                 Multiple Provisioning Domains API Requirements
                       draft-kline-mif-mpvd-api-reqs-00

Abstract

   RFC 7556 [RFC7556] provides the essential conceptual guidance an API
   designer would need to support use of PvDs. This document aims to
   capture the requirements for an API that can be used by applications
   that would be considered "advanced", according to section 6.3 [1] of
   RFC 7556 [RFC7556]. The "basic" [2] and "intermediate" [3] API
   support levels can in principle be implemented by means of layers
   wrapping the advanced API.

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

RFC 7556 [RFC7556] provides the essential conceptual guidance an API
designer would need to support use of PvDs. This document aims to
capture the requirements for an API that can be used by applications
that would be considered "advanced", according to section 6.3 [4] of
RFC 7556 [RFC7556]. The "basic" [5] and "intermediate" [6] API
support levels can in principle be implemented by means of layers
wrapping the advanced API.

This document also attempts to make some of the API implementation
requirements more concrete by discussion and example.

1.1. Requirements Language

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT",
"SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this
document are to be interpreted as described in RFC 2119 [RFC2119].

2. High level requirements

As described in section 2 [7] of RFC 7556 [RFC7556], a Provisioning
Domain ("PvD") is fundamentally a "consistent set of network
configuration information." This includes information like:

o the list of participating interfaces

o IPv4 and IPv6 addresses

o IPv4 and IPv6 routes: both default routes and more specifics (such
   as may be learned via RFC 4191 [RFC4191] Route Information Options
   ("RIOs"))

o DNS nameservers, search path, et cetera

o HTTP proxy configuration

and undoubtedly many more configuration elements yet to be specified
(like metering hints, transmission medium and speed, captive portal
URL, et cetera).

This configuration information as a whole may not be able to be
learned atomically, may need to be synthesized from multiple sources
including administrative provisioning, and cannot be presumed to be
unchanging over the lifetime of a node's association with a given
PvD.

In order for an application to make consistent use [8] of a given
PvD's network configuration several requirements are placed upon the
API itself and the host operating system providing the API.

2.1. Requirements for an API

At the highest level, the requirements for an API that enables
applications to make sophisticated use of multiple PvDs amount to
providing mechanisms by which they can:

R1 observe accessible PvDs

It MUST be possible for an application to be informed of the set
of all PvDs it can currently access, and to be informed of
changes to this set.

R2   observe configuration elements of an accessible PvD

It MUST be possible to learn requested configuration information
of any accessible PvD, and to be informed of any changes to the
configuration information comprising an accessible PvD.

R3   scope networking functionality to a specified PvD

For every existing API function that interacts with the node's
networking stack, be it at a relatively high level like
getaddrinfo() [9] or at the level of something like Sockets API's
sendmsg(), there MUST be a means by which an application can
specify the PvD within which networking operations are to be
restricted.

R4   use one and only specified scope per networking functionality
     invocation

For every unique invocation of a networking API function, there
MUST only be one specified PvD to which networking functionality
is to be restricted. At any given point in an application's
lifetime there MAY be several encapsulating layers of unspecified
PvDs (Section 3.2) through which the implementation must
progressively search to find a specified PvD, but ultimately a
networking function MUST use one and only one PvD for its
operations, even if that PvD is a "null PvD" (Section 3.3).

R5   make consistent use of programmatic references to PvDs

For uniformity and simplicity, every PvD-aware API functional
element SHOULD use (as return values of function calls, function
arguments, et cetera) the same programmatic reference for PvDs,
e.g. a construct containing a PvD identifier [10] or some
equivalent shorthand reference token (see Section 4.6 for a
discussion of implementation considerations). Regardless of the
implementation strategy chosen, a given programmatic reference
MUST remain constant over the lifetime of the node's continuous
attachment to the PvD to which it refers (until a disconnection
or disassociation event occurs). Additionally, references MAY
change with successive re-associations to the same PvD whereas
PvD identifiers, by definition, will not.

It is important to note that there is always a provisioning domain
within which networking functionality is scoped. For simply-

connected hosts this may be the implicit PvD [11] created by a single
networking interface connected to a traditional, shared LAN segment.
For multihomed hosts the "default provisioning domain" is likely a
matter of policy, but MAY be a "null" PvD, i.e. one completely devoid
of networking configuration information (no addresses, no routes, et
cetera). See Section 3 for further discussion.

The utility of such an API (allowing applications to learn of and
control the scope of networking functionality) suggests that the
Provisioning Domain is perhaps a more useful operational definition
for the original IPv6 concept of a "site-local scope" than the ill-
fated [RFC3879], "ill-defined concept" [12] of a site. It also
suggests one possible way by which operating system support for a
PvD-aware API might be implemented.

2.2. Requirements for supporting operating systems

The multiple PvD model of host behaviour is perhaps closer to the
Strong End System Model than the Weak End System Model characterized
in RFC 1122 [RFC1122] section 3.3.4.2 [13], but owing to its
recognition of a many-to-many relationship between interfaces and
PvDs should be considered a unique model unto itself.

In the PvD-aware End System Model, the "two key requirement issues
related to multihoming" are restated as:

a. A host MAY silently discard an incoming datagram whose
    destination address does not correspond to any PvD associated
    with the physical (or virtual) interface through which it is
    received.

b. A host MUST restrict itself to sending (non-source-routed) IP
    datagrams only through the physical (or virtual) interfaces that
    correspond to the PvD associated with the IP source address of
    the datagrams.

In order to support a PvD-aware application's use of multiple PVDs,
several additional requirements must be met by the host operating
system, especially when performing functions on behalf of
applications or when no direct application intervention is possible,
as discussed in the following sections.

2.2.1. Source address selection

Whenever a source address is to be selected on behalf of an
application it is essential for consistent use that only source
addresses belonging to the specified PvD be used a candidate set.
(See RFC 6418 [RFC6418] section 3.5 [14] for references to issues
arising from poor source address selection.)

For nodes following the PvD-aware End System Model, RFC 6724
[RFC6724] section 4 [15] is amended as follows:

R6 The candidate source addresses MUST be restricted to the set of
    unicast addresses associated with the concurrently specified PvD.

    Additionally, source address selection policies from PvDs other
    than the concurrently specified PvD MUST NOT be applied.

2.2.2. Route isolation

Whenever a routing lookup for a given destination is to be performed,
it is essential that only routes belonging to the currently specified
PvD be consulted. Applications and libraries that use the inherent
routing reachability check (and subsequent source address selection)
performed during something like the Sockets API connect() call on a
UDP socket to learn reachability information cheaply cannot function
correctly otherwise. RFC 6418 [RFC6418] section 4.2 [16] contains
more discussion and references to issues arising from insufficiently
isolated routing information.

For nodes following the PvD-aware End System Model:

R7 The set of routes consulted for any routing decision MUST be
    restricted to the routes associated with the concurrently
    specified PvD.

2.2.3. Automatic PvD metadata marking

In many cases, an application can examine a source address or the
destination address of a received datagram and use that address's
association with a given PvD to learn, for example, the PvD with
which an incoming connection may be associated. It may, however, be
impossible for an application to make this determination on its own
if, for example, an incoming TCP connection is destined to a RFC 1918
[RFC1918] address that happens to be configured in multiple PvDs at
the same time. In such circumstances, the supporting operating
system will need to provide additional assistance.

For nodes following the PvD-aware End System Model:

R8 When performing networking functionality on behalf of an
    application, the supporting operating system MUST record and make
    available to the application either (1) all the information the
    application might need to make a determination of the applicable
    PvD on its own or (2) the API's PvD programmatic reference
    directly.

    A supporting operating system SHOULD record and make available
    the API's PvD programmatic reference; other approaches invite
    ambiguity among applications' interpretation of available
    information.

2.2.4. Additional system and library support

Frequently, operating systems have several additional supporting
libraries and services for more advance networking functionality.
Using the system's own PvD API, and fulfilling the above
requirements, it should be possible to extend these services to
provide correct per-PvD isolation of information and enable
consistent application use of PvDs.

3. Conceptual PvDs

3.1. The 'default' PvD

Because there is always one specified provisioning domain to which an
individual invocation of networking functionality is restricted
(Section 2.1) there must necessarily exist a system "default PvD".
This provisioning domain is the one which networking functionality
MUST use when no other specified PvD can be determined.

Using the system's default PvD enables support of basic [17] uses of
the PvD API (i.e. backward compatibility for unmodified
applications).

The operating system MAY change the default PvD accordingly to
policy. It is expected that nodes will use a variety of information,
coupled with administrative policy, to promote one of any number of
concurrently available PvDs to be the system's default PvD.

R9 A PvD-aware API implementation MUST include a mechanism for
    applications to learn the programmatic reference to the system's
    concurrent default PvD.

R10 A PvD-aware API implementation SHOULD contain a mechanism
    enabling an application to be notified of changes to the
    concurrent default PvD in a comparatively efficient manner (i.e.
    more efficient than polling).

3.2. The 'unspecified' PvD

   An application may at some times wish to be specific about which PvD
   should be used for networking operations and at other times may
   prefer to defer the choice of specific PvD to one specified elsewhere
   (including the system default PvD).

   For example, if an application has specified the PvD to be used for
   all functions called by its process and child processes
   ([Section 4.3](#)), it may indicate that certain invocations should
   instead use the system default PvD by using a programmatic reference
   to the "unspecified PvD".

   R11 API implementors MUST reserve a programmatic reference to
       represent an "unspecified PvD": an indication that the
       application defers the selection of a specific PvD.

   R12 When invoked without a specific PvD, or with a programmatic
       reference to the "unspecified PvD", networking functionality MUST
       find a specific PvD to be used by examining the successive
       encapsulating layers of possible specificity supported by the API
       ([Section 4.3](#)), e.g. look first for a "fiber-specific default"
       PvD, then a "thread-specific default" PvD, a "process-specific
       default" PvD, and ultimately use the system's default PvD if no
       other specified PvD can be found.

3.3. The 'null' PvD

   If there are no PvDs accessible to an application, whether as a
   matter of policy (insufficient privileges) ([Section 4.5](#)) or as a
   matter of natural circumstance (the node is not connected to any
   network), the construct of a 'null' PvD may be useful to ensure
   networking functions fail (and fail quickly).

   R13 API implementors MAY reserve a programmatic reference to
       represent a "null PvD": an unchanging provisioning domain devoid
       of any and all networking configuration information.

   It is possible for operating systems to enforce that only PvD-aware
   applications may function normally by administratively configuring
   the default PvD to be the "null PvD".

3.4. The 'loopback' PvD

   TBD: is it useful to have a "loopback" PvD, i.e. one consisting
   solely of all addresses configured on the node and all locally
   delivered routes?

4. Requirements for new API functionality

4.1. Learning PvD availability

   R14 A PvD-aware API MUST implement a mechanism whereby an application
       can receive a set of the API's PvD programmatic references
       representing the complete set of PvDs (both explicit [18] and
       implicit [19]) with which the node is currently associated.

   R15 A PvD-aware API implementation SHOULD contain a mechanism
       enabling an application to be notified of changes in the above
       set of actively associated PvDs in a comparatively efficient
       manner (i.e. more efficient than polling).

   In may also be of use to applications to receive notifications of
   pending changes to the set of currently connected PvDs. For example,
   if it is known that a connection to a PvD is scheduled to be
   terminated shortly, an application may be able to take some
   appropriate action (migrate connections to another PvD, send
   notifications, et cetera).

4.2. Learning network configuration information comprising a PvD

   R16 A PvD-aware API MUST include a mechanism whereby by an
       application, using the API's PvD programmatic reference, can
       receive elements of the network configuration information that
       comprise a PvD. At a minimum, this mechanism MUST be capable of
       answering queries for:

       * the PvD identifier

       * all participating interfaces

       * all IPv4 and all non-deprecated IPv6 addresses

       * all configured DNS nameservers

   A PvD's network configuration information is neither guaranteed to be
   learned atomically nor is it guaranteed to be static. Addresses,
   routes, and even DNS nameservers and participating interfaces may
   each change over the lifetime of the node's association to a given
   PvD. Timely notification of such changes may be of particular
   importance to some applications.

R17 A PvD-aware API implementation SHOULD contain a mechanism
    enabling an application to be notified of changes in the
    networking configuration information comprising a PvD in a
    comparatively efficient manner (i.e. more efficient than
    polling).

R18 A network configuration query API implementation SHOULD take
    extensibility into account, to support querying for configuration
    information not yet conceived of with minimal adverse impact to
    applications.

4.3. Scoping functionality to a specific PvD

R19 A PvD-aware API implementation MUST include a mechanism for an
    application to specify the programmatic reference of the PvD to
    which all networking functionality MUST be restricted when not
    otherwise explicitly specified (a configurable, application-
    specific "default PvD").

R20 The API implementation MUST support setting such a "default PvD"
    for an application's entire process (and by extension its child
    processes). Additionally, the API SHOULD support an application
    setting a "default PvD" at every granularity of "programming
    parallelization", i.e. not only per-process, but also per-thread,
    per-fiber, et cetera. At every supported layer of granularity,
    if no PvD reference has been set the next coarser layer's setting
    MUST be consulted (up to and including the system's default PvD)
    when identifying the specified PvD to be used.

R21 For every degree of granularity at which an application may
    specify a "default PvD" there MUST exist a corresponding
    mechanism to retrieve any concurrently specified implementation-
    specific PvD programmatic reference. If no PvD has been
    specified for at the granularity of a given query, the
    "unspecified PvD" must be returned.

With access to this functionality it is possible to start non-PvD-
aware applications within a single PvD context with no adverse
impact. Furthermore, with judicious use of a sufficiently granular
API, existing general purpose networking APIs can be wrapped to
appear PvD-aware.

4.4. Explicit versus Implicit PvDs

R22 Because programmatic references to PvDs are returned for both
    explicit and implicit PvDs, the MPvD API implementation MUST be
    equally applicable and useful for any valid type of PvD; it MUST
    NOT be necessary for a PvD-aware application to distinguish
    between explicit and implicit PvDs to function properly.

4.5. Policy restrictions

This document does not make recommendations about policies governing
the use of any or all elements of a PvD API, save only to note that
some restrictions on use may be deemed necessary or appropriate.

R23 A PvD API implementation MAY implement policy controls whereby
    access to PvD availability information, configuration elements,
    and/or explicit scoping requests is variously permitted or denied
    to certain applications.

4.6. Programmatic reference implementation considerations

PvD identifiers may be of a length or form not easily handled
directly in some programming environments, and unauthenticated PvD
identifiers are assumed to be only probabilistically unique [20]. As
such, API implementations should consider using some alternative
programmatic reference (a node-specific "handle" or "token"), which
is fully under the control of the operating system, to identify an
instance of a single provisioning domain's network configuration
information.

Even though a PvD identifier may uniquely correspond to, say, a
network operator, there is no guarantee that the configuration
information (delegated prefixes, configured IP addresses, and so on)
will be the same with every successive association to the same PvD
identifier. An implementation may elect to change the value of the
programmatic reference to a given PvD identifier for each temporally
distinct association. Doing so presents some advantages worth
considering:

   Collisions in the PvD identifier space will inherently be treated
   as distinct by applications not concerned solely with identifiers.

   Changing the value of a reference can disabuse application writers
   of inappropriately caching configuration information from one
   association instance to another.

   Whether two PvDs are "identical" is perhaps better left to
   applications to decide since "PvD equivalence" for a given
   application may alternatively be determined by successfully
   accessing some restricted resource.

This document makes no specific requirement on the type of
programmatic reference used by the API.

5. Existing networking APIs

5.1. Updating existing APIs

From the perspective of a PvD-aware operating system, all previously
existing non-PvD-enabled networking functionality had historically
been executed within the context of a single, implicit provisioning
domain. A sufficiently granular API to specify which PvD is to be
used to scope subsequent networking functionality (Section 4.3) can
be used to wrap non-PvD-aware APIs, giving them this new PvD-aware
capability. However,

R24 Operating system implementors SHOULD consider updating existing
    networking APIs to take or return programmatic references to PvDs
    directly.

This may mean creating new functions with an additional PvD
programmatic reference argument, adding a PvD programmatic reference
field to an existing structure or class that is itself an argument or
return type, or finding other means by which to use a programmatic
reference with minimal or no disruption to existing applications or
libraries.

5.2. Requirements for name resolution APIs

RFC 3493 [RFC3493] getaddrinfo() [21] and getnameinfo() [22] APIs
deserve explicit discussion. Previously stated requirements make it
clear that it MUST be possible for an application to perform normal
name resolution constrained to the DNS configuration within a
specified PVD. This MUST be possible using at least the techniques
of Section 4.3.

The following additional requirements are places on PvD-aware
implementations of these functions:

R25 All DNS protocol communications with a PvD's nameservers MUST be
    restricted to use only source addresses and routes associated
    with the PvD.

R26 If getaddrinfo() is called with the AI_ADDRCONFIG flag specified,
    IPv4 addresses shall be returned only if an IPv4 address is
    configured within the specified provisioning domain and IPv6
    addresses shall be returned only if an IPv6 address is configured
    within the specified provision domain. The loopback address is
    (still) not considered for this case as valid as a configured
    address.

6. Acknowledgements

The core concepts presented in this document were developed during
the Android multinetworking effort by Lorenzo Colitti, Robert
Greenwalt, Paul Jensen, and Sreeram Ramachandran.

Additional thanks to the coffee shops of Tokyo.

7. IANA Considerations

This memo includes no request to IANA.

8. Security Considerations

An important new security impact of a PvD-aware API is that it
becomes much simpler (by design) to write a well-functioning
application to create a bridging data path between two PvDs that
would not otherwise have been so easily connected.

For some operating systems, existing APIs already make this bridging
possible, though some functionality like DNS resolution may have been
difficult to implement. Indeed, the very aim of an MPvD API is to
make implementing a PvD-aware application simple and to make its
functioning more "correct" ("first class" support for such
functionality).

Operating system implementations have several points of potential
policy control including:

o use of certain PvDs MAY be restricted by policy (e.g. only
   approved users, groups, or applications might be permitted
   access), and/or

o use of more than one PvD (or the MPvD API itself) MAY be similarly
   restricted.

9. References

9.1. Normative References

   [RFC1122] Braden, R., Ed., "Requirements for Internet Hosts -
              Communication Layers", STD 3, RFC 1122, DOI 10.17487/
              RFC1122, October 1989,
              <http://www.rfc-editor.org/info/rfc1122>.

   [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate
              Requirement Levels", BCP 14, RFC 2119, March 1997.

   [RFC3493] Gilligan, R., Thomson, S., Bound, J., McCann, J., and W.
              Stevens, "Basic Socket Interface Extensions for IPv6", RFC
              3493, DOI 10.17487/RFC3493, February 2003,
              <http://www.rfc-editor.org/info/rfc3493>.

   [RFC6724] Thaler, D., Ed., Draves, R., Matsumoto, A., and T. Chown,
              "Default Address Selection for Internet Protocol Version 6
              (IPv6)", RFC 6724, DOI 10.17487/RFC6724, September 2012,
              <http://www.rfc-editor.org/info/rfc6724>.

   [RFC7556] Anipko, D., Ed., "Multiple Provisioning Domain
              Architecture", RFC 7556, DOI 10.17487/RFC7556, June 2015,
              <http://www.rfc-editor.org/info/rfc7556>.

9.2. Informative References

   [RFC1918] Rekhter, Y., Moskowitz, R., Karrenberg, D., Groot, G., and
              E. Lear, "Address Allocation for Private Internets", BCP
              5, RFC 1918, February 1996.

   [RFC3879] Huitema, C. and B. Carpenter, "Deprecating Site Local
              Addresses", RFC 3879, DOI 10.17487/RFC3879, September
              2004, <http://www.rfc-editor.org/info/rfc3879>.

   [RFC4191] Draves, R. and D. Thaler, "Default Router Preferences and
              More-Specific Routes", RFC 4191, DOI 10.17487/RFC4191,
              November 2005, <http://www.rfc-editor.org/info/rfc4191>.

   [RFC6418] Blanchet, M. and P. Seite, "Multiple Interfaces and
              Provisioning Domains Problem Statement", RFC 6418, DOI
              10.17487/RFC6418, November 2011,
              <http://www.rfc-editor.org/info/rfc6418>.

Author's Address

Erik Kline
Google Japan KK
6-10-1 Roppongi
Mori Tower, 44th floor
Minato, Tokyo 106-6126
JP

Email: ek@google.com