Basic Socket Interface Extensions for
the Host Identity Protocol (HIP)

Abstract

   This document defines extensions to the current sockets API for the
   Host Identity Protocol (HIP). The extensions focus on the use of
   public-key-based identifiers discovered via DNS resolution, but also
   define interfaces for manual bindings between Host Identity Tags
   (HITs) and locators. With the extensions, the application can also
   support more relaxed security models where communication can be non-
   HIP-based, according to local policies. The extensions in this
   document are experimental and provide basic tools for further
   experimentation with policies.

Status of This Memo

   This document is not an Internet Standards Track specification; it is
   published for examination, experimental implementation, and
   evaluation.

   This document defines an Experimental Protocol for the Internet
   community. This document is a product of the Internet Engineering
   Task Force (IETF). It represents the consensus of the IETF
   community. It has received public review and has been approved for
   publication by the Internet Engineering Steering Group (IESG). Not
   all documents approved by the IESG are a candidate for any level of
   Internet Standard; see Section 2 of RFC 5741.

   Information about the current status of this document, any errata,
   and how to provide feedback on it may be obtained at
   http://www.rfc-editor.org/info/rfc6317.

Copyright Notice

Table of Contents

1. Introduction

   This document defines the C-based sockets Application Programming
   Interface (API) extensions for handling HIP-based identifiers
   explicitly in HIP-aware applications. It is up to the applications,
   or high-level programming languages or libraries, to manage the
   identifiers. The extensions in this document are mainly related to
   the use case in which a DNS resolution step has occurred prior to the
   creation of a new socket, and assumes that the system has cached or
   is otherwise able to resolve identifiers to locators (IP addresses).
   The DNS extension for HIP is described in [RFC5205]. The extensions
   also cover the case in which an application may want to explicitly
   provide suggested locators with the identifiers, including supporting
   the opportunistic case in which the system does not know the peer
   host identity.

   The Host Identity Protocol (HIP) [RFC4423] proposes a new
   cryptographic namespace by separating the roles of endpoint
   identifiers and locators by introducing a new namespace to the TCP/IP
   stack. Shim6 [RFC5533] is another protocol based on an identity-
   locator split. The APIs specified in this document are specific to
   HIP, but have been designed as much as possible to not preclude its
   use with other protocols. The use of these APIs with other protocols
   is, nevertheless, for further study.

   The APIs in this document are based on Host Identity Tags (HITs) that
   are defined as IPv6 addresses with the Overlay Routable Cryptographic
   Hash Identifiers (ORCHID) prefix [RFC4843]. ORCHIDs are derived from
   Host Identifiers using a hash and fitting the result into an IPv6
   address. Such addresses are called HITs, and they can be
   distinguished from other IPv6 addresses via the ORCHID prefix. Note
   that ORCHIDs are presently an experimental allocation by IANA. If
   the ORCHID allocation were to expire and HIT generation were to use a
   different prefix in the future, most users of the API would not be
   impacted, unless they explicitly checked the ORCHID prefix on
   returned HITs. Users who check (for consistency) that HITs have a
   valid ORCHID prefix must monitor the IANA allocation for ORCHIDs and
   adapt their software in case the ORCHID allocation were to be removed
   at a future date.

   Applications can observe the HIP layer and its identifiers in the
   networking stacks with varying degrees of visibility. [RFC5338]
   discusses the lowest levels of visibility in which applications are
   completely unaware of the underlying HIP layer. Such HIP-unaware
   applications in some circumstances use HIP-based identifiers, such as
   Local Scope Identifiers (LSIs) or HITs, instead of IPv4 or IPv6
   addresses and cannot observe the identifier-locator bindings.

This document specifies extensions to [RFC3493] to define a new
socket address family, AF_HIP. Similarly to other address families,
AF_HIP can be used as an alias for PF_HIP. The extensions also
describe a new socket address structure for sockets using HITs
explicitly and describe how the socket calls in [RFC3493] are adapted
or extended as a result.

Some applications may accept incoming communications from any
identifier. Other applications may initiate outgoing communications
without the knowledge of the peer identifier in opportunistic mode
(Section 4.1.6 of [RFC5201]) by just relying on a peer locator. This
document describes how to address both situations using "wildcards"
as described in Section 4.1.1.

This document references one additional API document [RFC6316] that
defines multihoming and explicit-locator handling. Most of the
extensions defined in this document can be used independently of the
above document.

The identity-locator split introduced by HIP introduces some policy-
related challenges with datagram-oriented sockets, opportunistic
mode, and manual bindings between HITs and locators. The extensions
in this document are of an experimental nature and provide basic
tools for experimenting with policies. Policy-related issues are
left for further experimentation.

To recap, the extensions in this document have three goals. The
first goal is to allow HIP-aware applications to open sockets to
other hosts based on the HITs alone, presuming that the underlying
system can resolve the HITs to addresses used for initial contact.
The second goal is that applications can explicitly initiate
communications with unknown peer identifiers. The third goal is to
illustrate how HIP-aware applications can use the Shim API [RFC6316]
to manually map locators to HITs.

This document was published as experimental because a number of its
normative references had experimental status. The success of this
experiment can be evaluated by a thorough implementation of the APIs
defined.

2. Terminology

   The terms used in this document are summarized in Table 1.

   +---------+---+
   | Term | Explanation |
   +---------+---+
FQDN	Fully Qualified Domain Name
HIP	Host Identity Protocol
HI	Host Identifier
HIT	Host Identity Tag, a 100-bit hash of a public key with
	a 28-bit prefix
LSI	Local Scope Identifier, a local, 32-bit descriptor for
	a given public key
Locator	Routable IPv4 or IPv6 address used at the lower layers
RR	Resource Record
   +---------+---+

                                 Table 1

3. Name Resolution Process

   This section provides an overview of how the API can be used. First,
   the case in which a resolver is involved in name resolution is
   described, and then the case in which no resolver is involved is
   described.

3.1. Interaction with the Resolver

   Before an application can establish network communications with the
   entity named by a given FQDN or relative hostname, the application
   must translate the name into the corresponding identifier(s). DNS-
   based hostname-to-identifier translation is illustrated in Figure 1.
   The application calls the resolver in step (a) to resolve an FQDN to
   one or more socket addresses within the PF_HIP family. The resolver,
   in turn, queries the DNS in step (b) to map the FQDN to one or more
   HIP RRs with the HIT and HI and possibly the rendezvous server of the
   Responder, and also (in parallel or sequentially) to resolve the FQDN
   into possibly one or more A and AAAA records. It should be noted
   that the FQDN may map to multiple Host Identifiers and locators, and
   this step may involve multiple DNS transactions, including queries
   for A, AAAA, HI, and possibly other resource records. The DNS server
   responds with a list of HIP resource records in step (c).
   Optionally, in step (d), the resolver caches the HIT-to-locator
   mapping with the HIP module. The resolver converts the HIP records
   to HITs and returns the HITs to the application contained in HIP
   socket address structures in step (e). Depending on the parameters
   for the resolver call, the resolver may also return other socket

address structures to the application. Finally, the application
receives the socket address structure(s) from the resolver and uses
them in socket calls such as connect() in step (f).

```
                                            +----------+
                                            |          |
                                            |   DNS    |
                                            |          |
                                            +----------+
                                               ^  |
                           b. QNAME=FQDN  |  | c. HIP and
                                          |  |    A/AAAA
                                          |  v    RR(s)
       +--------------+ a. getaddrinfo(<FQDN>)  +----------+
       |              |------------------------>|          |
       | Application  |                         | Resolver |
       |              |<------------------------|          |
       +--------------+     e. <HITs>           +----------+
              |                                      |
              |                                      | d. HIP and
              | f. connect(<HIT>)                    |    A/AAAA
              |    or any other socket call          |    RR(s)
              v                                      v
       +----------+                         +----------+
       |          |                         |          |
       | TCP/IP   |                         |   HIP    |
       | Stack    |                         |          |
       +----------+                         +----------+
```
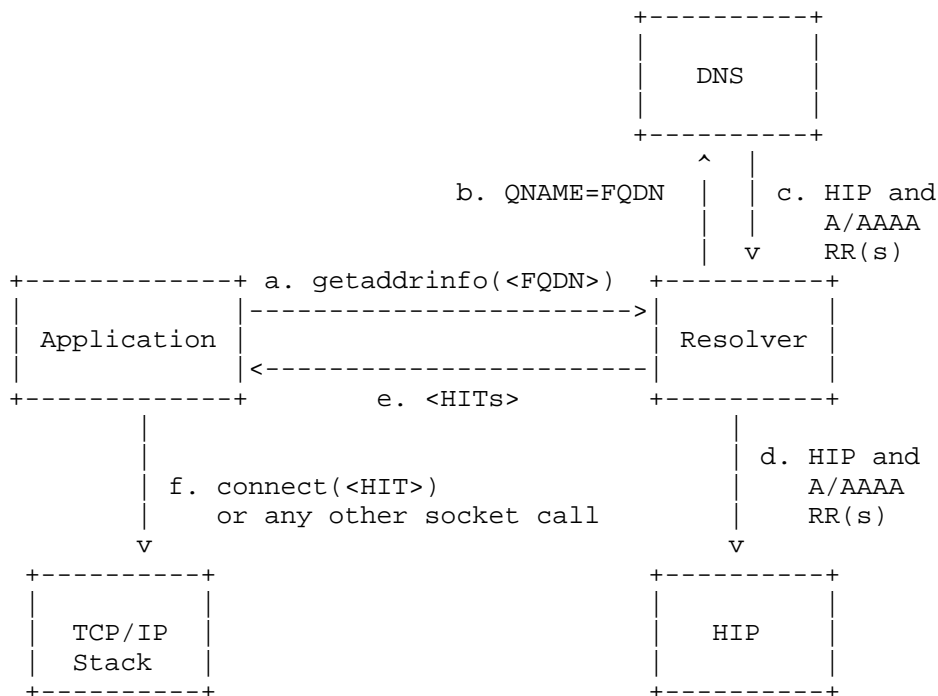
Figure 1

In practice, the resolver functionality can be implemented in
different ways. For example, it may be implemented in existing
resolver libraries or as a HIP-aware interposing agent.

3.2. Interaction without a Resolver

The extensions in this document focus on the use of the resolver to
map hostnames to HITs and locators in HIP-aware applications. The
resolver may implicitly associate a HIT with the corresponding
locator(s) by communicating the HIT-to-IP mapping to the HIP daemon.
However, it is possible that an application operates directly on a
peer HIT without interacting with the resolver. In such a case, the

application may resort to the system to map the peer HIT to an IP
address. Alternatively, the application can explicitly map the HIT
to an IP address using socket options as specified in Section 4.6.
Full support for all of the extensions defined in this document
requires a number of shim socket options [RFC6316] to be implemented
by the system.

4. API Syntax and Semantics

In this section, we describe the native HIP APIs using the syntax of
the C programming language. We limit the description to the
interfaces and data structures that are either modified or completely
new, because the native HIP APIs are otherwise identical to the
sockets API [POSIX].

4.1. Socket Family and Address Structure Extensions

The sockets API extensions define a new protocol family, PF_HIP, and
a new address family, AF_HIP. The AF_HIP and PF_HIP constants are
aliases to each other. These definitions shall be defined as a
result of including <sys/socket.h>.

When the socket() function is called with PF_HIP as the first
argument (domain), it attempts to create a socket for HIP
communication. If HIP is not supported, socket() follows its default
behavior and returns -1, and sets errno to EAFNOSUPPORT.

Figure 2 shows the recommended implementation of the socket address
structure for HIP in Portable Operating System Interface (POSIX)
format.

```
#include <netinet/hip.h>

typedef struct in6_addr hip_hit_t;

struct sockaddr_hip {
        uint8_t         ship_len;
        sa_family_t     ship_family;
        in_port_t       ship_port;
        uint32_t        ship_flags;
        hip_hit_t       ship_hit;
};
```

Figure 2

uint8_t ship_len: This field defines the length of the structure.
Implementations that do not define this field typically embed the
information in the following ship_family field.

sa_family_t ship_family: This mandatory field identifies the
structure as a sockaddr_hip structure. It overlays the sa_family
field of the sockaddr structure. Its value must be AF_HIP.

in_port_t ship_port: This mandatory field contains the transport
protocol port number. It is handled in the same way as the sin_port
field of the sockaddr_in structure. The port number is stored in
network byte order.

uint32_t ship_flags: This mandatory bit field contains auxiliary
flags. This document does not define any flags. This field is
included for future extensions.

hip_hit_t ship_hit: This mandatory field contains the endpoint
identifier. When the system passes a sockaddr_hip structure to the
application, the value of this field is set to a valid HIT, IPv4, or
IPv6 address, as discussed in Section 4.5. When the application
passes a sockaddr_hip structure to the system, this field must be set
to a HIT or a wildcard address as discussed in Section 4.1.1.

Some applications rely on system-level access control, either
implicit or explicit (such as the accept_filter() function found on
BSD-based systems), but such discussion is out of scope. Other
applications implement access control themselves by using the HITs.
Applications operating on sockaddr_hip structures can use memcmp() or
a similar function to compare the ship_hit fields. It should also be
noted that different connection attempts between the same two hosts
can result in different HITs, because a host is allowed to have
multiple HITs.

4.1.1. HIP Wildcard Addresses

   HIP wildcard addresses are similar to IPv4 and IPv6 wildcard
   addresses. They can be used instead of specific HITs in the ship_hit
   field for local and remote endpoints in sockets API calls such as
   bind(), connect(), sendto(), or sendmsg().

   In order to bind to all local IPv4 and IPv6 addresses and HIP HITs,
   the ship_hit field must be set to HIP_ENDPOINT_ANY. In order to bind
   to all local HITs, ship_hit must contain HIP_HIT_ANY. To only bind
   to all local public HITs, the ship_hit field must be HIP_HIT_ANY_PUB.
   The value HIP_HIT_ANY_TMP binds a socket to all local anonymous
   identifiers only as specified in [RFC4423]. The system may label
   anonymous identifiers as such depending on whether they have been

published or not. After binding a socket via one of the
HIP_HIT_ANY_* wildcard addresses, the application is guaranteed to
receive only HIP-based data flows. With the HIP_ENDPOINT_ANY
wildcard address, the socket accepts HIP, IPv6, and IPv4-based data
flows.

When a socket is bound or connected via a sockaddr_hip structure,
i.e., the PF_HIP protocol family, the system returns only addresses
of the AF_HIP family, i.e., sockaddr_hip structures, for this socket.
This applies to all functions that provide addresses to the
application, such as accept() or recvfrom(). If the data flow is
based on HIP, the ship_hit field contains the peer's HIT. For a
non-HIP IPv6 data flow, the field contains the peer's IPv6 address.
For a non-HIP IPv4 data flow, the field contains the peer's IPv4
address in IPv4-mapped IPv6 address format as described in
Section 3.7 of [RFC3493]. Section 4.5 describes how the application
can verify the type of address returned by the sockets API calls.

An application uses the sockets API as follows to set up a connection
or to send messages in HIP opportunistic mode (cf. [RFC5201]).
First, the application associates a socket with at least one IP
address of the destination peer via setting the
SHIM_LOCLIST_PEER_PREF socket option. It then uses outgoing socket
functions such as connect(), sendto(), or sendmsg() with the
HIP_ENDPOINT_ANY or HIP_HIT_ANY wildcard address in the ship_hit
field of the sockaddr_hip structure. With the HIP_HIT_ANY address,
the underlying system allows only HIP-based data flows with the
corresponding socket. For incoming packets, the system discards all
non-HIP-related traffic arriving at the socket. For outgoing
packets, the system returns -1 in the socket call and sets errno to
an appropriate error type when the system failed to deliver the
packet over a HIP-based data channel. The semantics of using
HIP_ENDPOINT_ANY are the subject of further experimentation in the
context of opportunistic mode. Such use may result in a data flow
either with or without HIP.

4.2. Extensions to Resolver Data Structures

The HIP APIs introduce a new address family, AF_HIP, that HIP-aware
applications can use to control the address type returned from the
getaddrinfo() function [RFC3493] [POSIX]. The getaddrinfo() function
uses a data structure called addrinfo in its "hints" and "res"
arguments, which are described in more detail in the next section.
The addrinfo data structure is illustrated in Figure 3.

```
#include <netdb.h>

struct addrinfo {
    int       ai_flags;         /* e.g., AI_CANONNAME */
    int       ai_family;        /* e.g., AF_HIP */
    int       ai_socktype;      /* e.g., SOCK_STREAM */
    int       ai_protocol;      /* 0 or IPPROTO_HIP */
    socklen_t ai_addrlen;       /* size of *ai_addr  */
    struct    sockaddr *ai_addr; /* sockaddr_hip */
    char      *ai_canonname;    /* canon. name of the host */
    struct    addrinfo *ai_next; /* next endpoint */
    int       ai_eflags;        /* RFC 5014 extension */
};
```

                              Figure 3

   An application resolving with the ai_family field set to AF_UNSPEC in
   the hints argument may receive any kind of socket address structures,
   including sockaddr_hip. When the application wants to receive only
   HITs contained in sockaddr_hip structures, it should set the
   ai_family field to AF_HIP. Otherwise, the resolver does not return
   any sockaddr_hip structures. The resolver returns EAI_FAMILY when
   AF_HIP is requested but not supported.

   The resolver ignores the AI_PASSIVE flag when the application sets
   the family in hints to AF_HIP.

   The system may have a HIP-aware interposing DNS agent as described in
   Section 3.2 of [RFC5338]. In such a case, the DNS agent may,
   according to local policy, transparently return LSIs or HITs in
   sockaddr_in and sockaddr_in6 structures when available. A HIP-aware
   application can override this local policy in two ways. First, the
   application can set the family to AF_HIP in the hints argument of
   getaddrinfo() when it requests only sockaddr_hip structures. Second,
   the application can set the AI_NO_HIT flag to prevent the resolver
   from returning HITs in any kind of data structures.

   When getaddrinfo() returns resolved outputs in the output "res"
   argument, it sets the family to AF_HIP when the related structure is
   sockaddr_hip.

4.2.1. Resolver Usage

   A HIP-aware application creates the sockaddr_hip structures manually
   or obtains them from the resolver. The explicit configuration of
   locators is described in [RFC6316]. This document defines

"automated" resolver extensions for the getaddrinfo() resolver
[RFC3493]. Other resolver calls, such as gethostbyname() and
getservbyname(), are not defined in this document. The getaddrinfo()
resolver interface is shown in Figure 4.

```
#include <netdb.h>

int getaddrinfo(const char *nodename,
                const char *servname,
                const struct addrinfo *hints,
                struct addrinfo **res)
void free_addrinfo(struct addrinfo *res)
```

Figure 4

As described in [RFC3493], the getaddrinfo() function takes nodename,
servname, and hints as its input arguments. It places the result of
the query into the res output argument. The return value is zero on
success, or a non-zero error value on error. The nodename argument
specifies the hostname to be resolved; a NULL argument denotes the
HITs of the local host. The servname parameter declares the port
number to be set in the socket addresses in the res output argument.
The nodename and servname arguments cannot both be NULL at the same
time.

The input argument "hints" acts like a filter that defines the
attributes required from the resolved endpoints. A NULL hints
argument indicates that any kind of endpoint is acceptable.

The output argument "res" is dynamically allocated by the resolver.
The application frees the res argument with the free_addrinfo
function. The res argument contains a linked list of the resolved
endpoints. The linked list contains only sockaddr_hip structures
when the input argument has the family set to AF_HIP. When the
family is zero, the list contains sockaddr_hip structures before
sockaddr_in and sockaddr_in6 structures.

The resolver can return a HIT that maps to multiple locators. The
resolver may cache the locator mappings with the HIP module. The HIP
module manages the multiple locators according to system policies of
the host. The multihoming document [RFC6316] describes how an
application can override system default policies.

It should be noted that the application can configure the HIT
explicitly without setting the locator, or the resolver can fail to
resolve any locator. In this scenario, the application relies on the
system to map the HIT to an IP address. When the system fails to
provide the mapping, it returns -1 in the called sockets API function
to the application and sets errno to EADDRNOTAVAIL.

4.3. The Use of getsockname() and getpeername() Functions

The sockaddr_hip structure does not contain a HIT when the
application uses the HIP_HIT_ANY_* or HIP_ENDPOINT_ANY constants. In
such a case, the application can discover the local and peer HITs
using the getsockname() and getpeername() functions after the socket
is connected. The functions getsockname() and getpeername() always
output a sockaddr_hip structure when the family of the socket is
AF_HIP. The application should be prepared to also handle IPv4 and
IPv6 addresses in the ship_hit field, as described in Section 4.1, in
the context of the HIP_ENDPOINT_ANY constant.

4.4. Selection of Source HIT Type

A client-side application can choose its source HIT by, for example,
querying all of the local HITs with getaddrinfo() and associating one
of them with the socket using bind(). This section describes another
method for a client-side application to affect the selection of the
source HIT type where the application does not call bind()
explicitly. Instead, the application just specifies the preferred
requirements for the source HIT type.

The sockets API for source address selection [RFC5014] defines socket
options to allow applications to influence source address selection
mechanisms. In some cases, HIP-aware applications may want to
influence source HIT selection, in particular whether an outbound
connection should use a published or anonymous HIT. Similar to
IPV6_ADDR_PREFERENCES defined in [RFC5014], the socket option
HIT_PREFERENCES is defined for HIP-based sockets. This socket option
can be used with setsockopt() and getsockopt() calls to set and get
the HIT selection preferences affecting a HIP-enabled socket. The
socket option value (optval) is a 32-bit unsigned integer argument.
The argument consists of a number of flags where each flag indicates
an address selection preference that modifies one of the rules in the
default HIT selection; these flags are shown in Table 2.

```
            +--------------------------+------------------------+
            | Socket Option            | Purpose                |
            +--------------------------+------------------------+
            | HIP_PREFER_SRC_HIT_TMP   | Prefer an anonymous HIT |
            | HIP_PREFER_SRC_HIT_PUBLIC | Prefer a public HIT    |
            +--------------------------+------------------------+
```

                               Table 2

   If the system is unable to assign the type of HIT that is requested,
   at HIT selection time, the socket call (connect(), sendto(), or
   sendmsg()) will fail, and errno will be set to EINVAL. If the
   application tries to set both of the above flags for the same socket,
   this also results in the error EINVAL.

4.5. Verification of HIT Type

   An application that uses the HIP_ENDPOINT_ANY constant may want to
   check whether the actual communication was based on HIP or not.
   Also, the application may want to verify whether a HIT belonging to
   the local host is public or anonymous. The application accomplishes
   this using a new function called sockaddr_is_srcaddr(), which is
   illustrated in Figure 5.

        #include <netinet/hip.h>

        short sockaddr_is_srcaddr(struct sockaddr *srcaddr,
                                  uint64_t flags);

                               Figure 5

   The sockaddr_is_srcaddr() function operates in the same way as the
   inet6_is_srcaddr() function [RFC5014], which can be used to verify
   the type of an address belonging to the local host. The difference
   is that the sockaddr_is_srcaddr() function handles sockaddr_hip
   structures in addition to sockaddr_in6, and possibly other socket
   structures in further extensions. Also, the length of the flags
   argument is 64 bits instead of 32 bits, because the new function
   handles the same flags as defined in [RFC5014], in addition to two
   HIP-specific flags, HIP_PREFER_SRC_HIT_TMP and
   HIP_PREFER_SRC_HIT_PUBLIC. With these two flags, the application can
   distinguish anonymous HITs from public HITs.

   When given an AF_INET6 socket, sockaddr_is_srcaddr() behaves the same
   way as the inet6_is_srcaddr() function as described in [RFC5014].
   With an AF_HIP socket, the function returns 1 when the HIT contained
   in the socket address structure corresponds to a valid HIT of the
   local host and the HIT satisfies the given flags. The function

returns -1 when the HIT does not belong to the local host or the
flags are not valid. The function returns 0 when the preference
flags are valid but the HIT does not match the given flags. The
function also returns 0 on a sockaddr_hip structure containing a
HIP_ENDPOINT_ANY or HIP_HIT_ANY_* wildcard.

The sockaddr_is_srcaddr() interface applies only to local HITs.
Applications can call the function hip_is_hit() to verify that the
given hit_hit_t pointer has the HIT prefix. The function is
illustrated in Figure 6.

        #include <netinet/hip.h>

        short hip_is_hit(hip_hit_t *hit);

                              Figure 6

The hip_is_hit() function returns 1 when the given argument contains
the HIT prefix. The function returns -1 on error and sets errno
appropriately. The function returns 0 when the argument does not
have the HIT prefix. The function also returns 0 when the argument
is a HIP_ENDPOINT_ANY or HIP_HIT_ANY_* wildcard.

4.6. Explicit Handling of Locators

The system resolver, or the HIP module, maps HITs to locators
implicitly. However, some applications may want to specify initial
locator mappings explicitly. In such a case, the application first
creates a socket with AF_HIP as the domain argument. Second, the
application may get or set locator information with one of the
following shim socket options as defined in the multihoming
extensions in [RFC6316]. The related socket options are summarized
briefly in Table 3.

```
+--------------------+--------------------------------------------+
| optname            | description                                |
+--------------------+--------------------------------------------+
| SHIM_LOC_LOCAL_PREF | Get or set the preferred locator on the   |
|                    | local side for the context associated with |
|                    | the socket.                                |
| SHIM_LOC_PEER_PREF | Get or set the preferred locator on the    |
|                    | remote side for the context associated with |
|                    | the socket.                                |
| SHIM_LOCLIST_LOCAL | Get or set a list of locators associated   |
|                    | with the local Endpoint Identifier (EID).  |
| SHIM_LOCLIST_PEER  | Get or set a list of locators associated   |
|                    | with the peer's EID.                       |
| SHIM_LOC_LOCAL_SEND | Set or get the default source locator of  |
|                    | outgoing IP packets.                       |
| SHIM_LOC_PEER_SEND | Set or get the default destination locator |
|                    | of outgoing IP packets.                    |
+--------------------+--------------------------------------------+
```

Table 3

As an example of locator mappings, a connection-oriented application
creates a HIP-based socket and sets the SHIM_LOCLIST_PEER socket
option on the socket. The HIP module uses the first address
contained in the option if multiple addresses are provided. If the
application provides one or more addresses in the SHIM_LOCLIST_PEER
setsockopt call, the system should not connect to the host via
another destination address, in case the application intends to
restrict the range of addresses permissible as a policy choice. The
application can override the default peer locator by setting the
SHIM_LOC_PEER_PREF socket option if necessary. Finally, the
application provides a specific HIT in the ship_hit field of the
sockaddr_hip in the connect() system call. If the system cannot
reach the HIT at one of the addresses provided, the outbound sockets
API functions (connect(), sendmsg(), etc.) return -1 and set errno to
EINVALIDLOCATOR.

Applications may also choose to associate local addresses with
sockets. The procedures specified in [RFC6316] are followed in this
case.

Another use case is to use the opportunistic mode when the
destination HIT is specified as a wildcard. The application sets one
or more destination addresses using the SHIM_LOCLIST_PEER socket
option as described earlier in this section, and then calls connect()
with the wildcard HIT. The connect() call returns -1 and sets errno
to EADDRNOTAVAIL when the application connects to a wildcard without
specifying any destination address.

Applications using datagram-oriented sockets can use ancillary data
to control the locators, as described in detail in [RFC6316].

5. Summary of New Definitions

Table 4 summarizes the new constants and structures defined in this
document.

```
+-----------------+-----------------------+
| Header          | Definition            |
+-----------------+-----------------------+
| <sys/socket.h>  | AF_HIP                |
| <sys/socket.h>  | PF_HIP                |
| <netinet/in.h>  | IPPROTO_HIP           |
| <netinet/hip.h> | HIP_HIT_ANY           |
| <netinet/hip.h> | HIP_HIT_ANY_PUB       |
| <netinet/hip.h> | HIP_HIT_ANY_TMP       |
| <netinet/hip.h> | HIP_ENDPOINT_ANY      |
| <netinet/hip.h> | HIP_HIT_PREFERENCES   |
| <netinet/hip.h> | hip_hit_t             |
| <netdb.h>       | AI_NO_HIT             |
| <netinet/hip.h> | sockaddr_hip          |
| <netinet/hip.h> | sockaddr_is_srcaddr() |
| <netinet/hip.h> | hip_is_hit()          |
+-----------------+-----------------------+
```

Table 4

6. Security Considerations

This document describes an API for HIP and therefore depends on the
mechanisms defined in the HIP protocol suite. Security concerns
associated with HIP itself are specified in [RFC4423], [RFC4843],
[RFC5201], [RFC5205], and [RFC5338].

The HIP_ENDPOINT_ANY constant can be used to accept incoming data
flows or create outgoing data flows without HIP. The application
should use the sockaddr_is_srcaddr() function to validate the type of
connection in order to, for example, inform the user of the lack of
HIP-based security. The use of the HIP_HIT_ANY_* constants is
recommended in security-critical applications and systems.

It should be noted that the wildcards described in this document are
not suitable for identifying end hosts. Instead, applications should
use getsockname() and getpeername() as described in Section 4.3 to
identify an end host.

Future proofing of HITs was discussed during the design of this API.
If HITs longer than 128 bits are required at the application layer,
this will require explicit support from the applications, because
they can store or cache HITs with their explicit sizes. To support
longer HITs, further extensions of this API may define an additional
flag for getaddrinfo() to generate different kinds of socket address
structures for HIP.

7. Contributors

Thanks to Jukka Ylitalo and Pekka Nikander for their original
contributions, time, and effort to the native HIP APIs. Thanks to
Yoshifuji Hideaki and Stefan Goetz for their contributions to this
document.

8. Acknowledgments

Kristian Slavov, Julien Laganier, Jaakko Kangasharju, Mika Kousa, Jan
Melen, Andrew McGregor, Sasu Tarkoma, Lars Eggert, Joe Touch, Antti
Jarvinen, Anthony Joseph, Teemu Koponen, Jari Arkko, Ari Keranen,
Juha-Matti Tapio, Shinta Sugimoto, Philip Matthews, Joakim Koskela,
Jeff Ahrenholz, Tobias Heer, and Gonzalo Camarillo have provided
valuable ideas and feedback. Thanks to Nick Stoughton from the
Austin group for POSIX-related comments. Thanks also to the APPS
area folks, including Stephane Bortzmeyer, Chris Newman, Tony Finch,
"der Mouse", and Keith Moore.

9. References

9.1. Normative References

[POSIX]   "IEEE Std. 1003.1-2008 Standard for Information
             Technology -- Portable Operating System Interface
             (POSIX). Open group Technical Standard: Base
             Specifications, Issue 7", September 2008,
             <http://www.opengroup.org/austin>.

[RFC3493]   Gilligan, R., Thomson, S., Bound, J., McCann, J., and W.
             Stevens, "Basic Socket Interface Extensions for IPv6",
             RFC 3493, February 2003.

[RFC4423]   Moskowitz, R. and P. Nikander, "Host Identity Protocol
             (HIP) Architecture", RFC 4423, May 2006.

[RFC4843]   Nikander, P., Laganier, J., and F. Dupont, "An IPv6
             Prefix for Overlay Routable Cryptographic Hash
             Identifiers (ORCHID)", RFC 4843, April 2007.

   [RFC5014]   Nordmark, E., Chakrabarti, S., and J. Laganier, "IPv6
                Socket API for Source Address Selection", RFC 5014,
                September 2007.

   [RFC5201]   Moskowitz, R., Nikander, P., Jokela, P., Ed., and T.
                Henderson, "Host Identity Protocol", RFC 5201, April
                2008.

   [RFC5205]   Nikander, P. and J. Laganier, "Host Identity Protocol
                (HIP) Domain Name System (DNS) Extensions", RFC 5205,
                April 2008.

   [RFC5338]   Henderson, T., Nikander, P., and M. Komu, "Using the Host
                Identity Protocol with Legacy Applications", RFC 5338,
                September 2008.

   [RFC6316]   Komu, M., Bagnulo, M., Slavov, K., and S. Sugimoto, Ed.,
                "Sockets Application Program Interface (API) for
                Multihoming Shim", RFC 6316, July 2011.

9.2. Informative References

   [RFC5533]   Nordmark, E. and M. Bagnulo, "Shim6: Level 3 Multihoming
                Shim Protocol for IPv6", RFC 5533, June 2009.

Authors' Addresses

   Miika Komu
   Aalto University
   Espoo
   Finland

   Phone: +358505734395
   Fax: +358947025014
   EMail: miika@iki.fi
   URI: http://cse.aalto.fi/research/groups/datacommunications/people/

   Thomas Henderson
   The Boeing Company
   P.O. Box 3707
   Seattle, WA
   USA

   EMail: thomas.r.henderson@boeing.com