application layer api

ietf://mif/93

v2

lots of question marks, lots of ellipses
:-)

PvD

- RFC7556#section-2
  - “A consistent set of network configuration information.”
- includes:
  - participating interfaces
  - addresses
  - routes
    - default routes, of course
    - but also RIO -type information
  - DNS servers and search path
  - HTTP proxy
  - yet to be specified: metering, medium, captive portal URL, …
- not learned atomically, not static

what things do apps need?

- PvD configuration information
  - get config data, get notified of updates
- PvD selection
  - granularity: system default + per process / thread / file descriptor / ...
- socket-level control
  - control routing and address selection
  - automatic PvD “tagging” of incoming traffic not already classified
- DNS resolution:
  - which DNS servers and search path to use
  - correct routing to those DNS servers
    - don’t want to query the right DNS server via the wrong network
- useful errors (a la `ENONET`, …)

source address and routing selection

- if a PvD has been specified:
  - it is RECOMMENDED that source address selection be restricted to PvD addresses
    - update RFC6274#section-4
  - it is important to return errors
    - might have two PvDs active: one IPv4-only and one IPv6-only
    - `ENONET`, `EPROTONOSUPPORT`, `EADDRNOTAVAIL`, `EHOSTUNREACH`, ...
- destination reachability:
  - userspace libraries often use `connect()` tricks to obtain source addresses for sorting
  - this MUST use the routing configuration of the desired PvD
- `getaddrinfo()` and `AI_ADDRCONFIG`
  - RFC3493#section-6.1 “...shall be returned only if an IPv4/v6 address is configured on the local system...” → “... within the requested PvD”

new things to define

- get PvD configuration data
  - should be extensible
    - `struct` with `#ifdef` extra members?
    - separate query for each configuration element of interest (a la `getsockopt()`)?
  - notification of configuration changes
  - figure out how to express PvD ↔ interface/scope_id relationship
- set/get process default PvD, thread default PvD
- a simple programmatic way to reference a specific PvD in these calls
  - e.g.: `typedef uint64_t pvd_reference_t`
  - separate attaches to the same PvD may be assigned different pvd_ref_t values
  - may help for distinguishing implicit PvDs
  - `PVD_UNSPECIFIED`

some sockets API considerations

- basically the strong host model
  - except that PvD ↔ interfaces is m:n
  - PvD IDs could be thought of as site-local scopes
- requests for a PvD not currently configured should return `ENONET`
  - other useful errors need to be returned throughout
  - some of this may want to be relaxed for privileged users
- per packet PvD selection? … maybe, maybe not
  - once a source address has been selected, using it to send traffic via a different PvD is essentially best effort / subject to system-specific policy

sockets API

- `setsockopt()` / `getsockopt()`
  - `IP_RECVPVD` / `IPV6_RECVPVD`
    - `recvmsg()` should include indication of PVD to which the packet arrived
  - `IP_PVD` / `IPV6_PVD`
    - source address selection and applicable routing table is restricted to the specified PvD
    - if a source address has already been selected from one PvD, transmitting via another PvD is NOT RECOMMENDED (but of course possible)
  - `PVD_UNSPECIFIED`
    - no PvD explicitly requested
    - also used to clear a process or thread default and revert to system default
  - for `PF_INET` / `PF_INET6` sockets

sockets API

- `socket()`
  - if a process-default or thread-default PvD has been set, the returned file descriptor must be “bound” to the PvD
    - i.e. as if `setsockopt(SOL_IPV6, IPV6_PVD, ...)` had been called
  - otherwise, the file descriptor defaults to `PVD_UNSPECIFIED`
- `bind()`
  - if a PvD is specified && address is unspecified, it is RECOMMENDED source address selection be restricted to this PVD
  - else if a PvD is specified && address is not unspecified, `EADDRNOTAVAIL` might be returned
  - else best effort / system-specific policies apply

sockets API

- `listen()`
  - if the file descriptor is already bound to a PvD, only traffic to one of the PvD's addresses should cause the file descriptor to become readable
  - other traffic should receive an ICMP error
- `accept()`
  - returned file descriptors should be bound to the PvD of:
    - the listening socket, if it was not bound to `PVD_UNSPECIFIED`
    - the PvD of the destination address on the system
- `connect()`
  - `bind()` discussion applies for source address selection
  - `EHOSTUNREACH` / `ENETUNREACH` might be returned

sockets API

- `sendmsg()` / `recvmsg()` `cmsg` semantics
  - `setsockopt()` / `getsockopt()` options apply
    - `IP_RECVPVD` / `IPV6_RECVPV6`
    - `IP_PVD` / `IPV6_PVD`
  - source address selection discussion applies for `IP_PVD` / `IPV6_PVD`
    - some combinations just may not work
    - some may require privileges to even attempt

DNS resolution

- can be implemented with per-PvD DNS server and search path state
- use sockets API changes for reaching specified nameservers
- maybe extend `getaddrinfo(..., hints, ...)`
  - ```
    struct addrinfo {
        int ai_flags;
        …
    #ifdef HAVE_PVD_API
        pvd_reference_t ai_pvd;
    #endif
    };
    ```
- what about `getnameinfo()`, `res_query()`, and getdnsapi ?
    ```

next steps?

- discussion
  - new functions and data types
  - how to deal with VPNs
    - should apps be required to know about them?
    - or does a VPN just affect `PVD_UNSPECIFIED` traffic?
  - accumulation of PvD configuration data is not atomic
    - signaling of PvD config data changes
  - where should policy / privileges influence behaviour?
  - many system-side things deliberately not discussed here
- fold feedback into draft-liu-mif-socket-api
- continue discussion on the list