defining a icmp_stack_t

See above.

11.2 Having the init/fini routines use netstack_register/unregister

The icmp_ddi_init() has been changed to only do:

```
5211 void
5212 icmp_ddi_init(void)
5213 {
5214     ICMP6_MAJ = ddi_name_to_major(ICMP6);
5215     icmp_max_optsize =
```

```

5216         optcom_max_optsize(icmp_opt_obj.odb_opt_des_arr,
5217         icmp_opt_obj.odb_opt_arr_cnt);
5218
5219     /*
5220     * We want to be informed each time a stack is created or
5221     * destroyed in the kernel, so we can maintain the
5222     * set of icmp_stack_t's.
5223     */
5224     netstack_register(NS_ICMP, rawip_stack_init, NULL, rawip_stack_fini);
5225 }

```

And `icmp_ddi_destroy()`

```

5227 void
5228 icmp_ddi_destroy(void)
5229 {
5230     netstack_unregister(NS_ICMP);
5231 }

```

11.3 Introduce `rawip_stack_init()` and `rawip_stack_fini()`

The `stack_init` allocates an `icmp_stack_t`, and then initializes it. Much of the initialization code came from the old `icmp_ddi_init()`.

```

5233 /*
5234 * Initialize the ICMP stack instance.
5235 */
5236 static void *
5237 rawip_stack_init(netstackid_t stackid, netstack_t *ns)
5238 {
5239     icmp_stack_t    *is;
5240     icmpparam_t     *pa;
5241
5242     is = (icmp_stack_t *)kmem_zalloc(sizeof (*is), KM_SLEEP);
5243     is->is_netstack = ns;
5244
5245     pa = (icmpparam_t *)kmem_alloc(sizeof (icmp_param_arr), KM_SLEEP);
5246     is->is_param_arr = pa;
5247     bcopy(icmp_param_arr, is->is_param_arr, sizeof (icmp_param_arr));
5248
5249     (void) icmp_param_register(&is->is_nd,
5250     is->is_param_arr, A_CNT(icmp_param_arr));
5251     is->is_ksp = rawip_kstat_init(stackid);
5252     return (is);
5253 }

```

`rawip_stack_fini` undoes what `rawip_stack_init` did:

```

5255 /*
5256 * Free the ICMP stack instance.
5257 */
5258 static void
5259 rawip_stack_fini(netstackid_t stackid, void *arg)
5260 {
5261     icmp_stack_t *is = (icmp_stack_t *)arg;
5262
5263     nd_free(&is->is_nd);
5264     kmem_free(is->is_param_arr, sizeof (icmp_param_arr));

```

```

5265         is->is_param_arr = NULL;
5266
5267         rawip_kstat_fini(stackid, is->is_ksp);
5268         is->is_ksp = NULL;
5269         kmem_free(is, sizeof (*is));
5270     }

```

11.4 Adding icmp_stack_t pointer

The icmp_t has an added pointer at the end:

```

69  /* Internal icmp control structure, one per open stream */
70  typedef struct icmp_s {
71      uint_t          icmp_state;      /* TPI state */
72      ...
141      icmp_stack_t   *icmp_is;        /* Stack instance */
142 } icmp_t;

```

11.5 icmp_open using netstack_find_by_cred()

The open routine has this added code:

```

1339         is = netstack_find_by_cred(credp)->netstack_icmp;
1340         ASSERT(is != NULL);

```

and later

```

1377         icmp->icmp_is = is;

```

icmp_close releases its reference on the netstack_t with:

```

644         netstack_rele(icmp->icmp_is->is_netstack);

```

11.6 Using `kstat_create_netstack` and `kstat_destroy_netstack`

```
5272 static void *
5273 rawip_kstat_init(netstackid_t stackid) {
5274     kstat_t *ksp;
5275
5276     rawip_named_kstat_t template = {
5277         { "inDatagrams",      KSTAT_DATA_UINT32, 0 },
5278         { "inCksumErrs",     KSTAT_DATA_UINT32, 0 },
5279         { "inErrors",        KSTAT_DATA_UINT32, 0 },
5280         { "outDatagrams",    KSTAT_DATA_UINT32, 0 },
5281         { "outErrors",       KSTAT_DATA_UINT32, 0 },
5282     };
5283
5284     ksp = kstat_create_netstack("icmp", 0, "rawip", "mib2",
5285                               KSTAT_TYPE_NAMED,
5286                               NUM_OF_FIELDS(rawip_named_kstat_t),
5287                               0, stackid);
5288     if (ksp == NULL || ksp->ks_data == NULL)
5289         return (NULL);
5290
5291     bcopy(&template, ksp->ks_data, sizeof (template));
5292     ksp->ks_update = rawip_kstat_update;
5293     ksp->ks_private = (void *) (uintptr_t) stackid;
5294
5295     kstat_install(ksp);
5296     return (ksp);
5297 }
5298
5299 static void
5300 rawip_kstat_fini(netstackid_t stackid, kstat_t *ksp)
5301 {
5302     if (ksp != NULL) {
5303         ASSERT(stackid == (netstackid_t) (uintptr_t) ksp->ks_private);
5304         kstat_delete_netstack(ksp, stackid);
5305     }
5306 }
```

12 Appendix B: large `foo_stack_t` structures

These are too large to include them in the body of the document.

12.1 ip_stack_t

Which is why it starts here:

```
157  /*
158   * IP stack instances
159   */
160  struct ip_stack {
161      netstack_t      *ips_netstack; /* Common netstack */
162
163      kmutex_t        ips_lock;      /* For ips_head, etc */
164      void            *ips_head;     /* Head for list of open conns */
165
166      struct ipparam_s *ips_param_arr; /* ndd variable table
*/
167      struct ipndp_s   *ips_ndp_arr;
168
169      mib2_ip_t        ips_ip_mib;    /* SNMP fixed size info */
170      mib2_icmp_t      ips_icmp_mib;
171      /*
172       * IPv6 mibs when the interface (ill) is not known.
173       * When the ill is known the per-interface mib in the ill is used.
174       */
175      mib2_ipv6IfStatsEntry_t ips_ip6_mib;
176      mib2_ipv6IfIcmpEntry_t  ips_icmp6_mib;
177
178      struct igmpstat      ips_igmpstat;
179
180      kstat_t             *ips_ip_mibkp; /* kstat exporting ip_mib data */
181      kstat_t             *ips_icmp_mibkp; /* kstat exporting icmp_mib data */
182      kstat_t             *ips_ip_kstat;
183      ip_stat_t           ips_ip_statistics;
184      kstat_t             *ips_ip6_kstat;
185      ip6_stat_t          ips_ip6_statistics;
186
187  /* ip.c */
188      krwlock_t           ips_ip_g_nd_lock;
189      kmutex_t            ips_igmp_timer_lock;
190      kmutex_t            ips_mld_timer_lock;
191      kmutex_t            ips_ip_mi_lock;
192      kmutex_t            ips_ip_addr_avail_lock;
193      krwlock_t           ips_ill_g_lock;
194      krwlock_t           ips_ipsec_capab_ills_lock;
195                          /* protects the list of IPsec capable ills */
196      struct ipsec_capab_ill_s *ips_ipsec_capab_ills_ah;
197      struct ipsec_capab_ill_s *ips_ipsec_capab_ills_esp;
198
199      krwlock_t           ips_ill_g_usesrc_lock;
200
201      struct ill_group *ips_illgrp_head_v4; /* Head of IPv4 ill groups */
202      struct ill_group *ips_illgrp_head_v6; /* Head of IPv6 ill groups */
203
204  /* ipclassifier.c - keep in ip_stack_t */
205      /* ipclassifier hash tables */
206      struct connf_s *ips_rts_clients;
207      struct connf_s *ips_ipcl_conn_fanout;
208      struct connf_s *ips_ipcl_bind_fanout;
209      struct connf_s *ips_ipcl_proto_fanout;
210      struct connf_s *ips_ipcl_proto_fanout_v6;
```

```

211     struct connf_s   *ips_ipcl_udp_fanout;
212     struct connf_s   *ips_ipcl_raw_fanout;
213     uint_t           ips_ipcl_conn_fanout_size;
214     uint_t           ips_ipcl_bind_fanout_size;
215     uint_t           ips_ipcl_udp_fanout_size;
216     uint_t           ips_ipcl_raw_fanout_size;
217     struct connf_s   *ips_ipcl_globalhash_fanout;
218     int              ips_conn_g_index;
219
220 /* ip.c */
221 /* Following protected by ips_igmp_timer_lock */
222 int             ips_igmp_time_to_next; /* Time since last timeout */
223 int             ips_igmp_timer_fired_last;
224 int             ips_igmp_deferred_next;
225 timeout_id_t    ips_igmp_timeout_id;
226 /* Protected by igmp_timer_lock */
227 boolean_t      ips_igmp_timer_setter_active;
228
229 /* Following protected by mld_timer_lock */
230 int             ips_mld_time_to_next; /* Time since last timeout */
231 int             ips_mld_timer_fired_last;
232 int             ips_mld_deferred_next;
233 timeout_id_t    ips_mld_timeout_id;
234 /* Protected by mld_timer_lock */
235 boolean_t      ips_mld_timer_setter_active;
236
237 /* Protected by igmp_slowtimeout_lock */
238 timeout_id_t    ips_igmp_slowtimeout_id;
239 kmutex_t        ips_igmp_slowtimeout_lock;
240
241 /* Protected by mld_slowtimeout_lock */
242 timeout_id_t    ips_mld_slowtimeout_id;
243 kmutex_t        ips_mld_slowtimeout_lock;
244
245 /* IPv4 forwarding table */
246 struct radix_node_head *ips_ip_ftable;
247
248 /* This is dynamically allocated in ip_ire_init */
249 struct irb      *ips_ip_cache_table;
250 /* This is dynamically allocated in ire_add_mrtun */
251 struct irb      *ips_ip_mrtun_table;
252
253 #define IPV6_ABITS           128
254 #define IP6_MASK_TABLE_SIZE (IPV6_ABITS + 1) /* 129 ptrs */
255
256 struct irb      *ips_ip_forwarding_table_v6[IP6_MASK_TABLE_SIZE];
257 /* This is dynamically allocated in ip_ire_init */
258 struct irb      *ips_ip_cache_table_v6;
259
260 uint32_t        ips_ire_handle;
261 /*
262  * ire_ft_init_lock is used while initializing ip_forwarding_table
263  * dynamically in ire_add.
264  */
265 kmutex_t        ips_ire_ft_init_lock;
266 kmutex_t        ips_ire_mrtun_lock; /* Protects mrtun table and count
*/
267 kmutex_t        ips_ire_srcif_table_lock; /* Same as above */
268 /*

```

```

269     * The following counts are used to determine whether a walk is
270     * needed through the reverse tunnel table or through ill
271     */
272     kmutex_t      ips_ire_handle_lock;      /* Protects ire_handle */
273
274     /* # of ires in reverse tun table */
275     uint_t        ips_ire_mrtun_count;
276
277     /* # of ires in all srcif tables */
278     uint_t        ips_ire_srcif_table_count;
279
280     uint32_t      ips_ip_cache_table_size;
281     uint32_t      ips_ip6_cache_table_size;
282     uint32_t      ips_ip6_fhtable_hash_size;
283
284     ire_stats_t   ips_ire_stats_v4;        /* IPv4 ire statistics */
285     ire_stats_t   ips_ire_stats_v6;        /* IPv6 ire statistics */
286
287     /* pending binds */
288     mblk_t        *ips_ip6_asp_pending_ops;
289     mblk_t        *ips_ip6_asp_pending_ops_tail;
290
291     /* Synchronize updates with table usage */
292     mblk_t        *ips_ip6_asp_pending_update; /* pending table updates */
293
294     boolean_t     ips_ip6_asp_uip;         /* table update in progress */
295     kmutex_t      ips_ip6_asp_lock;        /* protect all the above */
296     uint32_t      ips_ip6_asp_refcnt;      /* outstanding references */
297
298     struct ip6_asp *ips_ip6_asp_table;
299     /* The number of policy entries in the table */
300     uint_t        ips_ip6_asp_table_count;
301
302     int           ips_ip_g_forward;
303     int           ips_ipv6_forward;
304
305     time_t        ips_ip_g_frag_timeout;
306     clock_t       ips_ip_g_frag_timo_ms;
307
308     queue_t       *ips_ip_g_mrouter;
309
310     /* Time since last icmp_pkt_err */
311     clock_t       ips_icmp_pkt_err_last;
312     /* Number of packets sent in burst */
313     uint_t        ips_icmp_pkt_err_sent;
314     /* Used by icmp_send_redirect_v6 for picking random src. */
315     uint_t        ips_icmp_redirect_v6_src_index;
316
317     /* Protected by ip_mi_lock */
318     void          *ips_ip_g_head;         /* Instance Data List Head */
319
320     caddr_t       ips_ip_g_nd;           /* Named Dispatch List Head */
321
322     /* Multirouting/CGTP stuff */
323     struct cgtp_filter_ops *ips_ip_cgtp_filter_ops; /* CGTP hooks
*/
324     int           ips_ip_cgtp_filter_rev; /* CGTP hooks version */
325     boolean_t     ips_ip_cgtp_filter;    /* Enable/disable CGTP hooks */
326     /* Interval (in ms) between consecutive 'bad MTU' warnings */

```

```

327     hrtime_t         ips_ip_multirt_log_interval;
328     /* Time since last warning issued. */
329     hrtime_t         ips_multirt_bad_mtu_last_time;
330
331     kmutex_t         ips_ip_trash_timer_lock;
332     timeout_id_t     ips_ip_ire_expire_id; /* IRE expiration timer. */
333     struct ipsq_s     *ips_ipsq_g_head;
334     uint_t           ips_ill_index; /* Used to assign interface indices */
335     /* When set search for unused index */
336     boolean_t        ips_ill_index_wrap;
337
338     clock_t          ips_ip_ire_arp_time_elapsed;
339     /* Time since IRE cache last flushed */
340     clock_t          ips_ip_ire_rd_time_elapsed;
341     /* ... redirect IREs last flushed */
342     clock_t          ips_ip_ire_pmtu_time_elapsed;
343     /* Time since path mtu increase */
344
345     uint_t           ips_ip_redirect_cnt;
346     /* Num of redirect routes in ftable */
347     uint_t           ips_ipv6_ire_default_count;
348     /* Number of IPv6 IRE_DEFAULT entries */
349     uint_t           ips_ipv6_ire_default_index;
350     /* Walking IPv6 index used to mod in */
351
352     uint_t           ips_loopback_packets;
353
354     /* NDP/NCE structures for IPv4 and IPv6 */
355     struct ndp_g_s   *ips_ndp4;
356     struct ndp_g_s   *ips_ndp6;
357
358     /* ip_mroute stuff */
359     kmutex_t         ips_ip_g_mrouter_mutex;
360
361     struct mrtstat   *ips_mrtstat; /* Stats for netstat */
362     int              ips_saved_ip_g_forward;
363
364     /* numvifs is only a hint about the max interface being used. */
365     ushort_t         ips_numvifs;
366     kmutex_t         ips_numvifs_mutex;
367
368     struct vif        *ips_vifs;
369     struct mfcb       *ips_mfcs; /* kernel routing table */
370     struct tbf        *ips_tbf;
371     /*
372      * One-back cache used to locate a tunnel's vif,
373      * given a datagram's src ip address.
374      */
375     ipaddr_t         ips_last_encap_src;
376     struct vif        *ips_last_encap_vif;
377     kmutex_t         ips_last_encap_lock; /* Protects the above */
378
379     /*
380      * reg_vif_num is protected by numvifs_mutex
381      */
382     /* Whether or not special PIM assert processing is enabled. */
383     ushort_t         ips_reg_vif_num; /* Index to Register vif */
384     int              ips_pim_assert;
385

```

```

386 union ill_g_head_u *ips_ill_g_heads; /* ILL List Head */
387
388 kstat_t          *ips_loopback_ksp;
389
390 uint_t           ips_ipif_src_random;
391
392 struct idl_s      *ips_conn_drain_list; /* Array of conn drain lists */
393 uint_t           ips_conn_drain_list_cnt; /* Count of conn_drain_list */
394 int              ips_conn_drain_list_index; /* Next drain_list */
395
396 /*
397  * ID used to assign next free one.
398  * Increases by one. Once it wraps we search for an unused ID.
399  */
400 uint_t           ips_ip_src_id;
401 boolean_t        ips_srcid_wrapped;
402
403 struct srcid_map *ips_srcid_head;
404 krwlock_t        ips_srcid_lock;
405
406 uint64_t         ips_ipif_g_seqid;
407 union phyint_list_u *ips_phyint_g_list; /* start of phyint list */
408
409 /*
410  * Reflects value of FAILBACK variable in IPMP config file
411  * /etc/default/mpathd. Default value is B_TRUE.
412  * Set to B_FALSE if user disabled failback by configuring
413  * "FAILBACK=no" in.mpathd uses SIOCSIPMPFAILBACK ioctl to pass this
414  * information to kernel.
415  */
416 boolean_t ips_ipmp_enable_failback;
417
418 /* ip_net.c */
419 /*
420  * The taskq eventq_queue_in is for the upside inject messages.
421  * The taskq eventq_queue_out is for the downside inject messages.
422  * The taskq eventq_queue_nic is for the nic event messages.
423  */
424 struct ddi_taskq *ips_eventq_queue_in;
425 struct ddi_taskq *ips_eventq_queue_out;
426 struct ddi_taskq *ips_eventq_queue_nic;
427
428 hook_family_t ips_ipv4root;
429 hook_family_t ips_ipv6root;
430
431 /*
432  * Hooks for firewalling
433  */
434 hook_event_t ips_ip4_physical_in_event;
435 hook_event_t ips_ip4_physical_out_event;
436 hook_event_t ips_ip4_forwarding_event;
437 hook_event_t ips_ip4_loopback_in_event;
438 hook_event_t ips_ip4_loopback_out_event;
439 hook_event_t ips_ip4_nic_events;
440 hook_event_t ips_ip6_physical_in_event;
441 hook_event_t ips_ip6_physical_out_event;
442 hook_event_t ips_ip6_forwarding_event;
443 hook_event_t ips_ip6_loopback_in_event;
444 hook_event_t ips_ip6_loopback_out_event;

```

```

445     hook_event_t           ips_ip6_nic_events;
446
447     hook_event_token_t     ips_ipv4firewall_physical_in;
448     hook_event_token_t     ips_ipv4firewall_physical_out;
449     hook_event_token_t     ips_ipv4firewall_forwarding;
450     hook_event_token_t     ips_ipv4firewall_loopback_in;
451     hook_event_token_t     ips_ipv4firewall_loopback_out;
452     hook_event_token_t     ips_ipv4nicevents;
453     hook_event_token_t     ips_ipv6firewall_physical_in;
454     hook_event_token_t     ips_ipv6firewall_physical_out;
455     hook_event_token_t     ips_ipv6firewall_forwarding;
456     hook_event_token_t     ips_ipv6firewall_loopback_in;
457     hook_event_token_t     ips_ipv6firewall_loopback_out;
458     hook_event_token_t     ips_ipv6nicevents;
459
460     net_data_t             ips_ipv4;
461     net_data_t             ips_ipv6;
462 }
463 typedef struct ip_stack ip_stack_t;

```

```

tcp_stack_t
137 /*
138  * TCP stack instances
139  */
140 struct tcp_stack {
141     netstack_t             *tcps_netstack; /* Common netstack */
142
143     mib2_tcp_t             tcps_mib;
144
145     /* Protected by tcp_g_q_lock */
146     queue_t                *tcps_g_q;      /* Default queue */
147     uint_t                 tcps_g_q_ref;   /* Number of tcp_t's that use it */
148     kmutex_t               tcps_g_q_lock;
149     boolean_t              tcps_g_q_create_inprogress;
150     struct __ldi_handle *tcps_g_q_lh;
151     cred_t                 *tcps_g_q_cr;   /* For _inactive close call */
152
153     /* Protected by tcp_hsp_lock */
154     struct tcp_hsp         **tcps_hsp_hash; /* Hash table for HSPs */
155     krwlock_t              tcps_hsp_lock;
156
157     /*
158      * Extra privileged ports. In host byte order.
159      * Protected by tcp_epriv_port_lock.
160      */
161 #define TCP_NUM_EPRIV_PORTS    64
162     int                    tcps_g_num_epriv_ports;
163     uint16_t               tcps_g_epriv_ports[TCP_NUM_EPRIV_PORTS];
164     kmutex_t               tcps_epriv_port_lock;
165
166     /*
167      * The smallest anonymous port in the privileged port range which TCP
168      * looks for free port. Use in the option TCP_ANONPRIVBIND.
169      */
170     in_port_t              tcps_min_anonpriv_port;
171
172     /* Only modified during _init and _fini thus no locking is needed. */

```

```

173     caddr_t         tcps_g_nd;
174     struct tcpparam_s *tcps_params; /* ndd parameters */
175     struct tcpparam_s *tcps_wroff_xtra_param;
176     struct tcpparam_s *tcps_mdt_head_param;
177     struct tcpparam_s *tcps_mdt_tail_param;
178     struct tcpparam_s *tcps_mdt_max_pbufs_param;
179
180     /* Hint not protected by any lock */
181     uint_t         tcps_next_port_to_try;
182
183     /* TCP bind hash list - all tcp_t with state >= BOUND. */
184     struct tf_s     *tcps_bind_fanout;
185
186     /* TCP queue hash list - all tcp_t in case they will be an acceptor. */
187     struct tf_s     *tcps_acceptor_fanout;
188
189     /* The reserved port array. */
190     struct tcp_rport_s *tcps_reserved_port;
191
192     /* Locks to protect the tcp_reserved_ports array. */
193     krwlock_t      tcps_reserved_port_lock;
194
195     /* The number of ranges in the array. */
196     uint32_t       tcps_reserved_port_array_size;
197
198     /*
199      * MIB-2 stuff for SNMP
200      * Note: tcpInErrs {tcp 15} is accumulated in ip.c
201      */
202     kstat_t        *tcps_mibkp; /* kstat exporting tcp_mib data */
203     kstat_t        *tcps_kstat;
204     tcp_stat_t     tcps_statistics;
205
206     uint32_t       tcps_iss_incr_extra;
207     /* Incremented for each connection */
208     kmutex_t       tcps_iss_key_lock;
209     MD5_CTX        tcps_iss_key;
210
211     /* Packet dropper for TCP IPsec policy drops. */
212     ipdropper_t    tcps_dropper;
213
214     /*
215      * This controls the rate some ndd info report functions can be used
216      * by non-privileged users. It stores the last time such info is
217      * requested. When those report functions are called again, this
218      * is checked with the current time and compare with the ndd param
219      * tcp_ndd_get_info_interval.
220      */
221     clock_t        tcps_last_ndd_get_info_time;
222
223     /*
224      * These two variables control the rate for TCP to generate RSTs in
225      * response to segments not belonging to any connections. We limit
226      * TCP to sent out tcp_rst_sent_rate (ndd param) number of RSTs in
227      * each 1 second interval. This is to protect TCP against DoS attack.
228      */
229     clock_t        tcps_last_rst_intrvl;
230     uint32_t       tcps_rst_cnt;
231     /* The number of RST not sent because of the rate limit. */

```

```
232         uint32_t         tcps_rst_unsent;
233     };
234     typedef struct tcp_stack tcp_stack_t;
```

12.2 sctp_stack_t

```
70 /*
71  * SCTP stack instances
72  */
73 struct sctp_stack {
74     netstack_t      *sctps_netstack;          /* Common netstack */
75
76     mib2_sctp_t      sctps_mib;
77     /*
78      * Default queue used for sending packets.  No need to have lock for it
79      * as it should never be changed.
80      */
81     queue_t         *sctps_g_q;
82     uint_t           sctps_g_q_ref; /* Number of sctp_t's that use it */
83     boolean_t       sctps_g_q_create_inprogress;
84     struct __ldi_handle *sctps_g_q_lh;
85     cred_t           *sctps_g_q_cr; /* For _inactive close call */
86     /* The default sctp_t for responding out of the blue packets. */
87     struct sctp_s    *sctps_gsctp;
88
89     /* Protected by sctp_g_lock */
90     struct list      sctps_g_list; /* SCTP instance data chain */
91     kmutex_t         sctps_g_lock;
92
93 #define SCTP_NUM_EPRIV_PORTS    64
94     int              sctps_g_num_epriv_ports;
95     uint16_t         sctps_g_epriv_ports[SCTP_NUM_EPRIV_PORTS];
96     kmutex_t         sctps_epriv_port_lock;
97     uint_t           sctps_next_port_to_try;
98
99     mblk_t           *sctps_pad_mp; /* pad unaligned data chunks */
100
101     /* SCTP bind hash list - all sctp_t with state >= BOUND. */
102     struct sctp_tf_s  *sctps_bind_fanout;
103     /* SCTP listen hash list - all sctp_t with state == LISTEN. */
104     struct sctp_tf_s  *sctps_listen_fanout;
105     struct sctp_tf_s  *sctps_conn_fanout;
106     uint_t            sctps_conn_hash_size;
107
108     /* Only modified during _init and _fini thus no locking is needed. */
109     caddr_t           sctps_g_nd;
110     struct sctpparam_s *sctps_params;
111     struct sctpparam_s *sctps_wroff_xtra_param;
112
113 /* This lock protects the SCTP recvq_tq_list array and recvq_tq_list_cur_sz. */
114     kmutex_t         sctps_rq_tq_lock;
115     int              sctps_recvq_tq_list_max_sz;
116     taskq_t          **sctps_recvq_tq_list;
117
118     /* Current number of recvq taskq.  At least 1 for the default taskq. */
119     uint32_t         sctps_recvq_tq_list_cur_sz;
120     uint32_t         sctps_recvq_tq_list_cur;
121
122     /* Global list of SCTP ILLs */
123     struct sctp_ill_hash_s *sctps_g_ills;
124     uint32_t         sctps_ills_count;
125     krwlock_t        sctps_g_ills_lock;
```

```
126
127     /* Global list of SCTP IPIFs */
128     struct sctp_ipif_hash_s *sctps_g_ipifs;
129     uint32_t sctps_g_ipifs_count;
130     krwlock_t sctps_g_ipifs_lock;
131
132     /* kstat exporting sctp_mib data */
133     kstat_t *sctps_mibkp;
134     kstat_t *sctps_kstat;
135     sctp_kstat_t sctps_statistics;
136 };
137 typedef struct sctp_stack sctp_stack_t;
```

12.3 ipf_stack_t

```
40  /*
41  * IPF stack instances
42  */
43  struct ipf_stack {
44      netstack_t          *ifs_netstack;
45
46      /* ipf module */
47      fr_info_t           ifs_frcache[2][8];
48
49      filterstats_t       ifs_frstats[2];
50      frentry_t           *ifs_ipfilter[2][2];
51      frentry_t           *ifs_ipfilter6[2][2];
52      frentry_t           *ifs_ipacct6[2][2];
53      frentry_t           *ifs_ipacct[2][2];
54  #if 0 /* not used */
55      frentry_t           *ifs_ipnatrules[2][2];
56  #endif
57      frgroup_t           *ifs_ipfgroups[IPL_LOGSIZE][2];
58      int                 ifs_fr_refcnt;
59      /*
60       * For fr_running:
61       * 0 == loading, 1 = running, -1 = disabled, -2 = unloading
62       */
63      int                 ifs_fr_running;
64      int                 ifs_fr_flags;
65      int                 ifs_fr_active;
66      int                 ifs_fr_control_forwarding;
67      int                 ifs_fr_update_ipid;
68  #if 0
69      ushort_t           ifs_fr_ip_id;
70  #endif
71      int                 ifs_fr_chksrc;
72      int                 ifs_fr_minttl;
73      int                 ifs_fr_icmpminfragmtu;
74      int                 ifs_fr_pass;
75      ulong_t            ifs_fr_frouteok[2];
76      ulong_t            ifs_fr_userifqs;
77      u_long              ifs_fr_badcoalesces[2];
78      int                 ifs_fr_unreach;
79      uchar_t            ifs_ipf_iss_secret[32];
80      timeout_id_t       ifs_fr_timer_id;
81  #if 0
82      timeout_id_t       ifs_synctimeoutid;
83  #endif
84      int                 ifs_ipf_locks_done;
85
86      ipftoken_t          *ifs_ipftokenhead;
87      ipftoken_t          **ifs_ipftokentail;
88
89      ipfmutex_t          ifs_ipl_mutex;
90      ipfmutex_t          ifs_ipf_authmx;
91      ipfmutex_t          ifs_ipf_rw;
92      ipfmutex_t          ifs_ipf_timeoutlock;
93      ipfrwlock_t        ifs_ipf_mutex;
94      ipfrwlock_t        ifs_ipf_global;
95      ipfrwlock_t        ifs_ipf_frcache;
```

```

96     ipfrwlock_t     ifs_ip_poolrw;
97     ipfrwlock_t     ifs_ipf_frag;
98     ipfrwlock_t     ifs_ipf_state;
99     ipfrwlock_t     ifs_ipf_nat;
100    ipfrwlock_t     ifs_ipf_natfrag;
101    ipfmutex_t      ifs_ipf_nat_new;
102    ipfmutex_t      ifs_ipf_natio;
103    ipfrwlock_t     ifs_ipf_auth;
104    ipfmutex_t      ifs_ipf_stinsert;
105    ipfrwlock_t     ifs_ipf_ipidfrag;
106    ipfrwlock_t     ifs_ipf_tokens;
107    kcondvar_t      ifs_iplwait;
108    kcondvar_t      ifs_ipfauthwait;
109
110    ipftuneable_t    *ifs_ipf_tuneables;
111    ipftuneable_t    *ifs_ipf_tunelist;
112
113    /* ip_fil_solaris.c */
114    hook_t           ifs_ipfhook_in;
115    hook_t           ifs_ipfhook_out;
116    hook_t           ifs_ipfhook_nicevents;
117
118    /* flags to indicate whether hooks are registered. */
119    boolean_t        ifs_hook4_physical_in;
120    boolean_t        ifs_hook4_physical_out;
121    boolean_t        ifs_hook4_nic_events;
122    boolean_t        ifs_hook4_loopback_in;
123    boolean_t        ifs_hook4_loopback_out;
124    boolean_t        ifs_hook6_physical_in;
125    boolean_t        ifs_hook6_physical_out;
126    boolean_t        ifs_hook6_nic_events;
127    boolean_t        ifs_hook6_loopback_in;
128    boolean_t        ifs_hook6_loopback_out;
129
130    int               ifs_ipf_loopback;
131    net_data_t        ifs_ipf_ipv4;
132    net_data_t        ifs_ipf_ipv6;
133
134    /* ip_auth.c */
135    int               ifs_fr_authsize;
136    int               ifs_fr_authused;
137    int               ifs_fr_defaultauthage;
138    int               ifs_fr_auth_lock;
139    int               ifs_fr_auth_init;
140    fr_authstat_t    ifs_fr_authstats;
141    frauth_t          *ifs_fr_auth;
142    mb_t              **ifs_fr_authpkts;
143    int               ifs_fr_authstart;
144    int               ifs_fr_authend;
145    int               ifs_fr_authnext;
146    frauthent_t      *ifs_fae_list;
147    frentry_t         *ifs_ipauth;
148    frentry_t         *ifs_fr_authlist;
149
150    /* ip_frag.c */
151    ipfr_t            *ifs_ipfr_list;
152    ipfr_t            **ifs_ipfr_tail;
153    ipfr_t            **ifs_ipfr_heads;
154

```

```

155     ipfr_t           *ifs_ipfr_natlist;
156     ipfr_t           **ifs_ipfr_nattail;
157     ipfr_t           **ifs_ipfr_nattab;
158
159     ipfr_t           *ifs_ipfr_ipidlist;
160     ipfr_t           **ifs_ipfr_ipidtail;
161     ipfr_t           **ifs_ipfr_ipidtab;
162
163     ipfrstat_t       ifs_ipfr_stats;
164     int               ifs_ipfr_inuse;
165     int               ifs_ipfr_size;
166
167     int               ifs_fr_ipfrttl;
168     int               ifs_fr_frag_lock;
169     int               ifs_fr_frag_init;
170     ulong_t          ifs_fr_ticks;
171
172     frentry_t         ifs_frblock;
173
174
175     /* ip_ftp_pxy.c */
176     frentry_t         ifs_ftppxyfr;
177     int               ifs_ftp_proxy_init;
178     int               ifs_ippr_ftp_pasvonly;
179     int               ifs_ippr_ftp_insecure;
180     /* Do not require logins before transfers */
181     int               ifs_ippr_ftp_pasvrdr;
182     int               ifs_ippr_ftp_forcepasv;
183     /* PASV must be last command prior to 227 */
184
185     /* ip_h323_pxy.c */
186     frentry_t         ifs_h323_fr;
187     int               ifs_h323_proxy_init;
188
189     /* ip_htable.c */
190     iphtable_t        *ifs_ipf_htables[IPL_LOGSIZE];
191     ulong_t           ifs_ipht_nomem[IPL_LOGSIZE];
192     ulong_t           ifs_ipf_nhtables[IPL_LOGSIZE];
193     ulong_t           ifs_ipf_nhtnodes[IPL_LOGSIZE];
194
195
196     /* ip_ipsec_pxy.c */
197     frentry_t         ifs_ipsecfr;
198     ipftq_t           *ifs_ipsecnattqe;
199     ipftq_t           *ifs_ipsecstatetqe;
200     char              ifs_ipsec_buffer[1500];
201     int               ifs_ipsec_proxy_init;
202     int               ifs_ipsec_proxy_ttl;
203
204     /* ip_irc_pxy.c */
205     frentry_t         ifs_ircnatfr;
206     int               ifs_irc_proxy_init;
207
208     /* ip_log.c */
209     iplog_t           **ifs_iplh[IPL_LOGSIZE];
210     iplog_t           *ifs_iplt[IPL_LOGSIZE];
211     iplog_t           *ifs_ipll[IPL_LOGSIZE];
212     int               ifs_iplused[IPL_LOGSIZE];
213     fr_info_t         ifs_iplcrc[IPL_LOGSIZE];

```

```

214         int                ifs_ipl_suppress;
215         int                ifs_ipl_buffer_sz;
216         int                ifs_ipl_logmax;
217         int                ifs_ipl_logall;
218         int                ifs_ipl_log_init;
219         int                ifs_ipl_logsize;
220
221         /* ip_lookup.c */
222         ip_pool_stat_t     ifs_ippoolstat;
223         int                ifs_ip_lookup_inited;
224
225         /* ip_nat.c */
226         /* nat_table[0] -> hashed list sorted by inside (ip, port) */
227         /* nat_table[1] -> hashed list sorted by outside (ip, port) */
228         nat_t              **ifs_nat_table[2];
229         nat_t              *ifs_nat_instances;
230         ipnat_t            *ifs_nat_list;
231         uint_t             ifs_ipf_nattable_sz;
232         uint_t             ifs_ipf_nattable_max;
233         uint_t             ifs_ipf_natrules_sz;
234         uint_t             ifs_ipf_rdrrules_sz;
235         uint_t             ifs_ipf_hostmap_sz;
236         uint_t             ifs_fr_nat_maxbucket;
237         uint_t             ifs_fr_nat_maxbucket_reset;
238         uint32_t           ifs_nat_masks;
239         uint32_t           ifs_rdr_masks;
240         ipnat_t            **ifs_nat_rules;
241         ipnat_t            **ifs_rdr_rules;
242         hostmap_t          **ifs_mactable;
243         hostmap_t          *ifs_ipf_hm_maplist;
244
245         ipftq_t            ifs_nat_tq[IPF_TCP_NSTATES];
246         ipftq_t            ifs_nat_udptq;
247         ipftq_t            ifs_nat_icmptq;
248         ipftq_t            ifs_nat iptq;
249         ipftq_t            *ifs_nat_utqe;
250         int                ifs_nat_logging;
251         ulong_t            ifs_fr_defnatage;
252         ulong_t            ifs_fr_defnaticmpage;
253         natstat_t          ifs_nat_stats;
254         int                ifs_fr_nat_lock;
255         int                ifs_fr_nat_init;
256
257         /* ip_netbios_pxy.c */
258         frentry_t          ifs_netbiosfr;
259         int                ifs_netbios_proxy_init;
260
261         /* ip_raudio_pxy.c */
262         frentry_t          ifs_raudiofr;
263         int                ifs_raudio_proxy_init;
264
265         /* ip_rcmd_pxy.c */
266         frentry_t          ifs_rcmdfr;
267         int                ifs_rcmd_proxy_init;
268
269         /* ip_rpcb_pxy.c */
270         frentry_t          ifs_rpcbfr;
271         int                ifs_rpcbcnt; /* required locking */
272         int                ifs_rpcb_proxy_init;

```

```

273
274     /* ip_pool.c */
275     ip_pool_stat_t         ifs_ipoolstat;
276     ip_pool_t             *ifs_ip_pool_list[IPL_LOGSIZE];
277
278     /* ip_proxy.c */
279     ap_session_t          *ifs_ap_sess_list;
280     aproxy_t              *ifs_ap_proxylist;
281     aproxy_t              *ifs_ap_proxies; /* copy of lcl_ap_proxies */
282
283     /* ip_state.c */
284     ipstate_t             **ifs_ips_table;
285     ulong_t               *ifs_ips_seed;
286     int                   ifs_ips_num;
287     ulong_t               ifs_ips_last_force_flush;
288     ips_stat_t            ifs_ips_stats;
289
290     ulong_t               ifs_fr_tcpidletimeout;
291     ulong_t               ifs_fr_tcpclosewait;
292     ulong_t               ifs_fr_tcplastack;
293     ulong_t               ifs_fr_tcptimeout;
294     ulong_t               ifs_fr_tcpclosed;
295     ulong_t               ifs_fr_tcphalfclosed;
296     ulong_t               ifs_fr_udptimeout;
297     ulong_t               ifs_fr_udpacktimeout;
298     ulong_t               ifs_fr_icmptimeout;
299     ulong_t               ifs_fr_icmpacktimeout;
300     int                   ifs_fr_statemax;
301     int                   ifs_fr_statesize;
302     int                   ifs_fr_state_doflush;
303     int                   ifs_fr_state_lock;
304     int                   ifs_fr_state_maxbucket;
305     int                   ifs_fr_state_maxbucket_reset;
306     int                   ifs_fr_state_init;
307     ipftq_t               ifs_ips_tqtqb[IPF_TCP_NSTATES];
308     ipftq_t               ifs_ips_udptq;
309     ipftq_t               ifs_ips_udpacktq;
310     ipftq_t               ifs_ips iptq;
311     ipftq_t               ifs_ips_icmptq;
312     ipftq_t               ifs_ips_icmpacktq;
313     ipftq_t               *ifs_ips_utqe;
314     int                   ifs_ipstate_logging;
315     ipstate_t             *ifs_ips_list;
316     u_long                ifs_fr ipttimeout;
317
318     /* radix.c */
319     int                   ifs_max_keylen;
320     struct radix_mask      *ifs_rn_mkfreelist;
321     struct radix_node_head *ifs_mask_rnhead;
322     char                   *ifs_addmask_key;
323     char                   *ifs_rn_zeros;
324     char                   *ifs_rn_ones;
325 #ifdef KERNEL
326     /* kstats for inbound and outbound */
327     kstat_t               *ifs_kstatp[2];
328 #endif
329 };

```