

Miika Komu

Host Identity Protocol Application Programming Interfaces

Helsinki University of Technology
Department of Computer Science and Engineering
Telecommunications Software and Multimedia Laboratory

Author:	Miika Komu	
Name of the thesis:	Host Identity Protocol Application Programming Interfaces	
Date:	8th July 2004	Number of pages: 83
Department:	Department of Computer Science and Engineering	
Professorship:	T-110	
Supervisor:	Ph.D. Kimmo Raatikainen	
Instructors:	Ph.D. Pekka Nikander, M.Sc. Jukka Ylitalo	
<p>The goal of this thesis is to design and implement an API for HIP aware network applications using C language. The result of the design is evaluated against the requirements. Different design alternatives are introduced and analyzed in order to rationalize the design. A reference implementation is developed as a proof of concept. The API is experimented by porting a few example applications to use the API.</p> <p>The outcome of the design meets the requirements. The API follows the design of the sockets API closely and extends it only when reuse is not possible. The new API increases the control over the HIP layer for the advanced applications. Applications can also specify their own endpoint identities. Typical applications can utilize the API in a simple way that hides the details of the endpoint identifiers and locators. A HIP enabled application can fall back to plain TCP/IP seamlessly if the peer host does not support HIP.</p> <p>The API work brought up some future work items. The API may also be useful to other protocols based on the identity-locator split. A Quality of Service and a mobility event API need to be specified. FTP and other applications using “referrals” require also further work.</p>		
<p>Keywords: HIP, native, API, EID, socket, legacy, userspace, application</p>		

Tekijä:	Miika Komu	
Työn nimi:	Host Identity Protocol sovellusrajapinnat	
Päivämäärä:	8. heinäkuuta 2004	Sivuja: 83
Osasto:	Tietotekniikan osasto	
Professuuri:	T-110	
Työn valvoja:	Ph.D. Kimmo Raatikainen	
Työn ohjaajat:	Ph.D. Pekka Nikander, M.Sc. Jukka Ylitalo	
<p>Diplomityön tavoitteena on suunnitella ja toteuttaa sovellusrajapinta HIP tietoisille sovelluksille käyttäen C ohjelmointikieltä. Rajapinta arvoidaan asetettuja vaatimuksia vasten. Erilaisia toteutusmalleja esitellään valitun toteutuksen perustelemiseksi. Mallitoteutus kehitetään mallin realistisuuden toteamiseksi.</p> <p>Rajapintamalli täyttää asetetut vaatimukset. Rajapinta on virtaviivainen sockets API:n kanssa ja laajentaa sitä ainoastaan silloin, kun muu ei ole mahdollista. Sovellusrajapinta lisää edistyneempien verkkosovelluksien hallintamahdollisuuksia HIP tasoon. Sovellukset voivat määrittellä myös määrittellä oman identiteettinsä. Tavalliset sovellukset voivat hyödyntää rajapintaa yksinkertaisin keinoin puuttumatta identiteettien ja lokaattoreiden yksityiskohtiin. HIP:ia tukeva sovellus voi siirtyä saumattomasti tavalliseen TCP/IP kommunikointiin, jos vastapää ei tue HIP:ia.</p> <p>Rajapinnan kehittäminen toi muutamia jatkokehitysideoita. Rajapinta voi olla hyödyllinen myös muille protokollille, jotka perustuvat identiteetin ja lokaattorin erottamiseen toisistaan. Quality of Service ja mobiliteettitapahtumarajapinta täytyy määrillä. FTP ja muut sovellukset, jotka käyttävät "referral":a, kaipaavat myös lisätutkimusta.</p>		
Avainsanat: HIP, native, API, rajapinta, EID, socket, legacy, userspace, sovellus		

Acknowledgements

I want to thank Jukka Ylitalo for his time and effort for the design brainstorming sessions and excellent feedback. The credit for EID concept goes for Pekka Nikander as the idea originated from him, not me. Kristian Slavov gave me some corrections to the implementation and analysis chapters. It was also fun to brainstorm with you! Julien Laganier gave me some corrections to the design and analysis. He noticed that the `HIP_HI_ANY` macro was missing from the design. Jaakko Kangasharju provided some comments to the design and analysis chapters. He also gave me some corrections to the spelling and outlook. Mika “Berner” Kousa corrected some spelling errors too. Jan Melen gave some refinements to the design section and claimed that the API is implementable with their implementation too. I challenge you to do that! Thomas Henderson gave some comments on the structure and contents of the thesis when it was just a draft. I also wish to thank Andrew McGregor for some fruitful discussion sessions on the API. Sasu Tarkoma also gave some comments on the structure of the thesis.

Helsinki, 8th July 2004

Miika Komu

Contents

Terms and Abbreviations	viii
1 Introduction	1
1.1 Problem Statement	2
1.2 Scope	2
2 Background	3
2.1 Mobility Related Terminology	3
2.2 Host Identity Protocol	4
2.2.1 Restrictions in TCP/IP	4
2.2.2 A New Namespace	4
2.2.3 A New Layer	5
2.2.4 Mobility and Multihoming	6
2.3 Related APIs	7
2.3.1 Sockets API	7
2.3.2 SCTP Socket API Extensions	13
2.3.3 Legacy HIP API	14
3 Requirements	16
3.1 Non-functional Requirements	16
3.1.1 Usability	16
3.1.2 Compatibility	17
3.2 Functional Requirements	18
3.2.1 Host Identities	18
3.2.2 Addresses and Interfaces	20
3.2.3 Mobility, Multihoming and Policies	20

3.2.4	Security	21
3.2.5	Error Handling	21
3.2.6	Directories	21
3.3	Evaluation	22
4	Design	23
4.1	Architecture	23
4.1.1	Endpoint Identifier Descriptor	23
4.1.2	Layering Model	23
4.1.3	Namespace Model	24
4.1.4	Socket Bindings	25
4.1.5	Endpoint Discovery	26
4.2	Interface Syntax and Description	26
4.2.1	Data Structures	27
4.2.2	Functions	29
5	Implementation	34
5.1	Userspace Components	34
5.2	Kernelspace Components	34
5.3	HIP Networking Stack Hooks	35
5.4	Data Structures	36
5.5	Collaboration of Components	37
5.5.1	Connection Setup on the Server	38
5.5.2	Connection Setup on the Client	39
5.5.3	Sending Data	40
5.5.4	Receiving Data	41
5.6	HIP Enabled Telnet	41
6	Analysis	43
6.1	Evaluation	43
6.1.1	Socket Family	43
6.1.2	Endpoint Identifier Descriptor	44
6.1.3	Application Specified Identifiers	45
6.1.4	Resolver	45

6.2	Design Alternatives	46
6.2.1	IP Address Policy Based Approach	46
6.2.2	Host Identifier Based Approach	47
6.2.3	Shared Data structure for Identifier and Locator	47
6.2.4	Endpoint Identity Descriptor Based Binding Model	48
6.2.5	Alternative Resolver Model	49
7	Future Work	51
7.1	Design	51
7.1.1	Endpoint Identifier Descriptor	51
7.1.2	Host Identifiers	52
7.1.3	Locators	52
7.1.4	Referrals	52
7.1.5	Rendezvous Server	53
7.1.6	Protocol Integration	53
7.1.7	Events	53
7.1.8	Policy API	54
7.1.9	Standardized Interface to the HIP Module	54
7.2	Implementation	54
8	Conclusion	56
A	Application Code Examples	62
A.1	Connection Test Server	62
A.2	Connection Test Client	66
A.3	Connection Test Client with Application Specified Identifiers	69

Abbreviations

AID	Application Identifier
API	Application Programming Interface
DHT	Distributed Hash Table
DNS	Domain Name System
DoS	Denial of Service
EID	Endpoint Identifier Descriptor
FQDN	Fully Qualified Domain Name
FTP	File Transfer Protocol
GID	Group ID
GSS	Generic Security Service
HAA	Host Assigning Authority
HIP	Host Identity Protocol
HI	Host Identity
HIPL	HIP for Linux
HIT	Host Identity Tag
IETF	Internet Engineering Task Force
IKE	Internet Key Exchange
IP	Internet Protocol
IPv4	Internet Protocol version 4
IPv6	Internet Protocol version 6
IPsec	Internet Protocol security

LIN6 Location Independent Networking for IPv6
LSI Local Scope Identifier
PEM Privacy Enhanced Mail
PKI Public Key Infrastructure
POSIX Portable Operating System Interface
QoS Quality of Service
SA Security Association
SCTP Stream Control Transmission Protocol
SLP Service Location Protocol
SP Security Policy
SRV Service Record
SSH Secure Shell
TCP Transport Control Protocol
TLI Transport Layer Identifier
UDP User Datagram Protocol
UI User Interface
UID User ID
UMTS Universal Mobile Telecommunications Service
WLAN Wireless Local Area Network
XTI X/Open Transport Interface

Chapter 1

Introduction

The TCP/IP protocol suite was originally designed for trusted network environments with static structure. The nature of Internet has changed since then, but the protocol suite has remained the same. The Internet has become more insecure and the hosts are attached to the network dynamically.

A host attached to a network can identify itself from the other hosts in the network by its IP address. The IP address is used for routing packets to the host. Unfortunately, the same IP address is also used as a part of the Transport Layer Identifier (TLI). The drawback of this scheme is that the transport layer connections persist only as long as the network layer IP addresses remain the same.

Various proposals have been introduced to extend or redesign the TCP/IP protocol suite to face the new challenges in the Internet. Host Identity Protocol (HIP) is one of these proposals. Host Identity Protocol (HIP) introduces a new cryptographic namespace for Internet hosts. The cryptographic nature of the namespace allows security to be embedded seamlessly into the overall architecture.

One purpose of the new namespace is to decouple the transport layer identifiers from the network layer identifiers. The function of the transport layer identifiers is now to denote location independent connection endpoints. The network layer identifiers, the locators, denote the location of a host instead of an endpoint. They are used for routing purposes. The benefit of the decoupling is persistent transport layer connections between hosts while allowing the network layer locators to change.

A new conceptual layer, HIP layer, is required as a consequence of the separation of the namespaces. The HIP layer handles namespace conversions between the transport layer identifiers and the network layer locators. The HIP layer is located between the transport and network layers, which is a convenient place to handle the namespace conversions and to support persistent transport layer connections. The HIP layer also handles the authentication of the end-hosts and locator binding updates.

The new transport layer identifiers are also present in the application layer. The identifiers can be introduced to the application layer with varying degrees of visibility.

However, the most transparent level of visibility does not allow the application to detect that it is even using HIP. Further, it prevents the applications to control the HIP layer behaviour. The HIP layer can be utilized better by introducing a native HIP Application Programming Interface (API) for HIP aware applications.

1.1 Problem Statement

The main goal of this thesis is to design an API for HIP aware applications. To design the Application Programming Interface (API), requirements have to be surveyed and selected to support the basic functionality in HIP. A reference implementation of the API should be constructed.

1.2 Scope

The scope of this thesis is limited to designing an API for HIP aware network applications. The focus will be on the semantical issues of the API. The proof of concept is provided by the reference implementation. The reference implementation consists of a kernelspace socket handler, userspace resolver library and telnet applications, that will be ported to use the API. The programming language used for the design and implementation is C.

The API design should support the decoupling of transport and network layer namespaces by hiding the details of the network layer from the user of the API. Networking related issues, such as resolving identities, socket binding and network connection management should be handled in the API. The API is based on the sockets API [7] [39].

Quality of Service (QoS) management is out of the scope. Supporting other types of APIs, such as X/Open Transport Interface (XTI), are out of the scope. The API supports only Domain Name System (DNS). Support for other directories is out of scope. Solving the reversal DNS query problem, that is, resolving Host Identity Tags (HITs) to Internet Protocol (IP) addresses, is out of scope.

Chapter 2

Background

In this section, we give brief overviews of some of the background topics. We assume that the reader understands the basic concepts of the TCP/IP suite and has sufficient skill in C programming. The later chapters require detail information about the sockets API, so we will concentrate on it in this section.

2.1 Mobility Related Terminology

An "endpoint" is defined as one participant of an end-end communication; i.e. the fundamental agent of end-end communication. It is the entity which is performing a reliable communication on an end-end basis. [2]. The end-host is a computational unit hosting a number of communicating processes [32].

End-host mobility refers to the phenomenon where the host changes its topological point of attachment while the communication context is kept alive [32]. End-host multihoming is similar to the end-host mobility, but the difference is that the host has multiple points of attachments to the network. The communication context can be moved alive from one attachment point of the host to another.

A handover (handoff) is the process by which an active mobile node changes its point of attachment to the network, or when such a change is attempted. The access network may provide features to minimize the interruption to sessions in progress. Horizontal handover involves mobile node's moving between access points of the same type (in terms of coverage, data rate and mobility), such as, Universal Mobile Telecommunications Service (UMTS) to UMTS, or Wireless Local Area Network (WLAN) to WLAN. Vertical handover involves mobile nodes's moving between access points of different type, such as, UMTS to WLAN. [21]

2.2 Host Identity Protocol

This section is a short overview of the HIP drafts [26, 27, 31, 25, 3, 28]. A number of HIP related publications [32, 30, 13, 1, 49, 17, 37] are also available.

2.2.1 Restrictions in TCP/IP

The current Internet architecture is not very secure. Most of the network traffic is not encrypted, which makes it prone to eavesdropping or even tampering. IP addresses are easy to steal [29]. It does not support end-host mobility and multihoming either, because the TCP/IP suite was originally designed to be used in static network environments. The network layer addresses are reused in the transport layer, because of the static network topology assumptions. The drawback of this design choice is that the transport layer connections are still bound to the old IP addresses if the network layer addresses have changed. This causes the transport layer connections to break. HIP addresses these shortcomings in the Internet architecture by introducing a separate address space to the upper layers.

2.2.2 A New Namespace

HIP introduces a new namespace for the Internet. The namespace is disjoint from the IPv4 and IPv6 namespaces to provide location independent upper layer endpoints. Consequently, the decoupling of the network layer identifiers from the upper layers identifiers provides a sound foundation to build mobility and multihoming. The upper layers have stable endpoint identifiers, but the network layer addresses are allowed to change.

A Host Identity (HI) represents the endpoint for the upper layers in the HIP architecture. The HI is the public key from an asymmetric key pair. The cryptographical nature of the HI provides the direct means to prove the ownership of the endpoint identifier and makes the theft of the HIP endpoint identifiers very difficult.

The Transport Layer Identifier (TLI) in the current Internet model is named with the source IP address, source port, destination IP address and destination port. HIP changes the transport layer TLI by replacing the IP addresses with HIs. The locators, i.e. the IP addresses, are isolated to the network layer. The binding between a HI and the corresponding locators is dynamic to support mobility and multihoming. The bindings are illustrated in Figure 2.1.

HIP architecture includes also fixed sized representations of the Host Identity (HI). A HIT is an 128 bit hash of the HI. Further, the HIT either consist of the SHA-1 hash of the public key or it is in Host Assigning Authority (HAA) format [27, 28]. The former is just a “flat” sequence of bits whereas the latter includes hierarchical information on the domain of the HI. A Local Scope Identifier (LSI) is a 32 bit representation of the HIT. In addition, two types of HIs are defined, public and anonymous. A host is required to have at least one public and one anonymous HI.

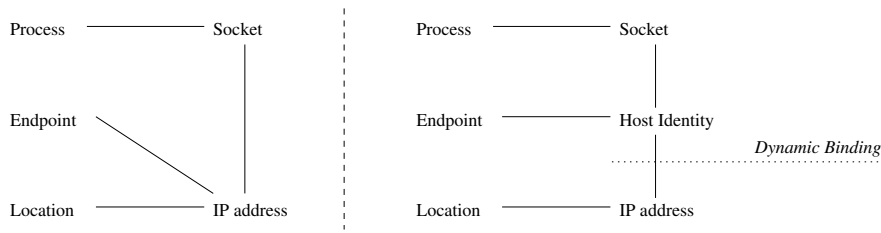


Figure 2.1: The current Internet binding model (left) and the HIP binding model

The HIs are usually stored in the DNS, or distributed using another mechanism, such as Public Key Infrastructure (PKI).

The Fully Qualified Domain Name (FQDN) of a host is stored in the DNS along with the associated locators. The HI of the host is also stored in the DNS in its various forms. The FQDN can be resolved both to the HI and locators. Reverse resolving a locator to an FQDN is also possible, but currently it is impossible to resolve certain types of HIs to a FQDN or a locator. The HIs that do not include any information of the domain of the HI cannot be reverse resolved, because DNS searches are based on hierarchical domain names. The resolving related namespace restrictions are illustrated in Figure 2.2.

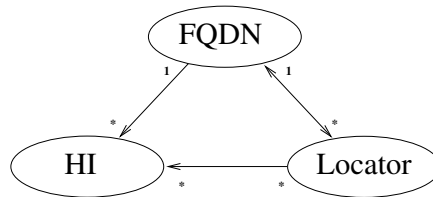


Figure 2.2: Namespace relationships

2.2.3 A New Layer

HIP requires modifications into the networking stack. HIP layer must be inserted between the transport and network layers. The layer is responsible of the various HIP protocol mechanisms, such as *the base exchange*.

The base exchange resembles a light-weight version of the IKE key exchange [11]. The base exchange is an authenticated Diffie-Hellman key exchange consisting of four packets. Two Security Associations (SAs), one in each direction, are constructed from the key material transmitted in the base exchange. The SAs are then used for protecting the network data traffic between hosts using Internet Protocol security (IPsec).

The base exchange also includes a “cookie” mechanism that protects the responder

from Denial of Service (DoS) attacks. The initiator of the connection is forced to sacrifice some CPU cycles in order to find a solution for the cookie sent by the responder. The cookie mechanism allows the responder to delay the allocation of resources for the initiator until the very last moment. This way, it is harder for the initiator to succeed in a DoS attack against the responder. The responder can also vary the difficulty level of the cookies.

The base exchange can also be initiated without a prior knowledge of the HI of the peer. This operation mode is called “opportunistic HIP”. The opportunistic mode is prone to man-in-the-middle-attacks because the initiating host has no prior knowledge of the identity of the responder and a malicious host could substitute the responder. The benefit of the opportunistic mode is that it does not require any infrastructure for storing HIs.

2.2.4 Mobility and Multihoming

The HIP layer implements also the end-host mobility and multihoming support. Existing transport layer connections are preserved by signaling the peer when the locators of the host change. Malicious hosts cannot tamper the integrity of the signaling messages, because they are signed with the public key of the host.

Preservation of existing connections is relatively straightforward in HIP, but it assumes that the initial contact with the peer has been already established. The locators of the peer are usually resolved from the DNS, along with endpoint identifiers. However, the DNS is quite static and it may not keep up with the updates in the locators. The Secure Domain Name System (DNS) Dynamic Update [46] is a better alternative, but even it is hindered by DNS caching.

The rendezvous server is a solution to the problem of the initial contact. The rendezvous server has a set of stable locators. The DNS configuration for a mobile node consists of the endpoint identifiers of the mobile node, but instead of the ephemeral locators of the mobile node, it contains the stable locators of the rendezvous server. Now, when a corresponding node initiates a connection to the mobile node using the information gathered from the DNS, the first HIP signaling message is routed to the rendezvous server instead of the mobile node. The rendezvous server forwards the packet to the current location of the mobile node. Rest of the HIP signaling messages are carried directly between the end-nodes to avoid triangular routing. The rendezvous server solves also the problem of double jump, i.e. both nodes change their location at the same time.

The problem of the initial contact is now solved using the rendezvous server, which knows always the location of the mobile node. When the mobile node changes its location, it informs the rendezvous server of the new locators using secured signaling messages. As the rendezvous server does change its location, both the mobile and corresponding node know how to contact it. The rendezvous server resembles the home agent in the Mobile IP [34] architecture.

2.3 Related APIs

This section overviews the APIs that are significantly related this document. For other related APIs excluded from the discussion, please see e.g. *LIN6 Multihoming API* [22], *Multihoming with Internet Protocol Version 6* [10], Generic Security Service (GSS) API C-bindings [20], *QoSockets: a New Extension to the Sockets API for End-to-End Application QoS Management* [4], `PF_KEY` [23], `NETLINK` [36], Service Location Protocol (SLP) [9] and `OpenSSL` [45].

2.3.1 Sockets API

This section provides a brief introduction to the sockets API [7]. The discussion in this section is based on [39].

Address Structures

The IP addresses are contained in special structures before they are passed to sockets API functions. The IPv4 addresses are encapsulated in `sockaddr_in` structures with the family set to `AF_INET`. Similarly, IPv6 addresses are stored in `sockaddr_in6` structures [6] with the family set to `AF_INET6`. The structures are shown in Figure 2.3.

The port and address fields in the `sockaddr_in` and `sockaddr_in6` structures are stored in network byte order. The other fields are stored in host byte order.

The sockets API [7] provides two abstraction structures for representing any kind of socket address. The first one is the `sockaddr` structure, which is usually passed as a pointer to the sockets API functions¹. The first two fields, the length and the family, are the same in all socket address structures, thus allowing the function to determine the intended length of the structure. The generic socket address structure is shown in Figure 2.4.

The other abstraction structure, `sockaddr_storage`, is defined in [6]. The structure is sufficiently large to represent an address structure of any family. It simplifies the writing of cross-platform and address independent applications.

Socket

The socket [47] is a very central concept in the sockets API. A socket denotes a TLI (half-association), which consists of an IP address and port number. A socket pair denotes a TLI pair (full connection association). It consists of the source IP address, source port, destination IP address and destination port.

¹In ANSI C, a void pointer could be used instead of a `sockaddr` pointer. However, the sockets API predates the ANSI C, and the void pointer is not used [39].

```

/* IPv4 socket address structure for 4.4BSD based systems */
struct sockaddr_in {
    uint8_t      sin_len;      /* length of structure (16) */
    sa_family_t  sin_family;
    in_port_t    sin_port;
    struct in_addr sin_addr;
    char         sin_zero[8]; /* unused */
}

/* IPv6 socket address structure for 4.4BSD based systems */
struct sockaddr_in6 {
    uint8_t      sin6_len;      /* length of this struct */
    sa_family_t  sin6_family; /* AF_INET6 */
    in_port_t    sin6_port;    /* transport layer port # */
    uint32_t     sin6_flowinfo; /* IPv6 flow information */
    struct in6_addr sin6_addr; /* IPv6 address */
    uint32_t     sin6_scope_id; /* set of interfaces for scope */
};

```

Figure 2.3: The socket address structures used for passing IPv4 and IPv6 addresses to the sockets functions. It is worth noting that the length and family fields are combined into a single field (`sin_family` or `sin6_family`) in the 4.3BSD [6].

```

struct sockaddr {
    uint8_t      sa_len;
    sa_family_t  sa_family;
    char         sa_data[14];
};

```

Figure 2.4: The generic socket address structure

The socket acts as the communication point between the application and the networking stack. The application must create a socket before it can establish any network communications. The `socket` function is used for creating a socket. It inputs three arguments for setting the attributes for the socket to be created. The first argument sets the communication domain, such as `PF_INET` for Internet Protocol version 4 (IPv4) or `PF_INET6` for Internet Protocol version 6 (IPv6) enabled socket ². The second argument sets the communication semantics. For example, `SOCK_STREAM` is used for creating a TCP socket and `SOCK_DGRAM` is for creating an UDP socket.

²The sockets API defines also the prefix `AF_`, such as in the `AF_INET` or `AF_INET6` constants. In practice, the `PF_` prefix is an alias for the `AF_` prefix. See [39] for the details.

The third argument is usually zero, but can be set to e.g. `IPPROTO_SCTP` to create an SCTP enabled socket. The prototype of the `socket` function is shown in Figure 2.5.

```
int socket(int domain, int type, int protocol);
```

Figure 2.5: The `socket` function

The `socket` function returns a positive *socket descriptor* value on success. The socket descriptor represents the socket and it will be used in the subsequent sockets API function calls.

Resolver

The *resolver* provides network address translation services to the application. It maps host names to the corresponding IP addresses and vice versa.

The `getaddrinfo` resolver function handles the nodename-to-address translation using the `addrinfo` data structure. The `getaddrinfo` function and the associated data structure are shown in Figure 2.6. The reverse functionality is provided by the `getnameinfo` interface.

```
struct addrinfo
{
    int          ai_flags;      /* Input flags */
    int          ai_family;    /* E.g. PF_INET6, PF_UNSPEC */
    int          ai_socktype;  /* Socket type, e.g. SOCK_STREAM */
    int          ai_protocol;  /* Usually just zero */
    socklen_t   ai_addrlen;    /* Length of socket address */
    struct sockaddr *ai_addr;   /* Socket address for socket */
    char        *ai_canonname; /* Canonical name */
    struct addrinfo *ai_next;  /* Pointer to the next addrinfo */
};

/* nodename-to-address translation */
int getaddrinfo(const char *node, const char *service,
               const struct addrinfo *hints,
               struct addrinfo **res);

void freeaddrinfo(struct addrinfo *res);
```

Figure 2.6: The `getaddrinfo` resolver and the associated data structure

The `getaddrinfo` function can be used both for resolving localhost and peer names.

The first argument, `node`, denotes the name to be resolved. A `NULL` argument is equal to the localhost. The second argument, `service`, describes the name of the service port to be used for the endpoint. The third argument, `hints`, sets the attributes required from the endpoint.

The function outputs the result of the query to the last argument, `res`. The result consists of a linked list of `addrinfo` structures. The `ai_addr` member in the structures is a socket address structure and it can be directly used the socket API calls. The result must be deallocated with the `freeaddrinfo` call when it is not needed anymore. The description of the other resolver function, `getnameinfo`, as well a comprehensive reference to the `getaddrinfo` function, can be found from [7, 39].

Interface Identification

Three network interface related functions are defined in [6]. An interface name, such as "eth0", can be converted with `if_nametoindex` function to the corresponding integer index. The reverse operation can be done with `if_indextoname` function. All interface names and indexes can be queried with `if_nameindex` function. The function returns an array of `if_nameindex` structures as shown in Figure 2.7.

```

struct if_nameindex {
    unsigned int  if_index; /* 1, 2, ... */
    char         *if_name; /* null terminated name: "le0", .. */
}

struct if_nameindex *if_nameindex(void);
void if_freenameindex(struct if_nameindex *ptr);

```

Figure 2.7: The `if_nameindex` function returns a dynamically allocated array of `if_nameindex` structures, which must be deallocated with the `if_freenameindex` function. The end of the array is indicated with an `if_index` of zero and a `NULL` `if_name`.

BSD based systems also include a function called `getifaddrs`. It is very similar to the `if_nametoindex` function.

Basic Usage

In this section, the rest of the basic functions in the sockets API are introduced using an example scenario. In the scenario, a “client” application sends some data to a “server” application using TCP or UDP. The server sends a response to the client. The client receives successfully the data and closes the connection. For simplicity, the sockets operate in blocking mode in the scenario.

The server application creates a socket with the `socket` function. The applications call the `bind` function, which associates the socket with the given source IP address and source port. It should be noted that the application can call the `bind` function only once for the same socket. The socket is “disposable”, because the socket cannot be reused to create another connection association. The `bind` interface is shown in Figure 2.8.

```
int bind(int sockfd, struct sockaddr *my_addr, socklen_t addrlen);
```

Figure 2.8: The `bind` function inputs a socket descriptor, an address of the localhost and the size of the address structure in octets. It returns zero on success, or a negative value on error.

If the server application creates a socket of `SOCK_STREAM` type, it first calls `listen` to indicate its willingness to accept incoming Transport Control Protocol (TCP) connections to the port specified in the `bind` call. Second, the server application calls `accept`, which blocks until the TCP handshake [35] is established. The `accept` call also returns the address of the client. The client initiates the handshake by calling `connect` function, which blocks. When the handshake is successfully completed, both the `connect` and `accept` function return and the applications can continue their execution. The `accept` function returns a new socket descriptor that the server application can use for communication with the client. It is worth noting that another call to the `accept` function would block in server application until another client connected to the server port, and returned a new socket descriptor corresponding to the new connection. The connection oriented function prototypes used for TCP are shown in Figure 2.9.

```
int listen(int s, int backlog);
int accept(int s, struct sockaddr *addr, socklen_t *addrlen);

int connect(int sockfd, const struct sockaddr *serv_addr,
            socklen_t addrlen);
```

Figure 2.9: TCP oriented sockets API functions

The client and server applications now begin to exchange data with each other. Data is sent using `send` function and received using `recv` function. The function prototypes are shown in Figure 2.10.

The connection example using TCP is summarized in Figure 2.11. The server creates a TCP socket, and calls optionally `getaddrinfo` if accepts connections only from a specific server address. Otherwise, it can accept connections from a wildcard IP address by specifying the constants `IN_ADDR_ANY` or `IN6ADDR_ANY_INIT` as the

```

ssize_t send(int s, const void *buf, size_t len, int flags);
ssize_t recv(int s, void *buf, size_t len, int flags);

```

Figure 2.10: Connection oriented functions for sending and receiving data

address. The server calls `bind`, `listen` and `accept`. The `accept` call blocks until the client side is ready. The client creates a socket, resolves the IP address of the `server` from the DNS and calls `connect`. Finally, the TCP handshake has been established and the applications can communicate with each other using the `send` and `recv` functions.

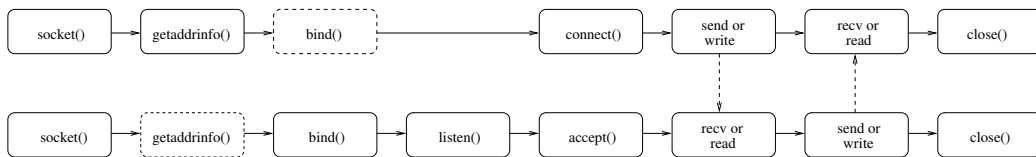


Figure 2.11: Typical client (above) and server (below) application interaction using TCP

User Datagram Protocol (UDP) based socket communication requires fewer functions calls than in TCP, because UDP is not connection oriented and there is no need establish the connection with a set of functions. The `listen`, `accept`, `connect`, `send` and `recv` functions are not necessary in UDP based sockets, but it is possible to use them to emulate the connection oriented socket programming model. The use of the `bind` function is not mandatory, but it usually makes sense to reserve a port especially in server applications.

A different set of functions, `sendto`, `recvfrom`, `sendmsg` and `recvmsg`, are used for UDP based data communication between the applications. The main difference to the connection oriented functions is that the destination IP address must be explicitly given. The UDP based functions are shown in Figure 2.12.

```

ssize_t sendto(int s, const void *buf, size_t len, int flags,
               const struct sockaddr *to, socklen_t tolen);
ssize_t recvfrom(int s, void *buf, size_t len, int flags,
                 struct sockaddr *from, socklen_t *fromlen);

ssize_t sendmsg(int s, const struct msghdr *msg, int flags);
ssize_t recvmsg(int s, struct msghdr *msg, int flags);

```

Figure 2.12: Datagram oriented functions for sending and receiving data

The example scenario using UDP is summarized in Figure 2.13. The client and the server applications create a socket. The server application calls `bind` to reserve a source port on the server host. Now, the client and server applications can communicate directly with each other using `sendto`, `recvfrom`, `sendmsg` and `recvmsg` functions.

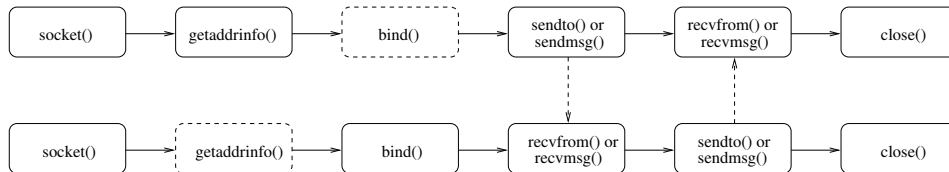


Figure 2.13: Typical client (above) and server (below) application interaction using UDP

It should be noticed, that both in the TCP and UDP examples, the client did not explicitly call `bind` to associate a port and IP address on client host. If the client does not call explicitly `bind`, the networking stack takes care of it automatically by assigning an *ephemeral port* and a wildcard address for the client application. The automatic assignment occurs during the `connect` call in the case of TCP. It is not common for a client to select its own port and IP address because the client applications do not usually care about the source ports on client host.

Socket Options

The socket options can queried and set using the functions shown in Figure 2.14. The first argument for both of the functions is the socket. The second argument is the level of the socket option. For example, the level can be set to `IPPROTO_TCP` or `IPPROTO_IPV6`. The third argument is the option specific value. It is a pointer to e.g. an integer or a data structure. The length of the object referenced by the pointer is indicated by the fourth argument.

```
int getsockopt(int s, int level, int optname, void *optval,
              socklen_t *optlen);
int setsockopt(int s, int level, int optname, const void *optval,
              socklen_t optlen);
```

Figure 2.14: Socket option functions

2.3.2 SCTP Socket API Extensions

Stream Control Transmission Protocol (SCTP) provides a reliable end-to-end message transportation service over IP-based networks [41]. It provides many enhance-

ments over TCP, such as support for multihomed hosts and multiple streams in a single SCTP association [41]. These and many other enhancements make SCTP superior for real-time multimedia and telephony applications than TCP.

SCTP API [40] provides a TCP style interface for enabling SCTP in applications with just a single change in the application code. The only change that is needed, is to set the last argument of the `socket` function, `protocol`, to `IPPROTO_SCTP`. However, such an application cannot utilize the multi-streaming ability of SCTP socket. Also, the TCP style interface is not the best way to use SCTP, because SCTP is message oriented, not byte oriented like TCP.

The application can benefit from the new SCTP features by using the UDP style interface. The UDP style interface can be utilized with the `sendmsg` and `recvmsg` interfaces, which were shown briefly in 2.3.1. The functions provide a scatter/gather array in the `msg_hdr` argument, which the application can use to compose (or receive) a message in a several non-contiguous buffers and yet have them all considered as one message [40].

The UDP based functions can be used to carry ancillary data, such as SCTP stream identifiers, to and from the SCTP socket handler. The SCTP event notifications are also carried in the ancillary data and a separate event notification interface is unnecessary. To receive notifications, the application must first set the socket option for a specific event type. The application then sends or receives data by calling the `sendmsg` and `recvmsg` functions as normally. The events can be read when the function call returns. The difference between normal application data and event data is indicated with a special flag in the `msg_hdr` argument.

2.3.3 Legacy HIP API

The legacy HIP API refers to the initial implementation specific HIP APIs, such as in [43] and especially [1]. The main motivation for the legacy API is to enable HIP transparently in the application by minimizing the number of changes in the application code.

In this section, we will limit to the the legacy API used in [1]. It should be noted that this specific API supports only IPv6.

In the legacy API, the network applications can be ported with no changes in the application code by modifying a component common to all applications, the resolver. The `getaddrinfo` resolver function has a new compile time option that enables *transparent mode* in the resolver. If the mode is enabled, the resolver returns HITs before the IPv6 addresses when it is called to resolve the IP addresses of a peer³. The client application is assumed to try the socket addresses in the order received from the resolver. As the first socket addresses are HITs, HIP connections are preferred

³The legacy API does not return HITs when resolving localhost addresses, because an unmodified `bind` function would normally reject it (unless a HIT is configured for the host with `route` and `ifconfig`) utilities. The local socket addresses must be bound using the `IN6ADDR_ANY_INIT` macro

over plain TCP/IP. If the resolver cannot find any HITs matching to the peer host name, it returns just the IPv6 addresses of the peer. The fallback to the IPv6 addresses is useful during the transition phase to HIP when all network hosts are not capable to support HIP.

The resolver also communicates the peer HIT-to-IPv6 address mapping directly to the networking stack. The IPv6 address cannot be just forgotten, because the HIT cannot be used for routing. Also, it would be very difficult to find the address corresponding to the HIT in the networking stack code.

The transparent mode is not flexible enough, because all applications in the host are forced to use it. Another approach is to use HIP only when an application requires it explicitly. The application can use HIP explicitly by setting a special flag, `AI_HIP`, in the resolver. The resolver returns HITs if the flag is set. In fact, it returns *only* HITs to emphasize that the application *requires* HIP. The resolver sends the mappings from the peer HITs to the IP addresses in the same way as in the transparent mode.

The transparent and explicit mode can be used in parallel. Table 2.1 summarizes the different combinations.

Transparent mode	AI_HIP	Resolver output
off	not set	no HITs
off	set	only HITs
on	set	only HITs
on	not set	HITs before IPv6 addresses

Table 2.1: The output of *getaddrinfo* with different combinations of the transparent mode flag and the `AI_HIP` flag.

Chapter 3

Requirements

In this chapter, we list and rationalize the requirements for the native HIP API design. The requirements are influenced by the design considerations in [6] and [38].

The requirements are organized into functional and non-functional sections. Functional requirements include topics related to the management of identities, locators, interfaces and directory services. Non-functional requirements include topics related to the target user group, security, usability and compatibility. We present also the methods for evaluating the requirements against the actual design choices.

3.1 Non-functional Requirements

Non-functional requirements state the high level goals for the design. They revolve around the topics of usability and compatibility.

3.1.1 Usability

HIP proposes two radical changes to the networking stack design, namely a new cryptographically based identity namespace and a new protocol layer. The changes are reflected in the native HIP API design. Before the changes can be utilized in the API, there are two problems to solve.

First, the management of the HI namespace and bindings between HI and locator namespace cause additional complexity. This may render the API difficult to use. Second, the native HIP API is targeted for UNIX network application developers, especially those who are already familiar with the sockets API [39]. The basic use of the native HIP API should be as simple as, or even simpler than the typical use of the sockets API. The usability of the advanced features of the native HIP API is considered as a secondary goal.

Basically, the native HIP API must reuse the sockets API design as much as possible and extend it where reuse is not possible. This guarantees that the target user group

learns to program the native HIP API quickly and becomes comfortable in using it with little effort.

3.1.2 Compatibility

Compatibility with related standards and APIs should be preserved as much as possible. However, some of the presented compatibility requirements have only secondary value in the scope of this thesis.

Backwards Compatibility

The early experimentation with the legacy HIP API [1] gives some background for the requirements. The legacy HIP API does not require any changes to the network application code in the transport mode, or just one flag in the explicit mode. This can be troublesome in those network applications that are very dependent of the TCP/IP protocol suite. Introducing a new namespace, without applications being aware of it, can result unexpected behavior or render it unusable in the worst case.

The lessons learned from the legacy API must be taken into consideration in the native HIP API design. The use of the native HIP API must be clearly distinguished from the sockets API use. Applications need to be modified to make them HIP aware. This way, the developer receives a discreet hint that enabling HIP may require also other changes in the application.

The legacy API resolver can fall back to IP addresses in the transparent mode if no HIs were found for a host. In such a case, the connection can be established without HIP. The same idea must be reused in the native HIP API.

The native HIP API introduces some changes to the underlying implementation and can break the legacy API support. The implementation changes should be compatible with the legacy HIP API even though the legacy HIP API.

Forward Compatibility

The HIP specifications have not been completely stabilized at the time of writing this thesis. Neither has HIP been evaluated in a larger scale. For example, it may turn out that the size of the HIT is too short. Therefore, the HITs should not be exposed to the typical applications to guarantee compatibility with the future changes in the protocol identifiers.

Other Compatibility Issues

One future research topic is to define a standardized communication interface for HIP related messages between the application and the host. Native HIP API does

not offer such an interface. Instead, the effort is focused on the interfaces visible to the network applications.

Our kernel oriented HIP implementation, HIP for Linux (HIPL) [15], is used for the evaluation of the native HIP API. Investigating of the scalability of the native HIP API with other known (mostly userspace oriented) HIP implementations [14] [24] [12] [18] is considered out of the scope. Portability to other UNIX based operating systems is also considered out of the scope. Compatibility with Portable Operating System Interface (POSIX) [7], conformance to GSS API [19, 20] and supporting Service Location Protocol (SLP) will not be evaluated. Some multi-homing protocols like SCTP [16] and Location Independent Networking for IPv6 (LIN6) [42] define their own userspace APIs [40, 22]. Compatibility with those APIs falls beyond the scope of this thesis.

3.2 Functional Requirements

Functional requirements are more concrete than their non-functional counterparts. They include topics concerning the attributes of the identities and the type of supported directories. Mobility, multihoming and security control requirements are also discussed.

3.2.1 Host Identities

Most network applications do not need to know the representation of the HI. Basically, a reference to the actual HI should be enough in most cases. However, there might be some specialized applications that have to deal with the representation of the identities directly. We have to find a balance between the typical and advanced network application. Also, the requirements for the ownership of the HIs need to be clarified, as well as the operations on HIs.

Representation

The representation of HIs, such as format and size issues, should be as transparent as possible to the applications. We can reach this goal by requiring abstraction and indirection in the native HIP API. The actual sizes and formats of the HIs can be later changed easier if the representation issues are hidden from most of the applications. The transparency may also turn out to be useful in other network protocols based on the identity-locator split but evaluating the usefulness is out of the scope.

Basically, the applications must be able to access the actual representation of the HIs. A complete design would be required to have a standard representation format for the HIs, but that falls out the scope. Both variable sized and fixed sized representations

of HI should be supported in the API. The API must support both anonymous and public HIs.

Application Specified Identities

“Host Identity Protocol” as a name easily gives a wrong impression on the ownership of identities. As the name implies, one could imagine that only the host has some preassigned HIs. The host must have at least one public and one anonymous HI according to the specifications [27], but it is not prohibited to have other sources of identities than just the ones preassigned to the host. The applications should be able to provide their own Host Identities and delegate the rights to use those identities to the host [48].

On the first sight, there is not necessarily any need for application specified identifiers because the identities supplied by the host should suffice for most the purposes. Perhaps a practical, albeit futuristic, use scenario motivates the need for application specific identities. Consider a corporation with an internal network secured from the rest of the Internet with a HIP enabled firewall. The firewall has been configured to accept all network traffic originated from any HI that is owned by an employee of the company. Now, if an employee of the company is telecommuting and needs to access the internal network of the company, he can do it using his private HI. The identity can be conveniently stored in a memory stick and dynamically assigned to the device the employee happens to be using at the moment.

The same HIs can be reused more dynamically on different hosts if the applications are allowed to present HIs to the host. Further on, the access privileges on a specific HI can be more fine-grained and distributed among different entities in the host. For example, the target entities can consist of the applications belonging to a specific user or group. This can save some resources on the host, as the same HI can be shared by multiple processes.

Operations on Host Identities

So far, only the actual identities themselves have been assigned some requirements and almost no focus has been put on how the identities can be utilized in the programming interface. The requirements of the operations that are supported by the interface need to be defined.

At the very least, a one-way communication mechanism to transfer Host Identities from the application to the host is required. Otherwise it is not possible to select between multiple host supplied HIs or input an application specified HIs to the host. Correspondingly, the other way of communication, querying of identities from the host, should also be supported for the benefit of those applications that need to know the details of the identities.

Some auxiliary interfaces should be also introduced for the convenience of the de-

veloper. Interfaces for creating an application specified HI, and loading and saving a HI to a file also have to be defined.

3.2.2 Addresses and Interfaces

The interfaces and especially the addresses, are considered ephemeral from the HIP viewpoint. It is hazardous to handle them explicitly in the API as they are prone to change. The details of the network layer should not be exposed to a typical application. On the other hand, access to the network layer should not be completely omitted for applications that need to access the network layer details. Thus one goal is to define a separate interface for accessing the network layer details.

A typical application does not care about the details of the network layer and trusts the HIP implementation to make the networking layer related choices on the behalf of it. The HIP implementation selects transparently the interfaces and addresses to be used for a connection.

The requirements for explicit locator selection are mostly related to the manual selection of the addresses and interfaces. First, the API should allow entering initial peer addresses manually because the network environment may not have a directory service. Second, it should be possible to select the localhost interface to be used for the network communication. On the other hand, the number of manually selected addresses and interfaces should not be restricted. Limiting the scope of addresses within the selected interfaces should also be possible by expressing the address family.

Basically, it is insignificant to a HIP enabled application whether an address belongs to a HIP rendezvous server or to an actual HIP endpoint. However, the difference should be explicitly shown in the native HIP API. Advanced network applications may need to differentiate between the two address types, e.g. for diagnostic purposes.

The API should support HIP in opportunistic mode, i.e. without a prior knowledge of the peer's HIs. In this case, the API should make it possible for the host to trigger the base exchange by just relying on the locator information of the peer.

3.2.3 Mobility, Multihoming and Policies

In this thesis, we assume that the host dynamically updates the locators transparently from the application. However, the application should configure the initial locators manually. After the peers have established a HIP association, the handoffs are not reflected anymore on the native HIP API. The API could contain an interface for tracking locator updates but designing one is out of the scope.

QoS, load balancing and other similar complex policy issues are also out of the scope. On the other hand, the high level design should still be simple and modular enough so those features can later be added into the API without completely scratching it.

3.2.4 Security

Advanced HIP aware applications utilize the new features of the HIP layer in the networking stack. The security attributes of HIP are an example of the new features. The native API should allow modifying of some of the HIP related security attributes to suit specific application needs. The application should be allowed to negotiate HIP related security attributes, such as the encryption level and algorithm being used for network connections and the selection of the challenge size at the responder. Enabling of the opportunistic mode and falling back to plain TCP/IP if the peer does not support HIP should also be supported.

As the applications are given more control over the HIP related attributes in the networking stack, some security constraints must be introduced to avoid introducing security flaws into the design:

1. Confidentiality: what the process or object is not allowed to see.
2. Integrity: what the process and other processes are not allowed to change.
3. Capability: the possibility for communication.

The private keys of the host provide us an example of the confidentiality constraint: applications running on normal user privileges are not allowed to see the private keys of the host. An example of the integrity constraint is that processes should not be allowed to modify the cookie difficulty of the other processes. An application that tries to select too heavy Diffie-Hellman group to be used in the base exchange exhausts CPU resources of the host serves An example of the capability is the application that is denied to set the Diffie-Hellman group to a substantially large value, because it would exhaust the CPU resources of the host.

3.2.5 Error Handling

The native HIP API should be strongly based on the sockets API. As a consequence, there is no need to introduce a new error management interface. Couple of HIP specific error values could be needed in a few case.

3.2.6 Directories

The native HIP API queries the peer identities and locators from a directory. The native HIP API design is required to support only DNS and the UNIX `/etc/hosts` file. Supporting other kind of directories, such as those based on Distributed Hash Tables (DHTs), are out of the scope. Supporting DNS Service Record (SRV) [8] is also out of the scope.

3.3 Evaluation

The main objective of the implementation is to prove that design can be made to work in practice. The performance, reliability and security of the implementation are not part of the evaluation criteria.

The native HIP API is evaluated by matching and comparing the requirements presented this chapter against the design outcome. Chapter 5 provides a discussion of the proof-of-concept implementation and the ported example applications. The results of the evaluation will be discussed and analyzed in chapter 6.

Chapter 4

Design

The architectural discussion in this chapter describes the semantics of the native API in a semi-formal way. We continue with the syntax of the API after the architectural discussion.

4.1 Architecture

In this section, the native HIP API design is described from an architectural viewpoint. We introduce the Endpoint Identifier Descriptor (EID) concept, which is a central idea in the API. The layering and namespace models are described along with the socket bindings. We conclude the discussion with a description of the endpoint resolving mechanism.

4.1.1 Endpoint Identifier Descriptor

The representation of endpoints is hidden from the applications as required in 3.2.1. The Endpoint Identifier Descriptor (EID) is a “handle” to a HI. The EID serves as a pointer to the corresponding HI entry in the HI database of the host. The EID is the Application Identifier (AID) [33] in the native HIP API model.

4.1.2 Layering Model

The application layer accesses the transport layer via the socket layer. The application layer uses the traditional TCP/IP IPv4 or IPv6 interface, or the new native HIP API interface provided by the socket layer. The layering model is illustrated in Figure 4.1. The IPsec layer has been excluded in the figure for simplicity.

The HIP layer is as a shim/wedge layer between the transport and network layers. The datagrams delivered between the transport and network layers are intercepted in the HIP layer to check if the datagram endpoints are HIs and require HIP

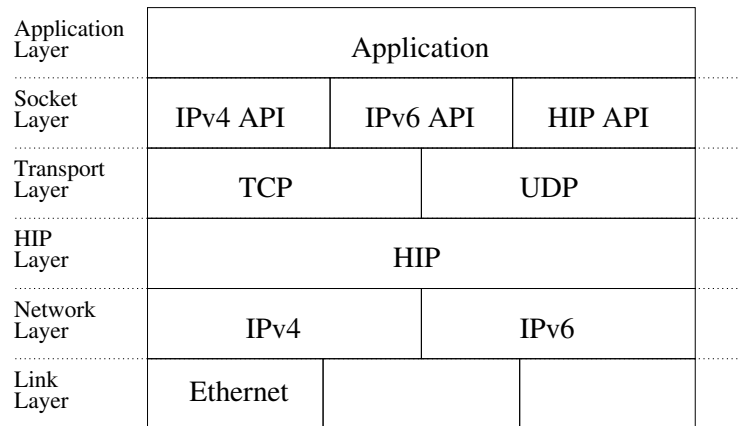


Figure 4.1: The layering model

intervention.

4.1.3 Namespace Model

The namespace model is shown in Figure 4.2. The identifiers of the namespaces are described in this section.

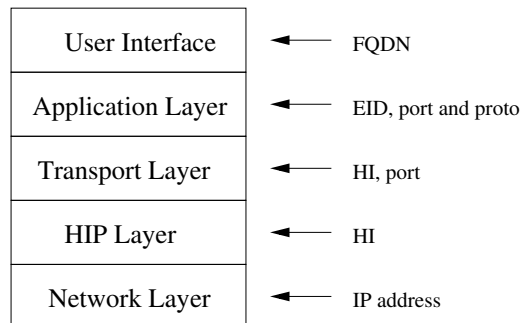


Figure 4.2: The EID centric namespace model

People prefer human-readable names when referring to network entities. Usually, the most commonly used identifier in the User Interface (UI) is the FQDN, but there are also other ways to name network entities. The FQDN format is still the preferred UI identifier in the context of the native HIP API.

Connection associations in the application layer are uniquely distinguished by the source IP address, destination IP address, source port, destination port and protocol. HIP changes this model by using HITs in the place of IP addresses. The HIP model is further expanded in the native HIP API model by using Endpoint Identifier Descriptors (EIDs) instead of HITs. Now, the application layer uses source

EID, destination EID, source port, destination port and transport protocol type to distinguish between the different connection associations ¹.

Basically, the difference to between the application and transport layer identifiers is that the transport layer uses HIs instead of EIDs. The Transport Layer Identifier (TLI) is named with source HI, destination HI, source port and destination port at the transport layer.

Correspondingly, the HIP layer uses HIs as identifiers. The HIP security associations are based on source HI and destination HI pairs rather than process pairs.

The network layer uses IP addresses, i.e. locators, for routing purposes. The network layer interacts with the HIP layer to exchange information of the changes in the localhost interfaces and peer addresses.

4.1.4 Socket Bindings

A HIP socket is associated to one source and one destination EID, along with their port numbers. The relationship between a socket and EID is many to one. The source EID is associated to multiple source HIs and the destination EID is associated to multiple destination HIs. Further, the source HI is associated to a set of network interface of the localhost. The destination HI in turn, is associated to a set of destination addresses of the peer. The socket bindings are visualized in Figure 4.3.

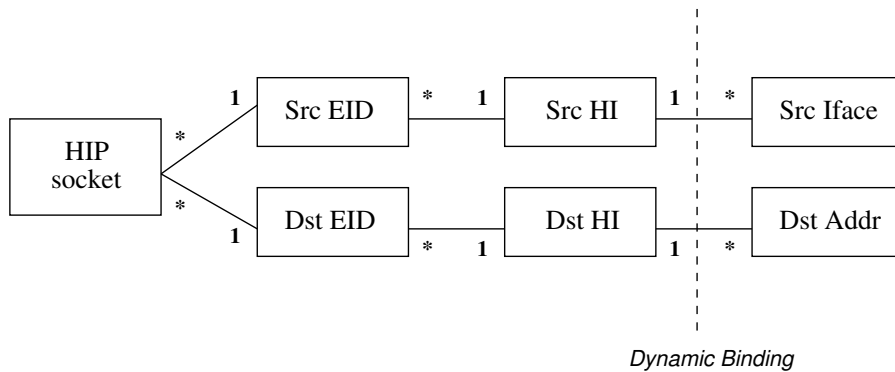


Figure 4.3: Socket bindings

The relationship between a source EID and a source HI is always many-to-one. However, there are two refinements to the relationship. First, a listening socket is allowed to accept connections from all source HIs of host. Second, the opportunistic mode allows the base exchange to be initiated to an unknown destination HI. In a way, the relationship between the source EID and source HI is many-to-undefined for a moment in both of the cases, but once the connection is established, the EID will be permanently associated to a certain HI.

¹See section 6.2.4 for restrictions in the binding to EIDs

The EID concept can only be used in HIP protocol family sockets. Other types of sockets are left intact to avoid breaking the backwards compatibility requirements of 3.1.2.

4.1.5 Endpoint Discovery

Endpoint discovery mechanism from DNS is illustrated in Figure 4.4. The application calls the resolver (step a) to query an FQDN from DNS (step b). The DNS server responds with a HI and a set of IP addresses (step c). The resolver does not directly pass the HI and the locators to the application, but sends them to the HIP module instead (step d). Finally, the resolver receives an EID from the HIP module (step e) and passes the EID to the application (step f).

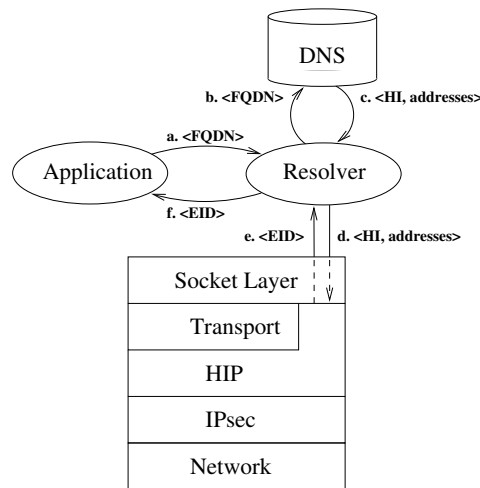


Figure 4.4: The path of resolving of an FQDN to an EID

The application can also receive multiple EIDs from the resolver if the FQDN is associated to multiple HIs. The endpoint discovery mechanism is still almost the same. The difference is that the DNS returns a set of HIs along with their locators to the resolver. The resolver sends all of them to the HIP module and receives a set of EIDs in return, each EID corresponding to a single HI. Finally, the EIDs are received by the application from the resolver.

4.2 Interface Syntax and Description

In this section, we describe the native HIP API using the syntax of C programming language and present only the “external” interfaces and data structures that are visible to the applications. We limit the description to those interfaces and data structures, that are either modified or completely new, because the native HIP API

is otherwise identical to the sockets API [7, 39].

4.2.1 Data Structures

We introduce a new protocol family, `PF_HIP`, for the sockets API. The `AF_HIP` constant is an alias for it. The use of the `PF_HIP` constant is mandatory with the `socket` function if the native HIP API is to be used in the application. The `PF_HIP` constant is given as the first argument (`domain`) to the `socket` function.

The EID abstraction is realized in `sockaddr_eid` structure, which is shown in figure Figure 4.5. The family of the socket, `eid_family`, is set to `PF_HIP`. The port number `eid_port` is two octets and the EID value `eid_val` is four octets. The EID value is just a opaque number to the application. The application should not try to associate it directly to a HI or even compare it to other EID values, because there are separate functions for those purposes. The family of the EID is stored in host byte order. The port and the EID value are stored in network byte order.

```

struct sockaddr_eid {
    unsigned short int eid_family;
    in_port_t eid_port;
    sa_eid_t eid_val;
}

```

Figure 4.5: The EID is contained in a `sockaddr_eid` structure. The structure is shown in 4.3BSD format [6].

The `eid_val` field is usually set by special native HIP API functions, which are described in the following section. However, three special macros can be directly set into the `eid_val` field. The macros are `HIP_HI_ANY`, `HIP_HI_ANY_PUB` and `HIP_HI_ANY_ANON`. They denote an EID value associated to a wildcard HI of any, public or anonymous type. This is useful to a “server” application that is willing to accept connections from all of the HIs of the host. The macros correspond to the sockets API macros `INADDR_ANY` and `IN6ADDR_ANY_INIT`, but they are applicable on the HIP layer. It should be noted that only one process at a time can bind with the `HIP_HI_*ANY` macro on a certain port to avoid ambiguous bindings.

The native HIP API has a new resolver function which is used for querying both endpoint identifiers and locators. The resolver introduces a new data structure, which is used both as the input and output argument for the resolver. The structure `endpointinfo` is shown in Figure 4.6.

The members of the `endpointinfo` structure are similar to `addrinfo` structure, but the member names have a different prefix. The socket address structure used for sockets API calls has been renamed to `ei_endpoint` to emphasize the difference to the `getaddrinfo` resolver. The family, `ei_family`, is set to `PF_HIP` when the socket

```

struct endpointinfo {
    int ei_flags;           /* flags, e.g. EI_FALLBACK */
    int ei_family;         /* e.g. PF_HIP */
    int ei_socktype;       /* e.g. SOCK_STREAM */
    int ei_protocol;       /* usually just zero */
    size_t ei_endpoint_len; /* length of the endpoint */
    struct sockaddr *ei_endpoint; /* endpoint socket address */
    char *ei_canonname;     /* canonical name of the host */
    struct endpointinfo *ei_next; /* next endpoint */
};

```

Figure 4.6: The resolver data structure

address structure contains an EID that refers to a HI.

The flags in the `endpointinfo` structure, control the behavior of the resolver and describe the attributes of the endpoints and locators. The `EI_ANON` flag forces the resolver to query only local anonymous identifiers. The default action is first to resolve the public endpoints and then the anonymous endpoints. If the application wants to configure locators manually, the `EI_NOLOCATORS` flag forces the resolver to discard the resolving of locators. The `EI_FALLBACK` flag hints the resolver to output the locators if no HIs are found. The `ei_endpoint` members in the resolver output are then filled with IPv4 or IPv6 addresses and the application can resort to plain TCP/IP connections using the IP addresses returned. The fallback flag must be explicitly enabled in the flags, because the resolver returns only HIs by default. The `EI_HI_ANY`, `EI_HI_ANY_PUB` and `EI_HI_ANY_ANON` flags causes the resolver to output only one socket address. It contains the EID that would be received using the corresponding `HIP_HI_*ANY` macro.

Application specified identities require the endpoint structures shown in Figure 4.7. The structure `endpoint` represents a generic endpoint and the `endpoint_hip` is the HIP specific endpoint. The HIP endpoint is public by default unless `HIP_ENDPOINT_FLAG_ANON` flag is set in the structure to anonymize the endpoint. The `id` union contains the HI in the `host_id` member in the format specified in the HIP draft [27]. The draft does not specify the format for the private key, so private key material is just appended to the `host_id` and the length is adjusted accordingly. The flag `HIP_ENDPOINT_FLAG_PRIVATE` is also set. The `hit` member of the union is used only when the `HIP_ENDPOINT_FLAG_HIT` flag is set.

An optional extension to the `getaddrinfo` interface is introduced too. A new flag, `AI_RENDEZVOUS`, is set in the `getaddrinfo` resolver output, if the resolved address belongs to a rendezvous server.

```
struct endpoint {
    se_family_t  family;
    se_length_t  length;
};

struct endpoint_hip {
    se_family_t family;
    se_length_t length;
    se_hip_flags_t flags;
    union {
        struct hip_host_id host_id;
        hit_t hit;
    } id;
};
```

Figure 4.7: The endpoint data structures

4.2.2 Functions

The new functions introduced to the sockets API are described in this section.

Resolver Interface

The native HIP API does not introduce changes to the interface syntax of the fundamental sockets API functions `bind`, `connect`, `send`, `sendto`, `sendmsg`, `recv`, `recvfrom` and `recvmsg`. The application usually inputs the functions with `sockaddr_eid` structures instead of `sockaddr_in` or `sockaddr_in6` structures. The source of the `sockaddr_eid` structures in the native HIP API is the resolver function `getendpointinfo` which is shown in figure 4.8.

```
int getendpointinfo(const char *nodename,
                   const char *servname,
                   const struct endpointinfo *hints,
                   struct endpointinfo **res)

void free_endpointinfo(struct endpointinfo *res)
```

Figure 4.8: The endpoint resolver function prototype

The `getendpointinfo` function inputs the `nodename`, `servname` and `hints` arguments. It places the result of the query into the `res` 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 localhost. The `servname` parameter sets the port number to be set in the socket addresses in the `res` output argument. Both the `nodename` and `servname` cannot be `NULL`.

The output argument `res` is dynamically allocated by the resolver. The application must free it with the `free_endpointinfo` function. It contains a linked list of the resolved endpoints. The input argument `hints` acts like a filter that defines the attributes required from the resolved endpoints. For example, setting the flag `HIP_ENDPOINT_FLAG_ANON` in the hints forces the resolver to return only anonymous endpoints in the output argument `res`. If the `hints` argument is `NULL`, any kind of endpoints are acceptable.

Application Specified Identities

Application specified local and peer endpoints be retrieved from files using the function shown in Figure 4.9. The function `load_hip_endpoint_pem` is used for retrieving a private or public key from the given file `filename`. The file must be in PEM encoded format [45]. The result is allocated dynamically and stored into the `endpoint` argument. The return value of the function is zero on success, or a non-zero error value on failure. The result is deallocated with the `free` system call.

```
int load_hip_endpoint_pem(const char *filename,
                        struct endpoint **endpoint)
```

Figure 4.9: The interface for retrieving an application specified identifier from a file

The endpoint structure cannot be used in the sockets API function calls. The application must exchange the endpoint into an EID with the localhost. Local endpoints are exchanged with the `setmyeid` function and peer endpoints with `setpeereid` function. The functions are illustrated in Figure 4.10. Both of functions are used in a similar way.

The result of the exchange is stored into the first argument, `my_eid` or `peer_eid`. The return value is zero on success, or a non-zero error value on failure.

The second argument, `servname`, is the service string. The function converts it to a numeric port number and fills the port number into the first argument for the convenience of the application.

The third argument is the endpoint retrieved with the `|load_hip_endpoint_pem|`. If the `endpoint` is `NULL`, an arbitrary HI of the host is selected and associated with the EID value of the third argument.

The fourth argument defines the initial locators of the localhost or peer host. The `setmyeid` function inputs the localhost locators as a pointer to an “interface index list”. The list can be obtained with the `if_nameindex` [6] function call. A `NULL` list pointer indicates all interfaces of the localhost. The `setpeereid` function inputs

```

int setmyeid(struct sockaddr_eid *my_eid,
            const char *servname,
            const struct endpoint *endpoint,
            const struct if_nameindex *ifaces,
            int flags)
int setpeereid(struct sockaddr_eid *peer_eid,
              const char *servname,
              const struct endpoint *endpoint,
              const struct addrinfo *addrinfo,
              int flags)

```

Figure 4.10: The functions for exchanging application defined endpoints to EID structures.

a pointer to a linked list of `addrinfo` structures containing the initial addresses of the peer. The list pointer can be obtained with a `getaddrinfo` [6] function call. A NULL list pointer indicates that the application trusts the host to already know the locators of the peer. We recommend that a NULL locator list pointer is not given to the `setpeereid` function to ensure reachability with the peer.

The last argument is the flags. The following flags are valid only for `setmyeid`. Flags `HIP_EID_*ANY` correspond to the use of the `HIP_HI_*ANY` macros. Flags `HIP_HI_REUSE_UID`, `HIP_HI_REUSE_GID` and `HIP_HI_REUSE_ANY` allow the HI to be reused for processes with the same User ID (UID), Group ID (GID) or any UID as the calling process. Flags `HIP_EID_IPV6` and `HIP_EID_IPV6` are used for limiting the address family scope of the interfaces.

It should be noticed that the `HIP_HI_ANY`, `HIP_HI_ANY_PUB` and `HIP_HI_ANY_PUB` macros are defined as calls to the `setmyeid` call with NULL endpoint, NULL interface arguments and the flag corresponding to the macro name set.

The functions `getmyeidinfo` and `getpeereidinfo` are shown in Figure 4.11. They are used for querying the HI and the locators associated to a given EID. The first argument is the EID. The second argument is the endpoint structure associated to the EID. The third argument is the list of locators associated with the HI. For `getmyeidinfo`, the locator list is a list of interfaces, and for `getpeereidinfo`, the locator list is a list of addresses.

The `getmyeidinfo` and `getpeereidinfo` functions are especially useful for an advanced application that receives multiple EIDs from the resolver. The advanced application can query the properties of the EIDs using `getmyeidinfo` and `getpeereidinfo` functions and select the EID that matches to the desired properties.

```

int getmyeidinfo(const struct sockaddr_eid *my_eid,
                struct endpoint **endpoint,
                struct if_nameindex **ifaces)
int getpeereidinfo(const struct sockaddr_eid *peer_eid,
                  struct endpoint **endpoint,
                  struct addrinfo **addrinfo)

```

Figure 4.11: The functions for querying EID information

Socket option	Purpose
SO_HIP_CHALLENGE_SIZE	Cookie challenge size
SO_HIP_HIP_TRANSFORM	The preferred HIP transform
SO_HIP_ESP_TRANSFORM	The preferred ESP transform
SO_HIP_DH_GROUP_ID	The Diffie-Hellman group ID
SO_HIP_SA_LIFETIME	Socket association lifetime in seconds
SO_HIP_RETRANS_INIT_TIMEOUT	HIP initial retransmission timeout
SO_HIP_RETRANS_INTERVAL	HIP retransmission interval in seconds
SO_HIP_RETRANS_ATTEMPTS	Number of retransmission attempts
SO_HIP_AF_FAMILY	The preferred address family. The default family is AF_ANY.
SO_HIP_PIGGYPACK	If set to one, HIP piggy-packing is preferred. Zero if piggy-packing must not be used.
SO_HIP_OPPORTUNISTIC	Try HIP in opportunistic mode if only the locators of the peer are known
SO_HIP_OPP_FALLBACK	The same as above, but fall back to plain TCP/IP if base exchange failed
SO_HIP_BEX_FALLBACK	Normal base exchange, but fall back to plain TCP/IP if the base exchange fails.

Table 4.1: HIP socket options

Socket Options

Reading and writing of HIP socket options is done using `getsockopt` and `setsockopt` functions. The first argument must be a socket descriptor that was instantiated with PF_HIP as the domain. The second argument must be specified as IPPROTO_HIP.

Some HIP socket option values are listed in Table 4.1. The length of the option must be two octets. The purpose of the option value must be interpreted in context of the protocol specifications [27, 31].

The socket options must be set before the hosts have established HIP Security Associations (SAs). The implementation may refuse to set the socket options if there is already an existing SA.

Chapter 5

Implementation

In this chapter, the implementation of the native HIP API is described. The emphasis is on the kernel components, because the userspace components are quite trivial. The HIP protocol module implementation is also briefly introduced to give the reader a more complete view of the overall implementation architecture and the interaction between the components.

5.1 Userspace Components

The userspace component of the native HIP API, the resolver library, is linked either statically or dynamically to applications. The implementation of the resolver is based on the *libinet6* library [44].

The userspace library has two functions. First, it maps HIs and locators to EIDs transparently from the typical applications. Second, it provides the functions that are needed for explicit handling of HIs and locators in the advanced applications.

The `getendpointinfo` resolver uses internally the `endpoint_hip` data structure that was presented in subsection 4.2.1. The data structure contains an union of a HI and HIT. In the resolver implementation, the HIT member is used when resolving the peer identifiers. The HI member is used when resolving the HIs of the localhost.

5.2 Kernel-space Components

The kernel-space components were originally implemented on the Linux 2.4 series kernel, but they were also ported with minimal effort to the 2.6 series. The discussion in this section is based on the 2.6 series implementation. It should also be noted that the implementation supports only IPv6.

The kernel-space component, the kernel module, is divided into two conceptual sub-components. The *HIP module* is the protocol implementation. It handles the base

exchange, update and other HIP protocol mechanisms. *HIP socket handler* refers to the HIP socket layer implementation. In practice, the HIP socket handler is a part of the HIP module implementation, but it is conceptually separated in the discussion to keep the focus on the native HIP API.

The socket handler is registered as a PF_HIP family socket handler into the networking stack. The `bind`, `connect`, `send` and other sockets API function calls arrive at the HIP socket handler if they are associated to the PF_HIP socket family. The socket handler is a “wrapper” to the IPv6 socket handler because its main purpose is to translate EIDs to HITs.

The rest of the Linux networking stack is not easy to separate into subcomponents, because there are no clear boundaries in the code. The *transport/network module* refers to the TCPv6, IPv6 packet handling and routing code. *IPsec module* contains the ESP handling functions in the kernel.

5.3 HIP Networking Stack Hooks

Figure 5.1 illustrates the interfaces of the networking stack. The interfaces for output packet flow are illustrated on the left side and, respectively, for the input packet flow on the right side. The networking stack code has is equipped with “hooks” both in the output and input side. The hooks bypass normal networking stack control flow to the HIP module for manipulation when the endpoints are HIs.

In the output interfaces, the application communicates with the HIP socket handler using EIDs. The socket handler uses the HITs to communicate with the transport layer.

The layer below transport layer involves both routing and IPsec handling. At this layer, the hooks are responsible of the conversion of the source and destination HITs to IPv6 addresses (`hip_handle_output`, `hip_get_saddr` and `hip_get_addr`). The base exchange is triggered by `hip_trigger_bex` hook. A global Security Policy (SP) forces all packets with a HIT as the destination addresses to be encapsulated into an ESP envelope.

The input flow is similar to the output flow, but fewer hooks are required. The `hip_handle_esp` replaces the IPv6 addresses with HITs before IPsec processing. The `hip_unknown_spi` function sends an R1 as a response to an unknown SPI.

Initially, it could seem possible to convert the EIDs directly to the IPv6 addresses and vice versa without the intermediate HIT form. However, that is not possible for at least three reasons. The first reason is that the HIP SAs are bound to HITs in the IPsec layer. Second, the transport layer needs the HITs for the pseudoheader checksum calculation. Third, we want to reuse the same hooks both for the legacy and native HIP API. If the socket handler were to map the EIDs straight to IPv6 addresses, we would need separate hooks for the native HIP API.

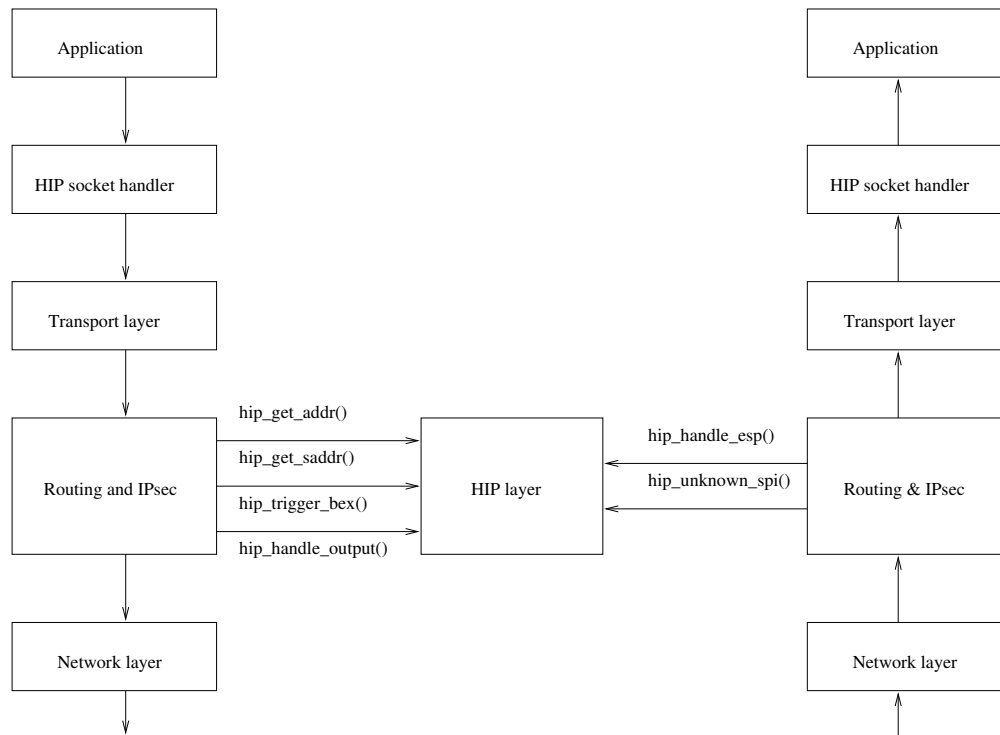


Figure 5.1: The output and input interfaces of the networking stack

5.4 Data Structures

The HIP module has four data structures that are related to the HIP socket handler. Two of them are used for storing local and peer identities, and one is for storing HIP state information related data. The fourth structure is the BSD socket structure, which is commonly used by the kernel socket handlers for various families.

The HIP socket handler has two data structures that are used for storing local and peer EID values. The EID value is associated with the ownership information of the EID entry. It guarantees the confidentiality and integrity of the EID related information. The ownership information consist of the UID and GID of the owner process. The data structures are illustrated in Figure 5.2.

The HIP module can be used without the HIP socket handler, i.e., using the legacy HIP API. Backwards compatibility with the legacy HIP API has a strong influence on the data structure organization in the kernel. The legacy HIP API does not use the HIP socket handler in any form. It is the responsibility of the HIP module to handle the HIP legacy applications and the HIP module must therefore have access to the most important data structures in the kernel. The HIP module can access the host association data structure as well as the local and peer host id data structures, which is the minimum requirement to establish HIP connections. The

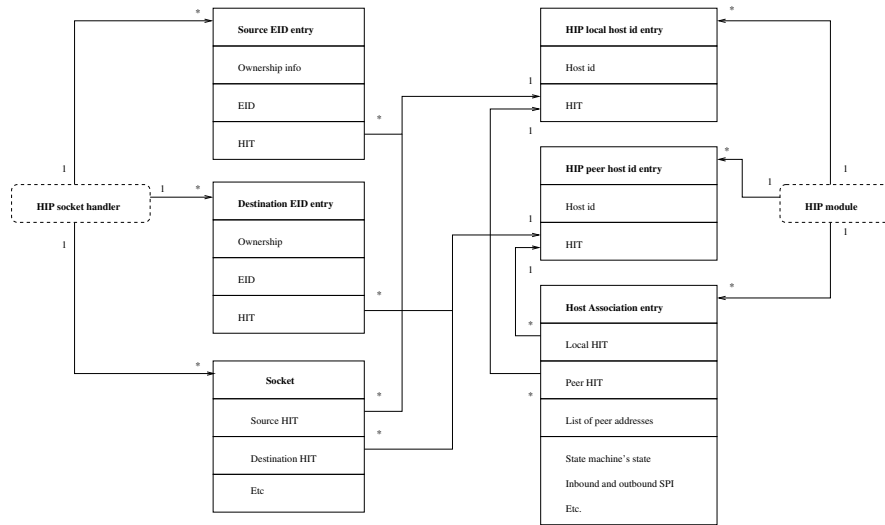


Figure 5.2: HIP kernel module data structures and their relationships

socket handler EID data structures were built on top of the other data structures to ensure the backwards compatibility with the legacy HIP API. The host identifier and EID data structures could have been merged into one structure without the legacy API compatibility restriction.

5.5 Collaboration of Components

The interaction between the components is illustrated using sequence diagrams. The sequence diagrams show the controls flow through different function calls from the userspace to the kernelspace. We do not show the full execution trace but instead focus on the most relevant functions. The reader should not become confused with naming of the functions, because the IPv6 module reuses some of the IPv4 functions (e.g. `inet_create`). However, some functions are IPv6 specific (e.g. `inet6_bind`).

The diagrams are based on a use scenario, where we have two simple network applications with host specified identities. The server application binds to a port on the server host, listens for connections and accepts the connections. The client application on the client host calls the resolver to get the EID of the server host. The client application then connects to the server port and sends some data to the server, which is successfully received by the server application. The applications use TCP for data transmission in this scenario and the sockets are assumed to block for simplicity.

5.5.1 Connection Setup on the Server

The socket initialization of the server application is visualized in Figure 5.3. The server application creates a socket with the `socket` call, which eventually calls `hip_create` function in the HIP socket handler. The socket handler wraps the call to the transport/network module, which calls `inet_create` to create a `socket` structure. The control returns to the server application and it calls `getendpointinfo` to return a local EID. The resolver queries the identifier and network interfaces of the localhost and inputs them to the HIP module using the `ioctl` interface. The HIP module generates a local EID, stores it to the local EID data structure and returns the EID to the resolver. The resolver returns the EID to the application.

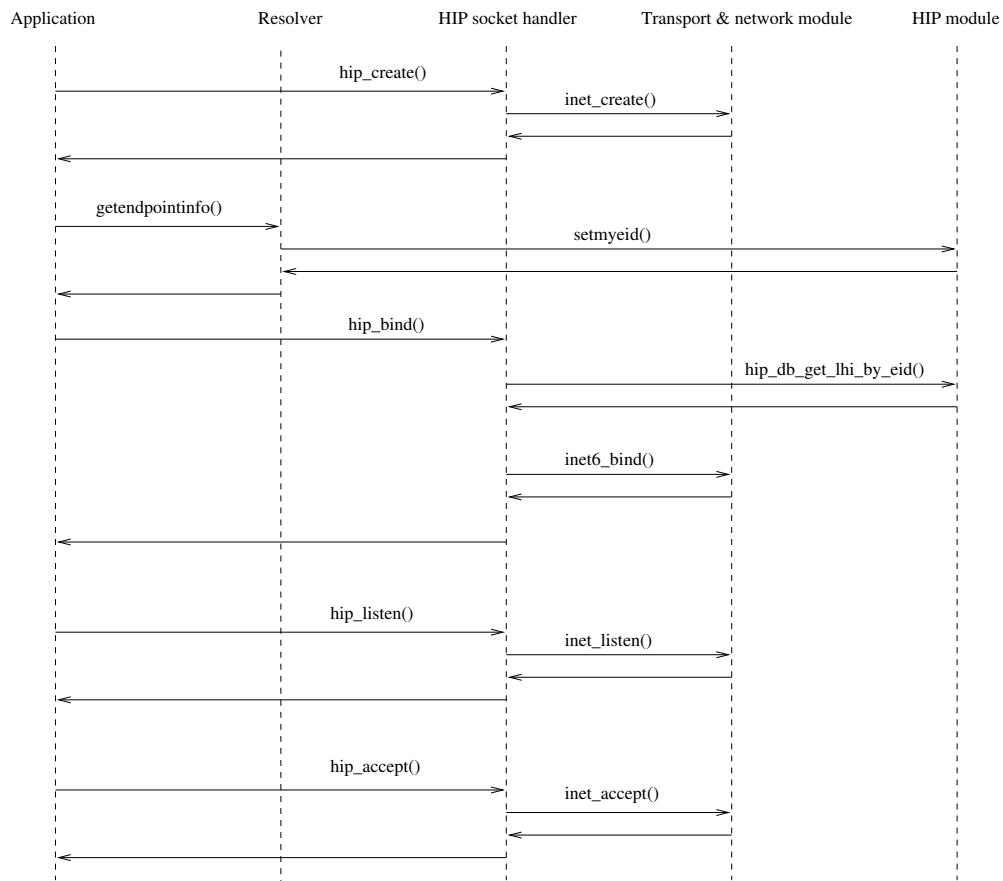


Figure 5.3: Bind sequence diagram

The server application is now ready to bind to the socket using the EID and port number. The `bind` call eventually translates to a `hip_bind` call in the HIP socket handler. The socket handler maps the EID to a local HIT using `hip_db_get_lhi_by_eid` and stores the HIT into the `socket` structure. It binds to the HIT by calling `inet6_bind` in the transport/network module. The bind call returns and the application is re-

sumed.

The server application is now almost ready to receive data from the socket. The server application calls `listen`, which calls `hip_listen` and `inet_listen` in a row. The application resumes its control and calls `accept` to receive a new socket descriptor, which the server application needs in order to communicate with the client. The `accept` call translates first to a `hip_accept` and then to a `inet_accept` call. The `accept` call blocks until the client connects to the server.

5.5.2 Connection Setup on the Client

The connection setup on the client is depicted in Figure 5.4. Initially, the connection setup is very similar to the server. A `socket` structure is created with `hip_create_socket`. The application calls `getendpointinfo` to resolve the peer endpoint. The call also sends the peer HI and locators to the HIP module. The HIP module generates an EID for the peer, stores it into the peer EID data structure and returns the EID.

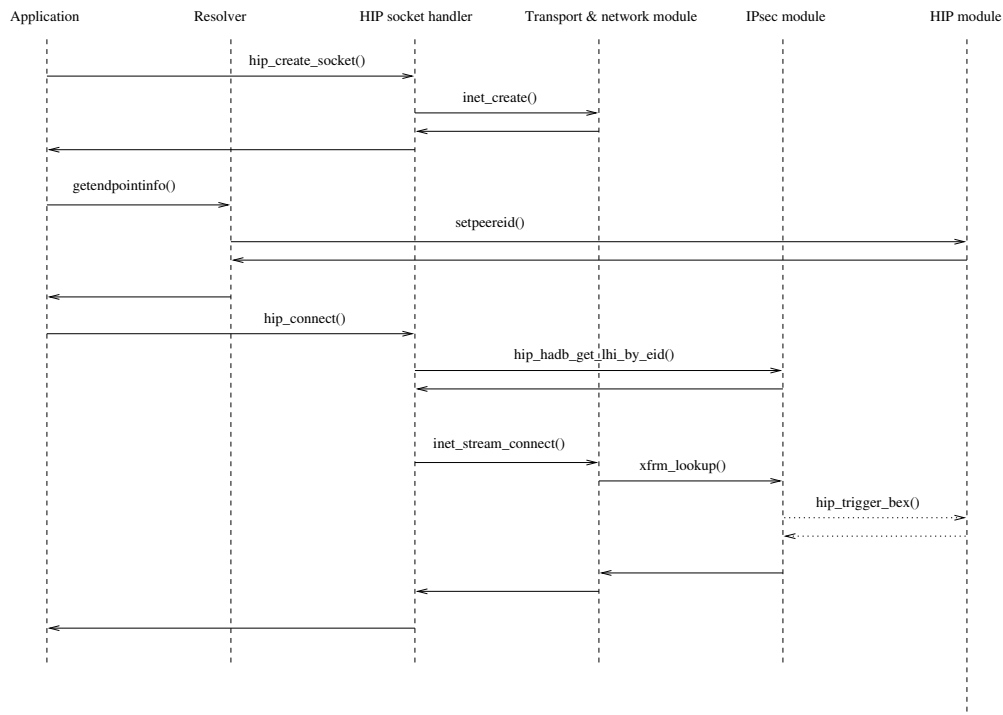


Figure 5.4: Connect sequence diagram

The client initiates the TCP connection using `connect`, which eventually calls `hip_connect` in the HIP socket handler. The socket handler maps the peer EID value to the peer HIT with `hip_hadb_get_lhi_by_eid`. The handler stores the HIT into the `socket` structure. The socket handler calls the `inet_stream_connect` in the trans-

port/network module to initiate TCP handshake. The transport/network module routes the first SYN packet and the global HIT based SP triggers IPsec processing. If the IPsec module cannot find a valid SA, it triggers a base exchange and the application sleeps until the SA is established. If a valid SA exists, the client sends the SYN packet encapsulated in an ESP packet. Finally, the application resumes its control.

There is one thing that is not illustrated in the figure, but is worth mentioning. The `inet_stream_connect` call also triggers `inet_autobind`, because the client application does not make an explicit bind. The call assigns an ephemeral port for the socket. The source EID is bound to a default HI of the host and the corresponding locator set is assigned to any interface available on the host.

5.5.3 Sending Data

Now it is time for the client application to send some data to the server application. The execution path is illustrated in Figure 5.5. The application makes a `send` call, which is translated into `hip_send` call in the HIP socket handler. There is no need for a EID to HIT conversion here as the corresponding `socket` structure has already been configured to use the peer HIT as the destination IPv6 address.

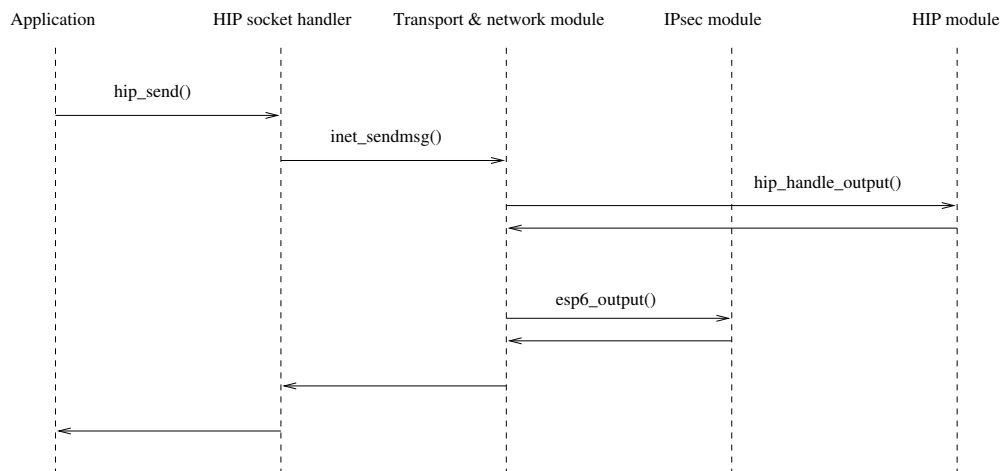


Figure 5.5: Send diagram

The socket handler calls `inet_sendmsg` in the transport/network module to transmit the data into the network. However, `hip_handle_output` hook intercepts the packet and notices that the destination address is a HIT, instead of an IPv6 address. The hosts have already established SAs, and the base exchange will not be triggered. Instead, the source and destination HITs are replaced with IPv6 addresses in the `hip_handle_output`. Finally, the hooks return the control of flow to the transport/network module, which encapsulates the packet into an ESP envelope using

`esp_output` and transmits it to the network.

5.5.4 Receiving Data

The data retrieval is illustrated in Figure 5.6. The server application calls `recv` to receive the data from the client application. The `recv` is translated into a `inet_recvmsg` call, which blocks until some data is received. The data packet arrives from the network and eventually enters the `xfrm6_rcv` function, which handles the ESP processing in the packet. The function is hooked with the `hip_handle_esp` to replace the IPv6 addresses with HITs so that the proper SA can be found in the IPsec module. Finally, the IPsec module wakes up the application to read the data from the socket.

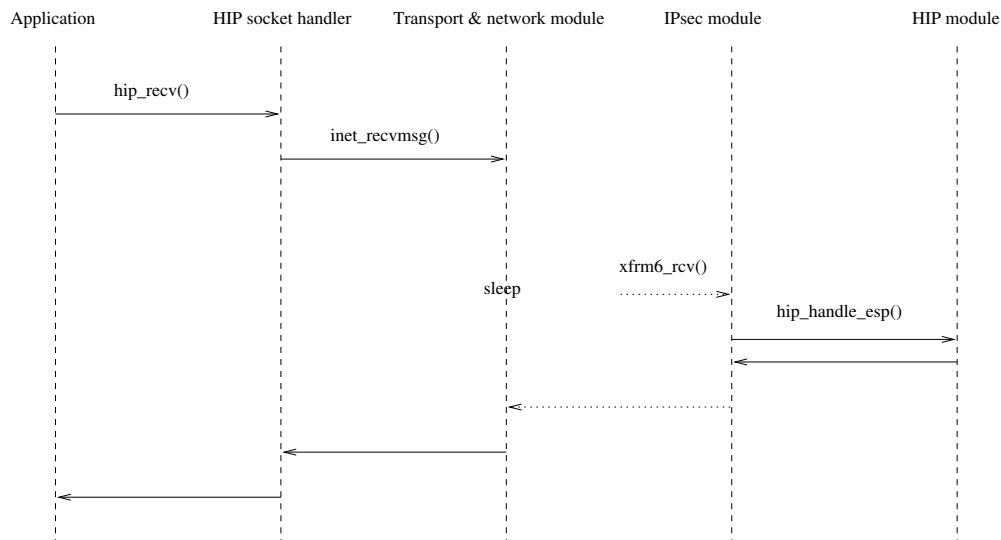


Figure 5.6: Recv diagram

5.6 HIP Enabled Telnet

IPv6 enabled telnet client and telnet daemon from [44] were ported to use the native HIP API. The native HIP API was configured as compile time option into the code.

The porting process was quite straightforward. It was merely enough to convert the `getaddrinfo` and `struct addrinfo` strings in the source code to the HIP correspondents, `getendpointinfo` and `struct endpointinfo`. The prefixes of the member names in the `addrinfo` and `endpointinfo` structures were different and had to be converted too.

Two test applications that were used in the initial stages of the development can be

found the appendix. They are similar to the client and server applications used in the example scenario in this chapter.

Chapter 6

Analysis

In this section, the design of the native HIP API is analyzed on a conceptual level. The analysis includes evaluation of the requirements against the design and description of the most important design alternatives.

6.1 Evaluation

The requirements are evaluated against the design in this section. The evaluation is organized around the new concepts that were introduced to the existing sockets API.

6.1.1 Socket Family

The legacy HIP API may sound attractive for application developers, because it requires none or just a few modifications in the application code. In a way, it is also deceptive at the same time, because some applications are tightly integrated to the TCP/IP protocol suite (e.g. ping, tcpdump and other diagnostic tools) and enabling HIP too hastily may break them or result unexpected behavior. This was recorded as a backwards compatibility requirement in 3.1.2 to avoid repeating the legacy HIP API behavior in the native HIP API.

The new socket family, `PF_HIP`, helps to meet this requirement by making the application more HIP aware. The developer must use the HIP family both for the `socket` and `getendpointinfo` calls before HIP connection can be established. It reminds the developer that the application may need also other modifications to make it work properly.

Besides making the application HIP aware, the family can be used for detecting the capabilities of the hosts. The HIP capability of the localhost is detected when trying to create a HIP socket or when resolving localhost HIs. Basically, the HIP capability of the peer is detected when the resolver is called, unless the peer does not have its

HIIs in the DNS for some reason.

The `AF_HIP` constant is an alias for the `PF_HIP`. There could have some benefit from making a difference between the two, but we did not consider it necessary. Besides, the distinction between the `AF` and `PF` prefixes has already been blurred in the sockets API [39] and it may be difficult to repair the damage.

The new socket family helps to isolate the HIP code from the other networking stack code in the implementation. The HIP socket handler was registered as a separate socket handler into the networking stack to avoid modifying the existing socket handlers. The socket handler was also successfully integrated into the existing HIP module without breaking the legacy HIP API capability of the module.

6.1.2 Endpoint Identifier Descriptor

The EID represents the endpoint for an application. The EID can be associated to both fixed sized identifiers (HITs) and variable sized identifiers (HIIs) as stated in the representation requirements in 3.2.1. Further, the identifiers are dynamically associated to the locators. This approach integrates seamlessly to the mobility and multihoming architecture of HIP. It also makes the transition to the IPv6 more transparent.

The relationship between an EID and a socket is many to one. This way, it is possible to reuse the same EID for multiple sockets. For example, consider a web browser that opens multiple sockets and lets the user to specify his own HI. The `setmyeid` call needs to be called only once because the same EID can be reused for multiple sockets.

The need for different representations of HIIs, such as the HIT, is diminished in the application layer because the EID replaces them in most cases. Consequently, the need for the upper bits in the HIT to distinguish it from an IPv6 address becomes almost superfluous. The most significant need for the upper bits of the HIT in the application layer is to support the legacy HIP API.

The EID value is stored in the `sockaddr_eid` structure, because the sockets API functions depend on the socket address structures. The other reason for the existence of the structure is to avoid confusion. As the EID value is an integer like the socket descriptor, there is a chance for the novice application developer to mistake it for the socket descriptor. It is safer to keep the EID value inside the socket structure.

One might argue that the EID concept is purely a HIP layer identifier and therefore should not contain a port number. However, the EID socket structure contains the port number for a two reasons. The first reason is to maintain the compatibility with the sockets API as the port number is also contained in the `sockaddr_in` and `sockaddr_in6` structures. Second, it makes the determination of the source and destination port number easier in the HIP socket handler. Otherwise the port number should be passed using the `setmyeid` or `setpeereid` function to the HIP module. The problem is that the HIP module cannot associate unambiguously the port number along with the associated EID value to the corresponding socket. This

problem could be solved by adding the socket file descriptor to the arguments of the EID setting function. This would require the socket descriptor argument to be added also to the resolver, because the resolver uses the same EID setting functions. The descriptor is not included in the arguments of the resolver function, because it would reduce the similarity with the sockets API resolver.

The `HIP_HI_ANY`, `HIP_HI_ANY_PUB` and `HIP_HI_ANY_ANON` macros can be set directly into the `eid_val` field in the `sockaddr_eid` structure, thus requiring no `setmyeid` call. If the macros were defined as constant integers, it would increase to complexity of the HIP socket handler, because it would have to handle EID values that are global to all applications. The complexity was avoided by defining the macros as a call to the `setmyeid` call with NULL endpoint, NULL interface arguments and with the appropriate flag set. This way, the special handling for the constant EID value is not needed.

It should be noted that there is also another, albeit minor, benefit from the exclusion of the socket descriptor in the prototypes of the `setmyeid` and `setpeereid` functions. The developer has more freedom in the calling order of the functions, because now the socket does not need to be instantiated with the `socket` function before setting the EID. As consequence, the resolver call is also independent of the socket instantiation even though the resolver uses the `setmyeid` and `setpeereid` functions internally.

6.1.3 Application Specified Identifiers

The requirement for the application specified identifiers is met both in the design and implementation. The `load_hip_endpoint_pem` function can be used to create an application specified endpoint with the help of the `setmyeid` or `setpeereid` functions. The application specified identifier scheme was tested with the test application in section A.3.

6.1.4 Resolver

The resolver simplifies the task of mapping HIs to locators by doing it transparently on the behalf of the application. The resolver outputs a set of EIDs, which the application can input directly to the sockets functions, such as the `bind` and `connect`. Still, the real representation of the HIs and locators behind the EID can be revealed with the `getmyeidinfo` or `getpeereidinfo` functions when needed.

Interfaces represent the locators of the localhost and IP addresses represent the locators of the peer, thus creating an asymmetry between local and peer locators. The asymmetry can be viewed as a minor usability problem as it is not present in the sockets API. Sockets API uses IP address both in the local and peer locator case. However, the resolver effectively hides the asymmetry, because it does not expose the locators to the application.

The interfaces are preferred for the localhost case instead of addresses, because

they are more stable than addresses. Relying on the interfaces instead of protocol dependent IP addresses supports also the idea of a seamless transition to IPv6. The interfaces can also be utilized better than the IP addresses in the HIP layer, because the mobility and multihoming mechanisms in the HIP protocol [31] are also based on IP address groups, i.e. interfaces. Further, designing a QoS policy API will be easier if based on interfaces rather than addresses.

The interface of the native HIP API resolver function along with its companion data structure, `getendpointinfo` and `endpointinfo`, resemble closely their sockets API counterparts, `getaddrinfo` and `addrinfo`. In fact, the `getendpointinfo` functionality could have been integrated into the `getaddrinfo` function because the syntax is almost identical. The deployment of the native HIP API would be more transparent with this approach. However, the names of the sockets API counterpart function and data structure are a bit misleading from the HIP point of view. We chose to emphasize the semantical differences between the sockets API and the native HIP API resolver, and isolated them from each other. This also leaves us more freedom to modify the native HIP API resolver to suit future needs that may even be syntactically incompatible with the sockets API resolver.

6.2 Design Alternatives

The design alternatives have played an important role to the outcome of the native HIP API. Many different alternatives have been considered and discarded. It is therefore mutually important to analyze the “invisible” part of the design.

6.2.1 IP Address Policy Based Approach

One design approach is to keep the sockets API unmodified and use HIP transparently from the application. In this approach, the application uses IP addresses as endpoints. The use of HIP is controlled in the HIP layer with a policy that is based on the IP address of the peer. The policy asserts that HIP will be used for a certain set of destination IP addresses. The application initiates connections to the peer using IP addresses as normally in the sockets API, but the HIP layer intercepts the connections matching to the policy and uses HIP for the connection transparently from the application.

The transparency is both an advantage and a drawback. The major advantage is the low deployment cost. The applications do not need any changes. The connections are prone to man-in-the-middle attacks if the policies are configured to use HIP in opportunistic mode. On the other hand, configuring the peer HIs manually to the policies results an administrative chaos if the policies are applied on each end-host. Another drawback is also that the application is “fooled” to using an IP address even though it will be using a HI. This may have some impact on QoS sensitive or TCP/IP dependent applications. To guarantee that even those applications work

too, they need to be changed, which brings us one step closer to the native HIP API model.

6.2.2 Host Identifier Based Approach

Another approach is that the application uses directly a HIT or even full HI as an endpoint identifier. In the native HIP API, the HIT based approach was not preferred to avoid breaking the forward compatibility of applications. The HIP specifications have not been completely stabilized yet and it is even possible for sizes of the HITs to change. The full HI does not have this limitation, because it represents the whole HI and it is variable sized by its nature. The problem with the HI approach is that the sockets API supports only socket addresses with very limited size. Not even the `sockaddr_storage` is sufficiently large for storing HIs.

The size of an EID is constant and small. It can be used effectively in the sockets API. The EID has also couple of cons compared to the HIT. The EID does not break the forward compatibility of applications, because it acts as a “pointer” or “handle” to the identity itself.

The HIT based approach has at least one advantage over the EID based approach. The HIP socket handler does not have to map between the EIDs and the HITs in the HIT based approach. As a consequence, the purpose of the `setmyeid` and `setpeereid` functions is slightly different. The EID argument is no longer needed for them and their purpose is just to associate the HIs to a set of locators.

6.2.3 Shared Data structure for Identifier and Locator

The EID socket address structure is not semantically identical to the other socket addresses structures, such as `sockaddr_in` and `sockaddr_in6`. The reason for the semantical difference is that the EID is an opaque handle to the HI and associated locators. The EID value cannot be used as a “referral”, i.e. passed from an application to another as it is.

Let us consider an alternative model to the EID based model where there is no need for the EID concept. In this model, the information referenced by the EID, the HI and the locators, are stored directly in a socket address structure as shown in Figure 6.1. The model is similar to [22].

The most significant benefit of this scheme is that the socket structure could be used as it is for sockets API calls. The structure already contains a HI and the corresponding locators. There is no need to call any mapping function, such as the `setmyeid` or `setpeereid`, before the socket structure can be used. The resolver just outputs `sockaddr_hip` structures but it does not need to call any mapping function. Only when the application specifies its own identifier, it needs to call a separate function for communicating the identifier to the HIP module.

This model has several problems. Most importantly, the endpoint structure may be

```

struct sockaddr_hip {
    struct endpoint hip_endpoint; /* Union of HI and HIT */
    union {
        struct sockaddr_storage ai_addr[HIP_MAX_LOCATORS];
        struct if_nameindex      ai_iface[HIP_MAX_LOCATORS];
    } hip_locators;
}

```

Figure 6.1: An alternative EID socket address structure

too large to fit directly into the socket address if the endpoint is a public key. In practice, only HITs could be used in the socket address structure.

The locators may also change after the connection has been initiated and the change cannot be reflected in the application. The locators become just nonce for the application as they will be never updated. The locator union may also be more difficult to handle from the usability point of view. Having variable, albeit limited, number of locators is unnatural for a socket address structure in the sockets API, as there are no other socket address structures of such kind. It was also specified in the requirements that the number of locators the API can handle should not be constrained.

6.2.4 Endpoint Identity Descriptor Based Binding Model

The TLI pair consists of source HI, source port, destination HI and destination port. Consider an extension to this model where the HIs would be replaced with EIDs. The TLI pair would then consist of source EID, source port, destination EID and destination port. This extension would allow a more elaborate kind of binding model, which is illustrated in the example connection association below:

$$\begin{aligned}
 1 &: \{EID_A^{src}, PORT_A^{src}, EID_B^{dst}, PORT_B^{dst}\} \\
 2 &: \{EID_C^{src}, PORT_A^{src}, EID_B^{dst}, PORT_B^{dst}\}
 \end{aligned}$$

The associations are within a single host. The source EIDs, EID_A^{src} and EID_C^{src} , are distinct EIDs, but are associated to the same host identity HI_A^{src} . The destination EID, EID_B^{dst} , is associated to the host identity HI_B^{dst} and it is used in both of the connection associations. This setup is possible only because the associations are based on EIDs rather than HIs.

Although this association model may seem appealing, it is incompatible with the HIP architecture. The host receives a packet from the network destined for the HI of the host, but cannot determine the mapping from the HI to the EID unambiguously because the HI can be mapped to multiple EIDs.

6.2.5 Alternative Resolver Model

It is the responsibility of the resolver to map identifiers and locators to EIDs in the the native HIP API. An alternative to this is to delegate the responsibility to the application as shown in Figure 6.2.

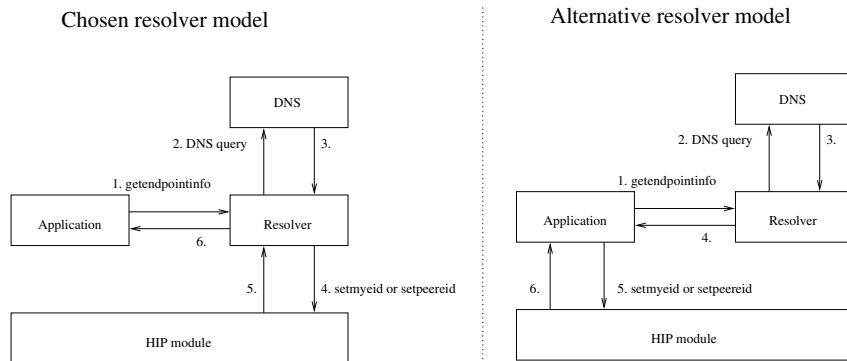


Figure 6.2: The resolver is responsible for the mapping the HI and locators to an EID in the chosen resolver model. In the alternative model, the responsibility is delegated to the application.

In this alternative model, the application must always call the `setmyeid` and `setpeereid` functions. To achieve this, the resolver always outputs explicitly the HIs and locators to the application. The application selects a HI and the associated locators and passes them to the `setmyeid` or `setpeereid` function to receive the corresponding EID. The interfaces to the functions remain syntactically the same as in the native HIP API, but the `endpointinfo` structure is altered as depicted in Figure 6.3.

This `endpointinfo` structure has two differences compared to the original. The endpoint is the public key of the endpoint instead of the EID. The structure also has a new union for the locators.

The usability and representation requirements of the native HIP API state that the application should not be exposed to the HIs and locators unless the application explicitly requires that. The alternative model fails meet this requirement as it always exposes the HIs and locators to the application. In addition, it does not follow the design of the sockets API strictly enough, because it requires always the `setmyeid` or `setpeereid`, to be called in all network applications.

```
struct endpointinfo {
    int ei_flags;
    int ei_family;
    int ei_socktype;
    int ei_protocol;
    size_t ei_endpoint_len;
    struct endpoint *ei_endpoint;
    union {
        struct addrinfo *ai_addr;
        struct if_nameindex *ai_iface;
    } ei_locators;
    char *ei_canonname;
    struct endpointinfo *ei_next;
};
```

Figure 6.3: The alternative resolver data structure

Chapter 7

Future Work

The design and implementation efforts spent on the native HIP API have brought up some future research and development ideas that are described in this chapter.

7.1 Design

The design related research ideas are discussed in this section.

7.1.1 Endpoint Identifier Descriptor

A couple of EID related functions are needed to make the API more complete. Comparison of EID values needs a new system function, because the design does not give any guarantees of the numerical properties of the EID value. Passing an EID to another process requires a new system function, as well the duplication of an EID within the same process. If the EID was be a real file descriptor, the duplication could be implemented with the existing `dup` [39] system function.

The endpoint identifiers in the HIP model are stable and the locators are ephemeral. In the future, even the identifiers could be allowed to change while providing persistent transport layer connections. In this model, the binding between the EID and HI is dynamic instead of static, i.e. the relationship between EID and HI is many-to-many. The application has a stable EID but the identifiers associated to the EID can be allowed to change transparently from the application. The benefit of such a HI mobility feature is questionable, but the EID concept would ease the development such as a feature as it adds a layer of indirection.

The EID concept could also be useful to other mobility related protocols, especially to those based on identity-locator split. There seems to be so many mobility related proposals that either do not have an API yet or deploy their own protocol specific APIs. The EID concept could be generic enough to be used in other protocols too, but it requires further analysis and experimentation.

7.1.2 Host Identifiers

The definitions of the `endpoint_hip` structure and `load_hip_endpoint_pem` function in section 4.2 were mostly designed to be compatible with the existing HIP implementation. As such, they may need revising before they can be deployed in other HIP implementations. The format of the `endpoint_hip` is suitable at least for **DSA!** (DSA) keys, but other types of keys were not expirimented because the implementation supports only DSA. The loading of public or private keys could be supported in other formats than Privacy Enhanced Mail (PEM), such as “SSH Public Key File Format” defined in [5].

7.1.3 Locators

The DNS **RR!**s (RRs) are “flat” by their nature instead of being hierarchical. The FQDN of a host is associated to a set of HIs and locators. Within the set of HIs and locators, it is not possible to associate an individual HI to a specific locator as illustrated in Figure 7.1. If a host has only one HI stored in the DNS, the relationship between the HI and the locators is obvious. However, the situation is different when the host has multiple HIs. It is not possible exclude a locator from belonging to a specific HI, because of the design of the resource records. This is the reason why the resolver just associates all of the peer locators redundantly with each HI.

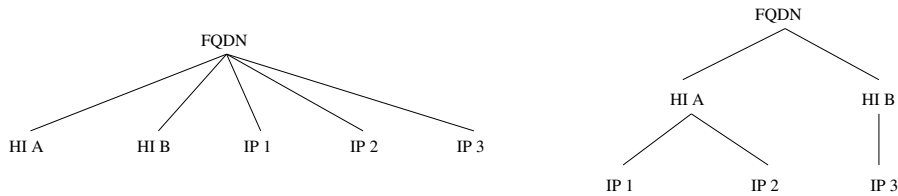


Figure 7.1: In the current DNS RR model (left), the FQDN is associated to set of HIs and locators, and it is not possible attach a certain HI to specific locators. The right side of the figure illustrates the case where this restraint would not exist.

7.1.4 Referrals

Some applications use the IP addresses as referrals, meaning that they pass addresses from one endpoint to another within the protocol. For example, File Transfer Protocol (FTP) applications require referrals. The EID is not a referral and it is not suitable for passing from an FTP application. If an FTP application were to use the native HIP API, it would have to query the host either for the HIs or locators corresponding to the EID, and use it as a referral. This area of applications still needs further work.

7.1.5 Rendezvous Server

[3] specifies a rendezvous model for communication between HIP-hosts and non-HIP hosts. In some scenarios, it is better to use this kind of rendezvous server. In other scenarios, the option of falling back to plain TCP/IP networking is more appealing. It may also benefit the applications and HIP implementations if the resolver differentiates between HIP enabled hosts and the addresses belonging to a HIP rendezvous server, especially in the HIP-to-HIP rendezvous server case.

7.1.6 Protocol Integration

The HIP specifications do not give an explicit statement on the function that should ultimately trigger the base exchange from the userspace. Implicitly, it should be triggered by the function that causes the first packet to be sent to the peer. This function is usually the `connect` function for in the case of TCP and the `sendto` function in the case of UDP. However, the base exchange could also be triggered e.g. with the `setsockopt` function, or in the `setpeereid` function. The analysis of this scheme is not complete, but some thoughts are given below.

If the base exchange would be triggered from the `setpeereid` function, it would be desirable to disable the triggering by default. There may be some applications that use the `getendpointinfo` resolver function with no intention of sending or receiving any data to or from the network. There may be only a few applications of this kind. Still, they generate unnecessary load on the hosts and waste bandwidth because calling `getendpointinfo` calls also `setpeereid` when resolving peer HIs.

There may be some benefit from triggering the base exchange before the application sends any data to the network. At least, it would give more control to the application.

Evaluation of the compatibility with other APIs, such as LIN6 [22] and SCTP [40], was not in the scope, but we still represent a few observations. The first argument, the domain, is used both in the native HIP and LIN6 APIs. The domain is set to `PF_HIP` in the native HIP API and to `AF_LIN6` in the LIN6 API. The SCTP API [40] sets the third argument, the protocol, to `IPPROTO_SCTP` to create an SCTP based socket. There is an obvious conflict between native HIP and LIN6 APIs, because they use the same argument to enable the protocol. It strongly implies that the socket cannot support both LIN6 and HIP at the same time. However, the same conflict does not occur with SCTP, as the argument for the selection of HIP and LIN6 is not the same. A further analysis of the interoperability with the other API functions, as well as a protocol level interoperability analysis, remains to be a future research item.

7.1.7 Events

SCTP has an API for receiving information on SCTP events [40]. The application indicates its willingness to listen for a specific SCTP event type by setting a socket

option. The events are received “using a normal data channel” [41] via `recvmsg` call. The output of the function call includes an indication on whether the output is data from the peer endpoint or an SCTP notification message.

A similar event notification interface should be defined for HIP, too. It would benefit, at least, some diagnostic applications and real time applications sensitive to changes in the network QoS parameters. This kind of applications could register for listening to UPDATE events. As another example, the application could be registered for listening to opportunistic base exchanges. The application could then prompt the user to accept the key of the peer like in Secure Shell (SSH).

7.1.8 Policy API

The native HIP API could be extended with an interface for setting application specified QoS related parameters. The `setsockopt` interface is probably the most straightforward way to implement the policy API. The second argument of the `setsockopt`, the level, is set to `IPPROTO_HIP` and a policy structure is given as the last fourth argument. The policy structure must have a standardized format, but defining one is out of the scope. The task of setting local and peer policies may become simpler because of the notion of the source and destination EIDs.

7.1.9 Standardized Interface to the HIP Module

A standardized interface for communicating host identifiers and other related information between the application and the HIP module could be useful in the future. Consider a Linux based system that has several independent HIP module implementations but the native HIP API functionality is implemented within a single `libc` library. The library needs either to understand each implementation specific communication interface or it needs to understand a single communication interface shared by all implementations. We find the latter alternative more appealing. The interface could be based on e.g. `PF_KEY` [23] or `NETLINK` [36]. Alternatively, each implementation could have a separate native HIP API library that needs to be linked explicitly into the application. This approach may not be very convenient and it also wastes implementation effort on many redundant library implementations.

7.2 Implementation

Some data structures could have been stored in the process context, such as the EID data structures described in section 5.4. Some functionality could have been implemented easier, if the related data structures were stored in the process context. For example, if the key material passed to the `setmyeid` function is probably easier to deallocate in the HIP module when a process exits. However, we tried to avoid modifying the existing networking stack as much as possible, and all of the data

structures were implemented in the HIP module.

Some requirements were not implemented. The ownership permission checking of EIDs was implemented only partially. The HIP module discards the interfaces for local EIDs. The use of interfaces should be integrated better into the UPDATE support. The `HIP_HI_ANY` constant along with its variants were not supported in the implementation. HIP specific socket options were not implemented. Fall back to IPv6 was not implemented.

The resolver library did not support DNS. It supported just the `/etc/hosts` file. It did not make a difference between rendezvous servers and endhosts, because denoting the rendezvous server in the `/etc/hosts` file breaks the existing resolver library. The resolver supported only public/private keys for localhost resolving and HITs for peer resolving. Interface selection by specifying the address family was not implemented as the implementation supported only IPv6 addresses. The `getmyeidinfo` and `getpeereidinfo` functions were not implemented either.

Chapter 8

Conclusion

The design of the native HIP API meets the requirements. The API follows the design of the sockets API closely and extends it only when reuse is not possible. The API increases the application's control over the HIP SAs. The advanced applications can control also HI and locator bindings explicitly. Typical applications just use the new endpoint resolver to hide the details of HIs and locators.

The EID provides the means to conceal details of the HIs and ephemeral locators. The EID is an opaque handle to a HI and it can be used directly in the sockets API function calls. The EID needs support in the networking stack.

The API allows the application to fall back to plain TCP/IP networking if the peer host does not support HIP. It is also possible to explicitly request for "opportunistic HIP mode" if the application is willing to establish a connection without a prior knowledge of the HI of the peer. The application can also specify its own HI and delegate the right to use the key to the host.

The kernelspace component of the implementation is the HIP socket handler, which was built on top of an existing kernel based HIP implementation. The socket handler is isolated from the rest of the networking stack by introducing the HIP specific protocol family `|PF_HIP|`. The isolation is necessary to avoid breaking the backwards compatibility with the existing sockets API.

The userspace component is the resolver library. It provides the new endpoint resolver. The resolver is syntactically almost identical to the sockets API resolver for ease of use. The implementation was experimented by porting a telnet client and server to use the API. The porting process was quite straightforward. The telnet client opened a TCP connection to the telnet server using HIP and opened a login terminal successfully to the server.

The length of the HIT may be inadequate some day. The EID approach guarantees forward compatibility with HIP as there is little need to reflect the change to the applications. Also, the transition to IPv6 is more transparent as the EID hides the details of IP addresses. The binding model used for the EID is quite flexible, because

it allows the reuse of an EID for multiple sockets.

There seems to be an abundance of different mobility related protocol proposals. Some of them lack an API altogether and the others have their own protocol specific APIs. It would be beneficial to provide an API generic enough that could be used in most of them. The EID approach is quite generic and could be worth analyzing with other related protocols, or at least with protocols based on the endpoint identifier/locator split. This requires further research and evaluation.

Bibliography

- [1] Catharina Candolin, Miika Komu, Mika Kousa, and Janne Lundberg. An implementation of HIP for linux. In *Proceedings of the Linux Symposium 2003, Ottawa, Ontario Canada, 23-26 July 2003 pp. 97-105*, July 2003. <http://archive.linuxsymposium.org/ols2003/Proceedings/>.
- [2] J. Noel Chiappa. *Endpoints and Endpoint Names: A Proposed Enhancement to the Internet Architecture*, 1999. <http://users.exis.net/~jnc/tech/endpoints.txt>.
- [3] Lars Eggert and Julien Laganier. *Host Identity Protocol (HIP) Rendezvous Mechanisms*. IETF, February 2004. [Internet Draft] <http://www.ietf.org/internet-drafts/draft-eggert-hip-rendezvous-00.txt>.
- [4] P. Florissi, Y. Yemini, and D. Florissi. *QoSockets: a New Extension to the Sockets API for End-to-End Application QoS Management*, May 1999.
- [5] J. Galbraith and R. Thayer. *SSH Public Key File Format*. Internet Engineering Task Force, August 2003. [Internet Draft] <http://www.vandyke.com/technology/draft-ietf-secsh-publickeyfile.txt>.
- [6] R. Gilligan, S. Thomson, J. Bound, J. McCann, and W. Stevens. *RFC 3493: Basic Socket Interface Extensions for IPv6*. Internet Engineering Task Force, February 2003. <http://www.ietf.org/rfc/rfc3493.txt>.
- [7] Open Group. *IEEE Std. 1003.1-2001 Standard for Information Technology – Portable Operating System Interface (POSIX)*. Open Group, December 2001. <http://www.opengroup.org/austin>.
- [8] A. Gulbrandsen, P. Vixie, and L. Esibov. *RFC 2782: A DNS RR for specifying the location of services (DNS SRV)*. Internet Engineering Task Force, February 2000. <http://www.ietf.org/rfc/rfc2782.txt>.
- [9] E. Guttman, C. Perkins, J. Veizades, and M. Day. *RFC 2608: Service Location Protocol, Version 2*. Internet Engineering Task Force, June 1999. <http://www.ietf.org/rfc/rfc2608.txt>.

- [10] Troels Walsted Hansen. *Multihoming with Internet Protocol Version 6*, December 1999. <http://www.vermicelli.pasta.cs.uit.no/ipv6/students/troels/thesis.pdf>.
- [11] Dan Harkins and Dave Carrel. *RFC 2409: The Internet Key Exchange (IKE)*. Internet Engineering Task Force, November 1998. <http://www.ietf.org/rfc/rfc2409.txt>.
- [12] Thomas Henderson, Jeff Ahrenholz, and et al. Boeing's unpublished hip implementation.
- [13] Thomas R. Henderson. Host mobility for IP networks: A comparison. *IEEE Network Magazine*, 17(6):18–26, November 2003.
- [14] Hip for BSD implementation. <http://www.hip4inter.net/>.
- [15] The HIPL Group. *Host Identity Protocol for Linux*. <http://www.gaijin.iki/hipl/>.
- [16] Malleswar Kalla, Ken Morneault, Vern Paxson, Ian Rytina, Hanns Jürgen Schwarzbauer, Chip Sharp, Randall Stewart, Tom Taylor, Qiaobing Xie, and Lixia Zhang. *RFC 2960: Stream Control Transmission Protocol*. Internet Engineering Task Force, October 2000. <http://www.ietf.org/rfc/rfc2960.txt>.
- [17] Miika Komu. Host identity payload in home networks. Seminar paper, Helsinki University of Technology, Espoo, Finland, April 2002.
- [18] Julien Laganier. Julien laganier's unpublished hip implementation.
- [19] J. Linn. *RFC 2743: Generic Security Service Application Program Interface Version 2, Update 1*. Internet Engineering Task Force, January 2000. <http://www.ietf.org/rfc/rfc2743.txt>.
- [20] J. Linn. *RFC 2744: Generic Security Service API Version 2: C-bindings*. Internet Engineering Task Force, January 2000. <http://www.ietf.org/rfc/rfc2744.txt>.
- [21] Jukka Manner and Markku Kojo. *RFC 3753: Service Location Protocol, Version 2*. Internet Engineering Task Force, June 2004. <http://www.ietf.org/rfc/rfc3753.txt>.
- [22] Arifumi Matsumoto. *LIN6 Multihoming API*. Internet Engineering Task Force, January 2004. [Expired Internet Draft].
- [23] D. McDonald, C. Metz, and B. Phan. *RFC 2367: PF_KEY Key Management API, Version 2*. Internet Engineering Task Force, July 1998. <http://www.ietf.org/rfc/rfc2367.txt>.
- [24] A. McGregor. Pyhip release 18 march 2003. <http://www.sharemation.com/adm01bass/pyhip-2003-03-18.tar.bz2>.

- [25] Robert Moskowitz. *Host Identity Payload Implementation*. Internet Engineering Task Force, February 2001. [Internet Draft] <http://homebase.htt-consult.com/~hip/draft-moskowitz-hip-impl-01.txt>.
- [26] Robert Moskowitz and Pekka Nikander. *Host Identity Payload Architecture*. Internet Engineering Task Force, September 2003. [Internet Draft] <http://www.ietf.org/internet-drafts/draft-moskowitz-hip-arch-05.txt>.
- [27] Robert Moskowitz, Pekka Nikander, Petri Jokela, and Thomas Henderson. *Host Identity Protocol*. Internet Engineering Task Force, February 2004. [Internet Draft] <http://www.ietf.org/internet-drafts/draft-ietf-hip-base-00.txt>.
- [28] P. Nikander and J. Laganier. *Using the Domain Name System (DNS) with the Host Identity Protocol*. IETF, May 2004. [Internet Draft] <http://julien.laganier.free.fr/pub/draft-nikander-hip-dns-00pre1.txt>.
- [29] Pekka Nikander. *An Address Ownership Problem in IPv6*. Internet Engineering Task Force, February 2001. [Expired Internet Draft] <http://www.tml.hut.fi/~pnr/publications/draft-nikander-ipng-address-ownership-00.txt>.
- [30] Pekka Nikander. A case for host identity payload: An architecture for multi-homed mobile hosts, February 2002. unpublished manuscript.
- [31] Pekka Nikander and Jari Arkko. *End-Host Mobility and Multi-Homing with Host Identity Protocol*. Internet Engineering Task Force, December 2003. [Internet Draft] <ftp://ftp.funet.fi/internet-drafts/draft-nikander-hip-mm-01.txt>.
- [32] Pekka Nikander, Jorma Wall, and Jukka Ylitalo. Integrating security, mobility, and multi-homing in a HIP way,. In *Proceedings of Network and Distributed Systems Security Symposium*, pages 87–99, San Diego, California, February 2003. Internet Society. <http://www.tcm.hut.fi/~pnr/publications/NDSS03-Nikander-et-al.pdf>.
- [33] Erik Nordmark. *Multihoming without IP Identifiers*. IETF, October 2003. [Expired Internet Draft].
- [34] C. Perkins. *RFC 3344: IP Mobility Support for IPv4*. Internet Engineering Task Force, August 2002. <http://www.ietf.org/rfc/rfc3344.txt>.
- [35] Jon Postel. *RFC 793: Transport Control Protocol*. Internet Engineering Task Force, September 1981. <http://www.ietf.org/rfc/rfc793.txt>.
- [36] Netlink S.a.s. Netlink - communication between kernel and user. [//www.netlink.it/](http://www.netlink.it/).
- [37] Kristian Slavov. *Implementing HIP Algorithms in Linux Kernel*. TKK, January 2004.

- [38] W. Stevens, M. Thomas, E. Nordmark, and T. Jinmei. *RFC 3678: Advanced Sockets Application Program Interface (API) for IPv6*. Internet Engineering Task Force, May 2003. <http://www.ietf.org/rfc/rfc3678.txt>.
- [39] W. Richard Stevens. *UNIX Network Programming, Volume 1: Networking APIs: Sockets and XTI*. Prentice Hall, Upper Saddle River, New Jersey, 2nd edition, 1997.
- [40] R. Stewart, L. Yarroll, J. Wood, K. Poon, K. Fujita, and M. Tuexen. *Sockets API Extensions for Stream Control Transmission Protocol (SCTP)*. Internet Engineering Task Force, April 2004. [Internet Draft] <http://www.ietf.org/internet-drafts/draft-ietf-tsvwg-sctpsocket-08.txt>.
- [41] Randall R. Stewart and Xiaobing Xie. *Stream Control Transmission Protocol (SCTP)*. Addison-Wesley, November 2001.
- [42] Fumio Teraoka, Masahiro Ishiyama, and Mitsunobu Kunishi. *LIN6: A Solution to Multihoming and Mobility in IPv6*. Internet Engineering Task Force, December 2003. [Internet Draft] <http://www.ietf.org/internet-drafts/draft-teraoka-multi6-lin6-00.txt>.
- [43] Jae H. Kim Thomas R. Henderson, Jeffrey M. Ahrenholz. Experience with the host identity protocol for secure host mobility and multihoming. In *IEEE Wireless Communications and Networking Conference*, March 2003.
- [44] USAGI project - linux IPv6 development project. <http://www.linux-ipv6.org/>.
- [45] John Viega, Matt Messier, and Pravir Chandra. *Networking Security with OpenSSL*. O'Reilly, jun 2002.
- [46] Brian Wellington. *RFC 3007: Secure Domain Name System (DNS) Dynamic Update*. Internet Engineering Task Force, November 2000. <http://www.ietf.org/rfc/rfc3007.txt>.
- [47] Joel M. Winett. *RFC 0147: The Definition of a Socket*. Internet Engineering Task Force, May 1971. <http://www.ietf.org/rfc/rfc0147.txt>.
- [48] J. Ylitalo, P. Jokela, J. Wall, and P. Nikander. *End-point identifiers in Secure Multihomed Mobility*, December 2002. <http://www.tml.hut.fi/~pnr/publications/NDSS03-Nikander-et-al.pdf>.
- [49] Jukka Ylitalo and Pekka Nikander. Blind: A complete identity protection framework for end-points. In *Twelfth International Workshop on Security Protocols, Cambridge, England*, April 2004.

Appendix A

Application Code Examples

A.1 Connection Test Server

```
/*
 * Echo server: get data from client and send it back. Use this with
 * comntest-client-native.
 */
#ifdef HAVE_CONFIG_H
#include <config.h>
#endif

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <netinet/in.h>
#include <netinet/ip.h>
#include <unistd.h>
#include <netdb.h>
#include <net/if.h>
/* Workaround for some compilation problems on Debian */
#ifndef __user
# define __user
#endif
#include <signal.h>

#include "tools/debug.h"

static void sig_handler(int signo) {
```

```
    if (signo == SIGTERM) {
        // close socket
        printf("Sigterm\n");
        exit(-1);
    } else {
        printf("Signal %d\n", signo);
        exit(-1);
    }
}

int main(int argc, char *argv[]) {
    struct endpointinfo hints, *res = NULL;
    struct sockaddr_eid peer_eid;
    char *port_name;
    char mylovestdata[IP_MAXPACKET];
    int recvnum, sendnum;
    int serversock = 0, sockfd = 0;
    int err = 0;
    int socktype;
    socklen_t peer_eid_len;
    int endpoint_family = PF_HIP;

    if (signal(SIGTERM, sig_handler) == SIG_ERR) {
        err = 1;
        goto out;
    }

    if (argc != 3) {
        printf("Usage: %s tcp|udp port\n", argv[0]);
        err = 1;
        goto out;
    }

    if (strcmp(argv[1], "tcp") == 0) {
        socktype = SOCK_STREAM;
    } else if (strcmp(argv[1], "udp") == 0) {
        socktype = SOCK_DGRAM;
    } else {
        printf("error: unknown socket type\n");
        err = 1;
        goto out;
    }

    port_name = argv[2];
```

```

serversock = socket(endpoint_family, socktype, 0);
if (serversock < 0) {
    perror("socket");
    err = 1;
    goto out;
}

memset(&hints, 0, sizeof(struct endpointinfo));
hints.ei_family = endpoint_family;
hints.ei_socktype = socktype;

err = getendpointinfo(NULL, port_name, &hints, &res);
if (err) {
    printf("Resolving of peer identifiers failed (%d)\n", err);
    goto out;
}

if (bind(serversock, res->ei_endpoint, res->ei_endpointlen) < 0) {
    perror("bind");
    err = 1;
    goto out;
}

if (socktype == SOCK_STREAM && listen(serversock, 1) < 0) {
    perror("listen");
    err = 1;
    goto out;
}

while(1) {
    if (socktype == SOCK_STREAM) {
        sockfd = accept(serversock, (struct sockaddr *) &peer_eid,
            &peer_eid_len);
        if (sockfd < 0) {
perror("accept");
err = 1;
goto out;
        }

        while((recvnum = recv(sockfd, mylovemostdata,
            sizeof(mylovemostdata), 0)) > 0 ) {
mylovemostdata[recvnum] = '\0';
if (recvnum == 0) {

```

```
    break;
}

/* send reply */
sendnum = send(sockfd, mylovemostdata, recvnum, 0);
if (sendnum < 0) {
    perror("send");
    err = 1;
    goto out;
}
}
} else { /* UDP */
    sockfd = serversock;
    while(recvnum = recvfrom(sockfd, mylovemostdata,
        sizeof(mylovemostdata), 0,
        (struct sockaddr *)&peer_eid,
        &peer_eid_len) > 0) {
mylovemostdata[recvnum] = '\0';
printf("%s", mylovemostdata);
if (recvnum == 0) {
    break;
}
}

/* send reply */
sendnum = sendto(sockfd, mylovemostdata, recvnum, 0,
    (struct sockaddr *) &peer_eid, peer_eid_len);
if (sendnum < 0) {
    perror("send");
    err = 1;
    goto out;
}
}
}

out:

if (res)
    free_endpointinfo(res);

if (sockfd)
    close(sockfd); // discard errors
if (serversock)
    close(serversock); // discard errors
```

```

    return err;
}

```

A.2 Connection Test Client

```

/*
 * Echo STDIN to a selected server which should echo it back.
 * Use this application with comntest-server-xx.
 *
 * usage: ./comntest-client-native host tcp|udp port
 *        (reads stdin)
 */
#if HAVE_CONFIG_H
#include <config.h>
#endif

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <netinet/in.h>
#include <netinet/ip.h>
#include <unistd.h>
#include <netdb.h>
#include <sys/time.h>
#include <time.h>
#include <arpa/inet.h>
#include <net/if.h>
#include "tools/debug.h"

int main(int argc, char *argv[]) {
    struct endpointinfo hints, *epinfo, *res = NULL;
    struct timeval stats_before, stats_after;
    unsigned long stats_diff_sec, stats_diff_usec;
    char mylovemostdata[IP_MAXPACKET];
    char receiveddata[IP_MAXPACKET];
    char *proto_name, *peer_port_name, *peer_name;
    int recvnum, sendnum;
    int datalen = 0;
    int proto;

```

```
int datasent = 0;
int datareceived = 0;
int ch;
int err = 0;
int sockfd = -1, socktype;
se_family_t endpoint_family;

if (argc != 4) {
    printf("Usage: %s host tcp|udp port\n", argv[0]);
    err = 1;
    goto out;
}

peer_name = argv[1];
proto_name = argv[2];
peer_port_name = argv[3];
endpoint_family = PF_INET;

/* Set transport protocol */
if (strcmp(proto_name, "tcp") == 0) {
    proto = IPPROTO_TCP;
    socktype = SOCK_STREAM;
} else if (strcmp(proto_name, "udp") == 0) {
    proto = IPPROTO_UDP;
    socktype = SOCK_DGRAM;
} else {
    printf("Error: only TCP and UDP supported.\n");
    err = 1;
    goto out;
}

sockfd = socket(endpoint_family, socktype, 0);
if (sockfd == -1) {
    printf("creation of socket failed\n");
    err = 1;
    goto out;
}

/* set up host lookup information */
memset(&hints, 0, sizeof(hints));
hints.ei_socktype = socktype;
hints.ei_family = endpoint_family;

/* lookup host */
```

```

err = getendpointinfo(peer_name, peer_port_name, &hints, &res);
if (err) {
    printf("getaddrinfo failed (%d): %s\n", err, gpi_strerror(err));
    goto out;
}

printf("family=%d value=%d\n", res->ei_family,
    ntohs(((struct sockaddr_eid *) res->ei_endpoint)->eid_val));

// data from stdin to buffer
bzero(receiveddata, IP_MAXPACKET);
bzero(mylovestdata, IP_MAXPACKET);

printf("Input some text, press enter and ctrl+d\n");

while ((ch = fgetc(stdin)) != EOF && (datalen < IP_MAXPACKET)) {
    mylovestdata[datalen] = (unsigned char) ch;
    datalen++;
}

epinfo = res;
while(epinfo) {
    err = connect(sockfd, (struct sockaddr *) epinfo->ei_endpoint,
        epinfo->ei_endpointlen);
    if (err) {
        perror("connect");
        goto out;
    }
    epinfo = epinfo->ei_next;
}

/* Send the data read from stdin to the server and read the response.
   The server should echo all the data received back to here. */
while((datasent < datalen) || (datareceived < datalen)) {

    if (datasent < datalen) {
        sendnum = send(sockfd, mylovestdata + datasent, datalen - datasent, 0);

        if (sendnum < 0) {
perror("send");
err = 1;
goto out;
        }
        datasent += sendnum;

```

```

    }

    if (datareceived < datalen) {
        recvnum = recv(sockfd, receiveddata + datareceived,
            datalen-datareceived, 0);
        if (recvnum <= 0) {
perror("recv");
err = 1;
goto out;
        }
        datareceived += recvnum;
    }
}

if (memcmp(mylovemostdata, receiveddata, IP_MAXPACKET)) {
    printf("Sent and received data did not match\n");
    err = 1;
    goto out;
}

out:

if (sockfd != -1)
    close(sockfd); // discard errors
if (res)
    free_endpointinfo(res);

printf("Result of data transfer: %s.\n", (err ? "FAIL" : "OK"));

return err;
}

```

A.3 Connection Test Client with Application Specified Identifiers

```

/*
 * Echo STDIN to a selected server which should echo it back.
 * Use this application with conntest-server-xx.
 *
 * usage: ./conntest-client-native-user-key host tcp|udp port
 *        (reads stdin)
 */
#if HAVE_CONFIG_H

```

```
#include <config.h>
#endif

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <netinet/in.h>
#include <netinet/ip.h>
#include <unistd.h>
#include <netdb.h>
#include <sys/time.h>
#include <time.h>
#include <arpa/inet.h>
#include <net/if.h>
#include "tools/debug.h"

int main(int argc, char *argv[]) {
    struct endpointinfo hints, *epinfo, *res = NULL;
    struct sockaddr_eid my_eid;
    struct timeval stats_before, stats_after;
    unsigned long stats_diff_sec, stats_diff_usec;
    char mylovestdata[IP_MAXPACKET];
    char receiveddata[IP_MAXPACKET];
    char *proto_name, *peer_port_name, *peer_name;
    int recvnum, sendnum;
    int datalen = 0;
    int proto;
    int datasent = 0;
    int datareceived = 0;
    int ch;
    int err = 0;
    int sockfd = 0, socktype;
    se_family_t endpoint_family;
    char *user_key_base = "/etc/hip/hip_host_dsa_key";
    struct endpoint *endpoint;

    if (argc != 4) {
        printf("Usage: %s host tcp|udp port\n", argv[0]);
        err = 1;
        goto out;
    }
}
```

```
peer_name = argv[1];
proto_name = argv[2];
peer_port_name = argv[3];
endpoint_family = PF_HIP;

/* Set transport protocol */
if (strcmp(proto_name, "tcp") == 0) {
    proto = IPPROTO_TCP;
    socktype = SOCK_STREAM;
} else if (strcmp(proto_name, "udp") == 0) {
    proto = IPPROTO_UDP;
    socktype = SOCK_DGRAM;
} else {
    printf("Error: only TCP and UDP supported.\n");
    err = 1;
    goto out;
}

sockfd = socket(endpoint_family, socktype, 0);
if (sockfd == -1) {
    printf("creation of socket failed\n");
    err = 1;
    goto out;
}

err = load_hip_endpoint_pem(user_key_base, &endpoint);
if (err) {
    printf("Failed to load user HIP key %s\n", user_key_base);
    goto out;
}

err = setmyeid(&my_eid, "", endpoint, NULL);
if (err) {
    printf("Failed to set up my EID (%d)\n", err);
    err = 1;
    goto out;
}

/* We have to bind to the EID to use it. */
err = bind(sockfd, (struct sockaddr *) &my_eid, sizeof(struct sockaddr_eid));
if (err) {
    perror("bind failed");
    goto out;
}
```

```

}

/* set up endpoint lookup information */
memset(&hints, 0, sizeof(struct endpointinfo));
hints.ei_socktype = socktype;
hints.ei_family = endpoint_family;

/* Lookup endpoint. We do not need to call setpeereid because
   getendpointinfo does it automatically. */
err = getendpointinfo(peer_name, peer_port_name, &hints, &res);
if (err) {
    printf("getendpointinfo failed (%d): %s\n", err, gepi_strerror(err));
    goto out;
}

printf("family=%d value=%d\n", res->ei_family,
       ntohs(((struct sockaddr_eid *) res->ei_endpoint)->eid_val));

// data from stdin to buffer
bzero(receiveddata, IP_MAXPACKET);
bzero(mylovemostdata, IP_MAXPACKET);

printf("Input some text, press enter and ctrl+d\n");

// horrible code
while ((ch = fgetc(stdin)) != EOF && (datalen < IP_MAXPACKET)) {
    mylovemostdata[datalen] = (unsigned char) ch;
    datalen++;
}

gettimeofday(&stats_before, NULL);

epinfo = res;
while(epinfo) {
    err = connect(sockfd, res->ei_endpoint, res->ei_endpointlen);
    if (err) {
        perror("connect");
        goto out;
    }
    epinfo = epinfo->ei_next;
}

gettimeofday(&stats_after, NULL);
stats_diff_sec = (stats_after.tv_sec - stats_before.tv_sec) * 1000000;

```

```
stats_diff_usec = stats_after.tv_usec - stats_before.tv_usec;

printf("connect took %.10f sec\n",
      (stats_diff_sec + stats_diff_usec) / 1000000.0);

/* Send the data read from stdin to the server and read the response.
   The server should echo all the data received back to here. */
while((datasent < datalen) || (datareceived < datalen)) {

    if (datasent < datalen) {
        sendnum = send(sockfd, mylovestdata + datasent, datalen - datasent, 0);

        if (sendnum < 0) {
perror("send");
err = 1;
goto out;
        }
        datasent += sendnum;
    }

    if (datareceived < datalen) {
        recvnum = recv(sockfd, receiveddata + datareceived,
                      datalen-datareceived, 0);
        if (recvnum <= 0) {
perror("recv");
err = 1;
goto out;
        }
        datareceived += recvnum;
    }
}

if (memcmp(mylovestdata, receiveddata, IP_MAXPACKET)) {
    printf("Sent and received data did not match\n");
    err = 1;
    goto out;
}

out:

if (sockfd)
    close(sockfd); // discard errors
if (res)
    free_endpointinfo(res);
```

```
printf("Result of data transfer: %s.\n", (err ? "FAIL" : "OK"));  
  
return err;  
}
```