Miika Komu

Application Programming Interfaces for the Host Identity Protocol

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

HELSINKI UNIVERSITY
OF TECHNOLOGY

ABSTRACT OF THE
MASTER'S THESIS

Author:	Miika Komu
Name of the thesis:	Application Programming Interfaces for the Host Identity Protocol
Date:	9th September 2004 **Number of pages:** 86
Department:	Department of Computer Science and Engineering
Professorship:	T-110
Supervisor:	Ph.D. Kimmo Raatikainen
Instructors:	Ph.D. Pekka Nikander, M.Sc. Jukka Ylitalo

The goal of this thesis was to design and implement an application programming interface for Host Identity Protocol (HIP) aware network applications using the C language. The results of the design are evaluated against the given requirements. Different design alternatives are introduced and analyzed in order to rationalize the design. A reference implementation was developed as a proof of concept. Few example applications were ported to use the API.

The outcome of the design meets the requirements. The API follows the design of the sockets API closely and extends it only when reuse of the design is not possible. The new API increases the control over the HIP layer for 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.

The 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.

Keywords: HIP, native, API, socket, legacy, userspace, application

TEKNILLINEN KORKEAKOULU DIPLOMITYÖN TIIVISTELMÄ

Tekijä:	Miika Komu	
Työn nimi:	Host Identity Protocol sovellusrajapinnat	
Päivämäärä:	9. syyskuuta 2004	**Sivuja:** 86
Osasto:	Tietotekniikan osasto	
Professuuri:	T-110	
Työn valvoja:	FT Kimmo Raatikainen	
Työn ohjaajat:	TkT Pekka Nikander, DI Jukka Ylitalo	

Tämän diplomityön tavoitteena oli suunnitella ja toteuttaa sovellusrajapinta HIP-tietoisille sovelluksille käyttäen C-ohjelmointikieltä. Rajapinta on arvioitu asetettuja vaatimuksia vasten. Erilaisia toteutusmalleja on esitelty valitun toteutuksen perustelemiseksi. Mallitoteutus on kehitetty mallin realistisuuden toteamiseksi.

Rajapintamalli täyttää asetetut vaatimukset. Rajapinta on yhdenmukainen olemassaolevan ohjelmointirajapinnan kanssa ja laajentaa sitä ainoastaan silloin, kun muu ei ole mahdollista. Sovellusrajapinta lisää edistyneempien verkkosovelluksien hallintamahdollisuuksia HIP-tasoon. Sovellukset voivat myös määritellä oman identiteettinsä. Tavalliset sovellukset voivat hyödyntää rajapintaa yksinkertaisin keinoin puuttumatta identiteettien ja IP-osoitteiden yksityiskohtiin. HIP:ia tukeva sovellus voi siirtyä saumattomasti tavalliseen TCP/IP-kommunikointiin, jos vastapää ei tue HIP:ia.

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 ovat sidottu lujasti IP-osoitteisiin, kaipaavat myös lisätutkimusta.

Avainsanat: HIP, native, API, rajapinta, 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 the endpoint descriptor concept goes to Pekka Nikander as the idea originated from him, not me. Pekka gave also many other insightful comments. 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 the Ericsson's implementation too. Thomas Henderson gave some comments on the structure of the thesis in general, and comments on the design and future work chapters. I wish to thank Andrew McGregor for some fruitful discussion sessions on the API. Sasu Tarkoma gave some comments on the structure of the thesis. Antti Järvinen gave some typographical comments. Lars Eggert and Joe Touch gave some comments on the source locators in the HIP research group meeting at IETF60. Anthony Joseph corrected some spelling errors and gave some feedback.

Helsinki, 9th September 2004

Miika Komu

Contents

Abbreviations

AID	Application Identifier
API	Application Programming Interface
DSA	Digital Signature Algorithm
DHT	Distributed Hash Table
DNS	Domain Name System
DoS	Denial of Service
ED	Endpoint 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 Identifier
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
RR	Resource Record
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 a relatively trusted network environment that had a static structure. The nature of network has changed since then, but the protocol suite has basically remained the same during the last years. The network has become more insecure. The hosts are attached to the network in a dynamic manner.

A host attached to a network can distinguish itself from the other hosts in the network by its IP address and the IP addresses are used for routing packets to the hosts. Unfortunately, the same IP address is also reused at the transport layer. The drawback of this scheme is that the transport layer connections persist only as long as the underlying network layer IP addresses remain the same. As an analogy to everyday life, the situation would be similar if a person's name and home address were the same. When the person moved to a new appartment, also his name would change.

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

One objective of the new namespace is to decouple the transport layer identifiers from the network layer identifiers. In the new design, the function of the transport layer identifiers is to denote location independent connection endpoints. The network layer identifiers, aka the locators, denote the location of a host instead of the identity of an endpoint. They are used only in routing. The benefit of the decoupling is that transport layer connections may persist between hosts while the network layer locators change.

A new conceptual layer, the 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 authentication of the end-host identities and is the origin of binding update messages.

Depending on the API, the new transport layer identifiers are typically also present in the application layer. The identifiers can be presented to the application layer with varying degrees of visibility. For example, the most transparent level of visibility does not even allow the application to detect that it is using HIP. Consequently, it prevents the applications to control the HIP-layer behavior.

The main goal of this thesis is to design an Application Programming Interface (API) for HIP aware applications. The applications can utilize the HIP layer better using the API. In general, requirements have to be surveyed and selected in order to support the underlying basic functionality. In this thesis, a reference implementation of the API is also constructed.

1.1 Scope

The scope of this thesis is limited to designing an API for HIP aware applications. The focus is on the semantical issues of the API.

The reference implementation consists of a kernelspace socket handler, a userspace resolver library, and a telnet application, that was ported to use the API.

An explicit goal for the defined API was to support decoupling of transport and network layer namespaces by hiding the details of the network layer from the user of the API. The API should support basic networking related issues, such as resolving identities, binding sockets and network connections managing. The API is based on the sockets API [8] [40].

The following issues were explicitly left out the scope: Quality of Service (QoS) management, support for other types of APIs (e.g. X/Open Transport Interface (XTI)), support for other directories than DNS and solving the reversal DNS query problem (translating HITs to IP addresses).

Chapter 2

Background

In this section, we give a number of 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 skills in C programming. The focus of this chapter will be on the currently existing sockets API [40, 7] as the later chapters require detailed knowledge about the sockets API.

2.1 Mobility Related Terminology

An *endpoint* is defined as one participant of an end-to-end communication context, i.e. the fundamental agent of end-to-end communication. It is the entity which is performing a reliable communication on an end-end basis. [3] The *end-host* is a computational unit hosting a number of communicating processes [33].

End-host mobility refers to the phenomenon where an end-host changes its topological point of attachment while the communication context is kept alive [33]. *End-host multihoming* is similar to the end-host mobility, but the difference is that the host has multiple points of attachment to the network. The communication context can be moved "alive" from one attachment point of the host to another.

A mobile node is an IP node capable of changing its point of attachment to the network. Correspondent node is a peer with which a mobile node is communicating. A correspondent node may be either mobile or stationary. 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. *Horizontal handover* involves mobile nodes 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 moving between access points of different type, such as, UMTS to WLAN. [36, 22]

2.2 Host Identity Protocol

In this section we give a brief overview of HIP based on the currently available Internet drafts [27, 28, 32, 26, 4, 29, 43]. A number of HIP related publications [33, 31, 14, 2, 51, 18, 38] are also available.

2.2.1 Restrictions in the Current 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 relatively easy to steal [30].

In the current Internet model, the network layer IP addresses are reused at the transport layer. A Transport Layer Identifier (TLI) in the current Internet model is formed of a source IP address, a source port, a destination IP address, a destination port and a protocol. The drawback of the reuse of IP addresses at the transport layer is that the transport layer connections are still bound to the old IP addresses even 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.

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 for location independent identification of upper-layer endpoints. Consequently, the decoupling of the network-layer identifiers from the upper-layer 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 Identifier (HI) represents an endpoint in the HIP architecture. The HI is the public key from an asymmetric key pair. As the endpoint owns the private key of the key pair, it is rather straightforward for the endpoint to prove that it owns the HI. In other words, it is extremely difficult for other endpoints to claim ownership of the HI.

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 can be made one-to-many to support mobility and multihoming. The bindings are illustrated in Figure 2.1.

HIP architecture also includes fixed sized representations of the HI. A Host Identity Tag (HIT) is an 128-bit long hash 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. Hosts are strongly encouraged to have at least one public and one anonymous HI. The public HIs are usually stored in the Domain Name System (DNS)

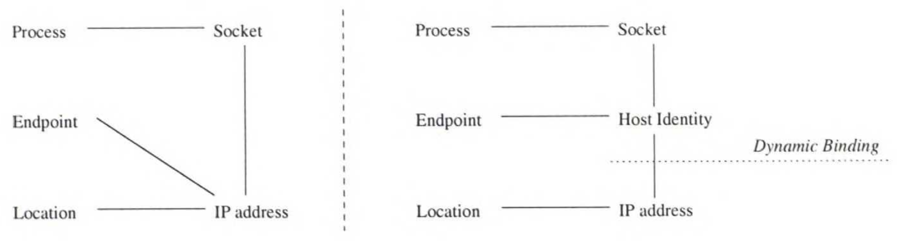

Figure 2.1: The current Internet binding model (left) and the HIP binding model [33]

or distributed using some other mechanism, such as a Public Key Infrastructure (PKI).

The Fully Qualified Domain Name (FQDN) is the search key for a host in the DNS. It is associated with a set of locators. The HI of the host may also be stored in the DNS in its various forms. The FQDN can be resolved both to the HI and the locators. Reverse resolving a locator to an FQDN is also possible but currently it is not possible to resolve a HI to a FQDN or a locator [1]. 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 namespace relationships are illustrated in Figure 2.2.

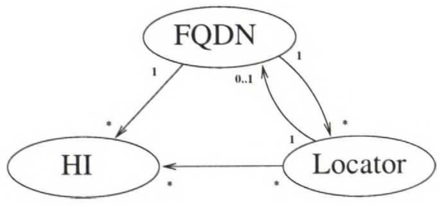

Figure 2.2: Namespace relationships

2.2.3 A New Layer

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

The base exchange resembles the IKE key exchange [12]. The base exchange implements an authenticated Diffie-Hellman key exchange. Two Security Associations (SAs), one in each direction, are constructed from the key material created by the base exchange. The SAs are then used for protecting the data traffic between

[1]Reverse resolving would possible using, e.g., Distributed Hash Table (DHT) mechanism, but such a mechanism is not globally deployed yet

hosts using Internet Protocol security (IPsec).

The base exchange also includes a "cookie" mechanism that protects the responder from certain types of Denial of Service (DoS) attacks. The initiator of the connection is forced to sacrifice some CPU cycles in order to to find a solution for a puzzle 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 resource exhausting DoS attack against the responder. The responder can also vary the difficulty level of the puzzles.

The base exchange can also be initiated without prior knowledge of the HI of the peer. This operation mode is called "opportunistic HIP". The operating 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 distributing HIs.

2.2.4 Mobility and Multihoming

Existing transport layer connections break if the locators of a mobile node are updated. To avoid this, the HIP layer also implements a mechanism for end-host mobility and multihoming. The mobile node informs its correspondent nodes directly when the locators of the mobile node are updated. The message used for informing the correspondent node may be signed with the public key of the mobile node to protect the integrity of the message. In addition, the correspondent node may also request the mobile node to verify the validity of the new set of locators. This *return routability test* makes the protocol more robust against certain types of redirection DoS attacks [32].

It is relatively straightforward for a mobile node to keep its correspondent node updated with its current set of locators assuming that the initial contact with the correspondent node has already been established. The mobile node usually establishes the initial contact by resolving the locators of the correspondent node from the DNS, along with the endpoint identifiers of the correspondent node. However, the DNS is fairly static and it may not be always up-to-date with the changes in the locators of the correspondent node because the correspondent node may also be mobile. The Secure Domain Name System (DNS) Dynamic Update [48] is a better alternative, but even it is hindered by DNS caching.

The HIP rendezvous server alleviates the problems related to the DNS. The rendezvous server is reachable by 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 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.

The rest of the HIP signaling messages are carried directly between the end-nodes to avoid triangular routing. The rendezvous server also solves the problem of double jump, i.e., both nodes changing their location at the same time.

The problem of the initial contact is now solved using the rendezvous server, which always knows 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 not change its location, both the mobile and corresponding node always know how to contact it. The rendezvous server resembles the home agent in the Mobile IP [35] architecture.

2.3 Related APIs

In this section, we give an overview the APIs that are most significantly related to this work. For other related APIs not discussed here, please see e.g. *LIN6 Multihoming API* [23], *Multihoming with Internet Protocol Version 6* [11], Generic Security Service (GSS) API C-bindings [21], *QoSockets: a New Extension to the Sockets API for End-to-End Application QoS Management* [5], *A Layered Naming Architecture for the Internet* [1], PF_KEY [24], NETLINK [37], Service Location Protocol (SLP) [10], and OpenSSL [47].

2.3.1 Sockets API

Sockets API is important from the view point of networking because it is used for all network communication. This section provides a brief introduction to the sockets API [8]. The discussion in this section is based on [40]. The emphasis is on the topics that are related to the native HIP API design, such as sockets API address structures, network interfaces, resolver and socket options.

Address Structures

The IP addresses are contained in the so called socket address structures before they are passed to sockets API functions. IPv4 specific addresses are encapsulated in `sockaddr_in` structures with the family set to `AF_INET`. Similarly, IPv6 addresses are stored in `sockaddr_in6` structures [7] 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 [8] 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 functions[2]. The first two fields, the length and the

[2]In ANSI C, a void pointer could be used instead of a `sockaddr` pointer. However, the sockets

```
/* 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;
}

/* 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 in 4.4BSD format. It is worth noting that the length and family fields are combined into a single field (`sin_family` or `sin6_family`) in 4.3BSD API [7], used, for example, in Linux.

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.

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

Figure 2.4: The generic socket address structure

The other abstraction structure, `sockaddr_storage`, is defined in [7]. The structure is sufficiently large to represent an address structure of any family. It simplifies writing of cross-platform and address independent applications.

API predates the ANSI C, and the void pointer is not used [40].

Sockets

The socket [49] is a very central concept in the sockets API. A socket denotes a Transport Layer Identifier (TLI) (half-association). A TLI is consists of an IP address and a port number. A socket pair denotes a TLI pair (full connection association). It consists of the source IP address, source port, destination IP address, destination port and protocol.

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` system function call is used for creating a socket in the API. It takes three arguments for setting the attributes for the socket to be created. The first argument sets the communication domain, such as `PF_INET` for IPv4 or `PF_INET6` for IPv6 enabled socket[3]. The second argument sets the communication semantics. For example, `SOCK_STREAM` is used for creating a sequenced, reliable, two-way, connection-based byte stream. A datagram oriented, connectionless and unreliable socket is created using `SOCK_DGRAM` constant. In practice, `SOCK_STREAM` means Transport Control Protocol (TCP) and `SOCK_DGRAM` means User Datagram Protocol (UDP) based communication. 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 is used in the subsequent sockets API function calls, such as `bind`, `accept`, etc.

Resolver

The *resolver* provides a name and address mapping service 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 `addrinfo` data structure are shown in Figure 2.6. The reverse functionality is provided by the `getnameinfo` interface.

The `getaddrinfo` function can be used for resolving both local and remote names. The first argument, `node`, denotes the name to be resolved. A `NULL` argument denotes the local host. The second argument, `service`, describes the name of the service

[3]The 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 [40] for the details.

```
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

port to be used for the endpoint. The service can be specified by name (e.g. `"http"`) or numerically (e.g. `"80"`). The third argument, `hints`, sets the attributes required from the endpoint.

The function writes 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 ib sockets 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 compherensive reference to the `getaddrinfo` function, can be found in [8, 40].

Interface Identification

Three network interface related functions are defined in [7]. 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.

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

```
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`.

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 successfully receives 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` call in the API. The application calls the `bind` function, which associates the socket with a given local IP address and port. It should be noted that the application can call the `bind` function only once for any given socket. The sockets are "disposable", because a 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 takes a socket descriptor, a socket address structure and the size of the structure in octets.

If the server application creates a socket of the `SOCK_STREAM` type, it needs to call API function `listen` to indicate its willingness to accept incoming connections to the port specified in the `bind` call. Later, the server application calls `accept`, which blocks until the connection is established. The `accept` call also returns the address of the client. The client initiates the handshake by calling the `connect` function, which typically blocks until the handshake is successfully completed. 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 the server application until another client is connected to the server port. The connection oriented function prototypes, used for, e.g. 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.

```
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

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 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.

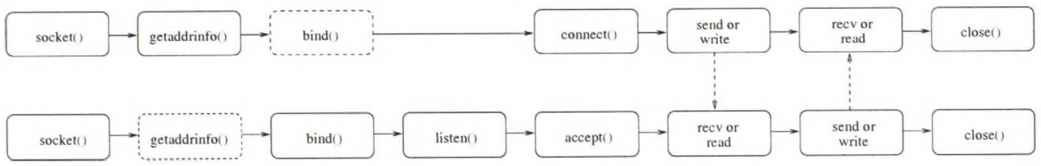

Figure 2.11: Typical client (above) and server (below) application interaction using TCP

UDP based socket communication requires fewer API function calls than in TCP, because UDP is not connection oriented and there is no need to establish a connection. 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 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 datagram oriented communication between the applications. The main difference to the connection oriented functions is that the destination IP address must be explicitly given each time when sending or receiving data. The datagram oriented 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 sock_addr *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 each create a socket. The server application calls `bind` to reserve a local port on the server host. Now, the client and server applications can communicate directly with each other using the `sendto`, `recvfrom`, `sendmsg` and `recvmsg` API calls.

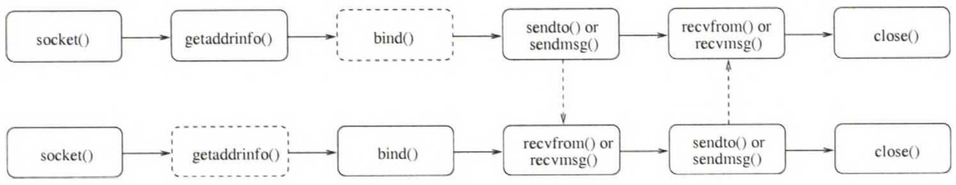

Figure 2.13: Typical client (above) and server (below) application interaction using UDP

It should be noticed that, both in the TCP and UDP examples, the client did not explicitly call `bind` to associate a local port and IP address on client host. If the client does not call explicitly `bind`, the networking stack takes care of automatically assigning an *ephemeral port* and an appropriate local address for the client application. In the TCP case, the automatic assignment occurs during the `connect` call. 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 ports on client host.

Socket Options

The socket options can be queried and set using the functions shown in Figure 2.14. The first argument is the socket. The second argument defines the socket handler for which the socket option is targeted. For example, it can be set to IPPROTO_TCP to access TCP related attributes for a socket. The third argument is the name of the option. The fourth argument is a pointer to the socket option object. The object can be a pointer to e.g. an integer or a data structure. The length of the object is indicated by the fifth 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 [42]. It provides many enhancements over TCP, such as support for multihomed hosts and multiple streams in a single SCTP association [42]. These and many other enhancements make SCTP superior to TCP for real-time multimedia and telephony applications.

The SCTP API [41] 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, this does not allow applications to 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 stream oriented like TCP.

Applications can benefit from the new SCTP features by using an UDP-style interface. The UDP-style interface can be utilized with the `sendmsg` and `recvmsg` API calls, which were briefly shown in section 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 [41].

The datagram-oriented 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; a separate event notification interface is unnecessary. To receive notifications, the application must first set the right socket option for each specific event type. The application then sends or receives data by calling the `sendmsg` and `recvmsg` functions, as normally. The event information can be formed from the `msg_hdr` argument when the function call returns. The difference between normal application data and event data is indicated with a special flag in the `msghdr` argument.

2.3.3 Legacy HIP API

The legacy HIP API refers to the initial implementation specific HIP APIs, such as in [45] and especially [2]. The main motivation for the legacy API was to enable HIP transparently by minimizing the number of changes in the application code.

In this section, we will limit the discussion to the the legacy API used in [2]. 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 the so called *transparent HIP 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 HIP enabled host [4]. The client application is assumed to try the socket addresses in the order received from the resolver. As the first socket address is actually a HIT, HIP enabled connections are preferred 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 host. The fall back to the IPv6 addresses is useful during the transition phase to HIP when all network hosts are not capable of supporting 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 very flexible, 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 explicitly use HIP by passing the `AI_HIP` flag to the resolver. The resolver returns HITs only if the flag is set. In fact, it returns *only* HITs and no IP addresses at all, 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.

The source code is not always included with all legacy applications. In such a case, the transparent API may be the only choice to make the application use HIP. Still, it is required that the resolver library is not statically compiled into the application because otherwise the HIP enabled library cannot used at all.

[4]The legacy API does not return HITs when resolving local host 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` constant.

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 listed in [7] and [39].

The requirements are organized into functional and non-functional groups. 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 used 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, management of the HI namespace and the 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. The basic use of the native HIP API should be as simple as, or even simpler than, the typical use of the current 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 develop applications using 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 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 in 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 also require 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 for 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.

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 typical applications to guarantee compatibility with the future changes in the protocol identifiers.

Other Compatibility Issues

Our kernel oriented HIP implementation, HIP for Linux (HIPL) [16], 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 [15] [25] [13] [19] 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) [8], conformance to GSS API [20, 21] and supporting Service Location Protocol (SLP) will not be evaluated. Some multi-homing protocols like SCTP [17] and Location Independent Networking for IPv6 (LIN6) [44] define their own userspace APIs [41, 23]. 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 name resolution mechanisms. Mobility, multihoming and security control requirements are also discussed.

3.2.1 Host Identities

Most network applications do not need to have access to the actual binary the representation of the HI. Basically, a reference to the actual HI should be enough. However, there might be some specialized applications that have to deal directly with the identities. We have to find a balance between the typical and advanced network applications. 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 the typical applications. This transparency may also turn out to be useful in other network protocols based on the identity-locator split; evaluating the usefulness is out of scope.

Conversely, we assume that there are applications that 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 the wrong impression of the ownership of identities. As the name implies, one could imagine that only the host has some preassigned HIs. According to the specifications [28], the host must have at least one public and one anonymous HI but it is not prohibited from having 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 [50].

At first glance, 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 originating 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 smart card and dynamically assigned to the device the employee happens to be using at the moment.

The application specified identifiers are essentially public-private key pairs which tend to be relatively long. This bring an additional concern of resource consumption. It is probable that a process wants to share its identifier among a set of other processes, such as all the processes sharing the same Group ID (GID). In such a case, the underlying system should be intelligent enough to avoid replicating the shared identifier for each process. Instead, some form of referencing mechanism is encouraged.

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 would not be possible to select between multiple host supplied HIs or input 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 introduced for the convenience of the developer. An interface for creating an application specified HI has to be defines. Also, interfaces need to be defined for loading a HI from a file and saving a HI to a file.

3.2.2 Addresses and Interfaces

The network interfaces, and especially the IP addresses, are considered ephemeral from the HIP point of view. It is error prone to handle them explicitly in the API as they can change undeterministically. The details of the network layer entities should not be exposed to a typical application. On the other hand, access to the network layer should not be completely prohibited for applications that need to access the network layer details. Thus, one goal is to define a separate API 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 any networking layer related choices on its behalf. The HIP implementation transparently selects the interfaces and addresses used for a connection.

The requirements for explicit locator selection are mostly related to the manual selection of addresses and interfaces. First, the API should allow entering initial peer addresses manually as the network environment may not have a directory service. Second, it should be possible to select which of the local network interfaces are to be used for network communication. The number of manually selectable addresses and interfaces should not be restricted. Limiting the scope of addresses within the selected interfaces should also be possible.

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

The API should support HIP in opportunistic mode, i.e., without 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 hosts dynamically updates the locators transparently from applications. However, the applications should be able to configure the initial locators manually. After the locators are configured, a typical application cannot observe anymore the handovers. The API could contain an interface for tracking locator updates; designing one is out of the scope of this work.

QoS, load balancing, and other similar complex policy issues are also out of scope. On the other hand, the high level design should still be simple and modular enough so that those features can later be added into the API without completely redesigning 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 such new features. The native API should allow inspecting and modifying of some of the HIP related security attributes to suit specific application needs. The applications should be allowed to negotiate HIP related security attributes, such as the encryption level, crypto algorithms being used for network connections, and the selection of the puzzle difficulty level. Enabling of the opportunistic mode and falling back to plain TCP/IP 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 in order to avoid introducing security flaws into the design:

1. Confidentiality: what the processes or objects are not allowed to see.
2. Integrity: what any given process and other processes are not allowed to change.
3. Capability: the process should not be able to consume all the limited resources of the underlying host.

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 puzzle difficulty of the other processes. An example of the capability constraint is an application that is denied when it tries 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. A few HIP specific error values might be needed.

3.2.6 Name Resolution

The native HIP API resolves the peer identities and locators from a DNS directory. The native HIP API design is required to support only DNS and the UNIX `/etc/hosts` file. Supporting other kind of name resolution services, such as those based on DHTs, are out of scope. Supporting DNS Service Record (SRV) [9] is also out of the scope.

3.3 Evaluation

The main objective of the implementation is to prove that the basic design can be made to work in practice. The performance, reliability, and actual implementation level of security are secondary 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 are 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 point of view. We introduce the Endpoint Descriptor (ED) concept, which is a central idea in the API. We describe the layering and namespace models along with the socket bindings. We conclude the discussion with a description of the endpoint identifier resolution mechanism.

4.1.1 Endpoint Descriptor

The representation of endpoints is hidden from the applications, as required in section 3.2.1. The Endpoint Descriptor (ED) is a "handle" to a HI. A given ED serves as a pointer to the corresponding HI entry in the HI database of the host. The ED is the Application Identifier (AID) [34] in the native HIP API model.

4.1.2 Layering Model

The application layer accesses the transport layer via the socket interface. 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. For simplicity, the IPsec layer has been excluded from the figure.

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 see if the datagrams are HIP related and require HIP intervention.

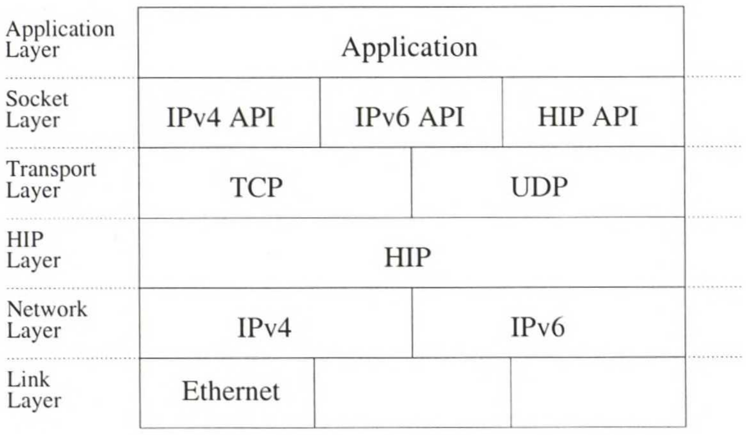

Figure 4.1: The layering model

4.1.3 Namespace Model

The used namespace model is shown in Figure 4.2. The namespace identifiers are described in this section.

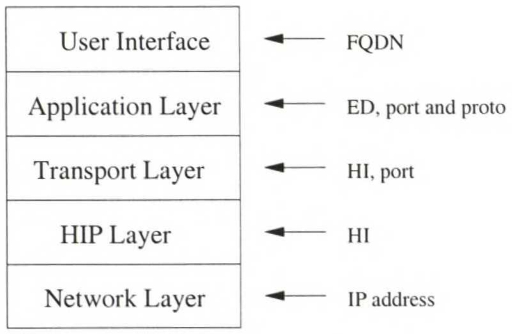

Figure 4.2: The ED centric namespace model

People prefer human-readable names when referring to network entities. 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 level identifier in the context of the native HIP API.

In the current 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 Descriptors (EDs) instead of HITs. Now, the application layer uses source ED, destination ED, source port, destination port, and transport protocol type, to distinguish between the different connection associations[1].

Basically, the difference between the application and transport layer identifiers is that the transport layer uses HIs instead of EDs. 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.

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

4.1.4 Socket Bindings

A HIP socket is associated with one source and one destination ED, along with their port numbers and protocol type. The relationship between a socket and ED is a many-to-one one. Multiple EDs can be associated with a single HI. Further, the source HI is associated with a set of network interfaces at the local host. The destination HI, in turn, is associated with a set of destination addresses of the peer. The socket bindings are visualized in Figure 4.3.

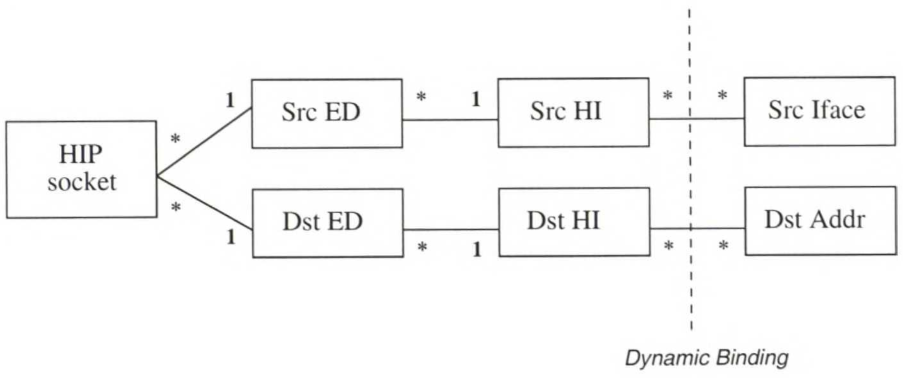

Figure 4.3: Socket bindings

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

[1]See section 6.2.4 for restrictions in the binding to EDs

The ED 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 section 3.1.2.

4.1.5 Endpoint Discovery

The DNS based endpoint discovery mechanism is illustrated in Figure 4.4. The application calls the resolver (step a.) to resolve an FQDN (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 (step d.). Finally, the resolver receives an ED from the HIP module (step e.) and passes the ED to the application (step f.).

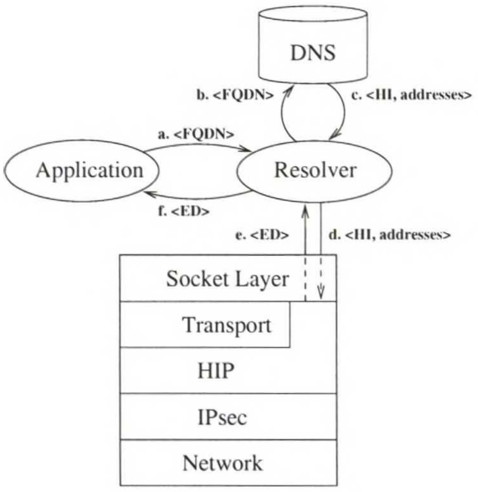

Figure 4.4: The path of resolving of an FQDN to an ED

The application can also receive multiple EDs from the resolver if the FQDN is associated with 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 EDs in return, each ED corresponding to a single HI. Finally, the EDs are sent to the application.

4.2 Interface Syntax and Description

In this section, we describe the native HIP API using the syntax of the 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 [8, 40].

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

```
struct sockaddr_ed {
        unsigned short int ed_family;
        in_port_t ed_port;
        sa_ed_t ed_val;
}
```

Figure 4.5: The ED is contained in a sockaddr_ed structure. The structure is shown in 4.3BSD format [7].

The ed_val field is usually set by special native HIP API functions, which are described in the following section. However, three special macros can be used to directly set a value into the ed_val field. The macros are HIP_HI_ANY, HIP_HI_ANY_PUB and HIP_HI_ANY_ANON. They denote an ED value associated with a wildcard HI of any, public, or anonymous type. This is useful to a "server" application that is willing to accept connections to 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 new 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 with the getaddrinfo resolver. The family, ei_family, is set to PF_HIP when the

```
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

socket address structure contains an ED that refers to a Host Identifier (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 for local anonymous identifiers. The default action is first to resolve the public endpoints and then the anonymous endpoints.

Some applications may prefer configuring the locators manually and can set the `EI_NOLOCATORS` to prohibit the resolver from resolving any locators. 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 suggests the resolver to return 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 cause the resolver to output only a single socket address containing an ED that would be received using the corresponding `HIP_HI_*ANY` macro.

Application specified endpoint identifiers are essentially private keys. To support application specified identifiers in the API, we need new data structures for storing the private keys. The private keys need an uniform format so that they can be easily used in API calls. The keys are stored in the endpoint structures shown in figure 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 [28]. 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.

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

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

Figure 4.7: The endpoint data structures

An optional extension to the getaddrinfo interface is introduced too. A new flag, AI_HIP_RVS, is used both in the input and output of the resolver. By default, the getaddrinfo resolver does not return IP addresses belonging to a HIP rendezvous server. The resolver returns rendezvous server addresses only when the AI_HIP_RVS flag is set in the resolver hints. This way, legacy applications can never receive any addresses belonging to a rendezvous server. The flag is also set in the getaddrinfo resolver output to denote that the resolved address belongs to a HIP rendezvous server.

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 calls the functions with sockaddr_ed structures instead of sockaddr_in or sockaddr_in6 structures. The source of the sockaddr_ed structures in the native HIP API is the resolver function getendpointinfo which is shown in figure 4.8.

The getendpointinfo function takes the nodename, servname, and hints as its input 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

```
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

argument specifies the host name to be resolved; a `NULL` argument denotes the local host. 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 zero, any kind of endpoints are acceptable.

Application Specified Identities

Application specified local and peer endpoints can be retrieved from files using the function shown in Figure 4.9. The function `hip_endpoint_load_pem` is used for retrieving a private or public key from a given file `filename`. The file must be in PEM encoded format [47]. 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 hip_endpoint_pem_load(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 directly in the sockets API function calls. The application must convert the endpoint into an ED first. Local endpoints are converted with the `getlocaled` function and peer endpoints with `getpeered` function. The functions are illustrated in Figure 4.10. Both of functions are used in a similar way.

The result of the conversion, an ED socket address, is returned by the functions. A failure in the conversion causes a `NULL` return value to be returned and the `errno` to

```
struct sockaddr_ed *getlocaled(const struct endpoint *endpoint,
                               const char *servname,
                               const struct addrinfo *addrs,
                               const struct if_nameindex *ifaces,
                               int flags)
struct sockaddr_ed *getpeered(const struct endpoint *endpoint,
                              const char *servname,
                              const struct addrinfo *addrs,
                              int flags)
```

Figure 4.10: The functions for converting application defined endpoints to ED structures.

be set accordingly. The caller of the functions is responsible of freeing the returned socket address structure.

The endpoint argument is retrieved e.g. with the `hip_endpoint_load_pem` function. If the `endpoint` is `NULL`, an arbitrary HI of the host is selected and associated with the ED value of the third argument.

The `servname` argument is the service string. The function converts it to a numeric port number and fills the port number into the returned ED socket structure for the convenience of the application.

The `addrs` argument defines the initial IP addresses of the local host or peer host. The argument is 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` [7] function call. A `NULL` pointer indicates that the application trusts the host to already know the locators of the peer. We recommend that a `NULL` pointer is not given to the `getpeered` function to ensure reachability with the peer.

The `getlocaled` function accepts also a list of network interface indexes in the `ifaces` argument. The list can be obtained with the `if_nameindex` [7] function call. A `NULL` list pointer indicates all the interfaces of the local host. Both the IP addresses and interfaces can be combined to select a specific address from a specific interface.

The last argument is the flags. The following flags are valid only for the `getlocaled` function:

- `HIP_ED_*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 binding to be reused for processes with the same User ID (UID), GID or any UID as the calling process.
- Flags `HIP_ED_IP` and `HIP_ED_IPV6` are used for limiting the address family

scope of the interfaces.

It should noticed that the HIP_HI_ANY, HIP_HI_ANY_PUB and HIP_HI_ANY_ANON macros can be defined as calls to the getlocaled call with a NULL endpoint, NULL interface, NULL address argument and the flag corresponding to the macro name set.

Querying Endpoint Related Information

The getlocaled and getpeered functions have also their reverse counterparts. Given an ED, the getlocaledinfo and getpeeredinfo functions search for the HI and the current set of locators associated with the ED. The first argument is the ED to be searched for. The functions write the results of the search, the HIs and locators, to the rest of the function arguments. The function interfaces are depicted in figure Figure 4.11. The caller of the functions is responsible for freeing the memory reserved for the search results.

```
int getlocaledinfo(const struct sockaddr_ed *my_ed,
                 struct endpoint **endpoint,
                 struct addrinfo **addrs,
                 struct if_nameindex **ifaces)
int getpeeredinfo(const struct sockaddr_ed *peer_ed,
                  struct endpoint **endpoint,
                  struct addrinfo **addrs)
```

Figure 4.11: The functions for querying ED information

The getlocaledinfo and getpeeredinfo functions are especially useful for an advanced application that receives multiple EDs from the resolver. The advanced application can query the properties of the EDs using getlocaledinfo and getpeeredinfo functions and select the ED that matches the desired properties.

Socket Options

As usually, getting and setting of HIP socket options is done using getsockopt and setsockopt functions. To set HIP layer specific socket options, the first argument must be a socket descriptor that was instantiated with PF_HIP as the domain, and the second argument must be specified as IPPROTO_HIP [2].

Some HIP socket option names are listed in Table 4.1. The length of the option must be natural word size of the underlying processor, typically 32 or 64 bits. The purpose of the option value must be interpreted in context of the protocol specifications [28, 32].

[2]Naturally, the HIP enabled socket does not constrain access to the socket options of other layers. The second argument can be, e.g., IPPROTO_IP or IPPROTO_TCP

Socket option	**Purpose**
SO_HIP_CHALLENGE_SIZE	Puzzle challenge size
SO_HIP_HIP_TRANSFORMS	Integer array of the preferred HIP transforms
SO_HIP_ESP_TRANSFORMS	Integer array of the preferred ESP transforms
SO_HIP_DH_GROUP_IDS	Integer array of the preferred Diffie-Hellman group IDs
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 IP 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	Try normal base exchange, but fall back to plain TCP/IP if the base exchange fails.

Table 4.1: HIP socket options

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 associated with the given socket.

Chapter 5

Implementation

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

5.1 Userspace Components

The userspace component of the native HIP API, i.e., the resolver library, is linked either statically or dynamically to applications. The implementation of the resolver is based on the *libinet6* library [46].

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

The `getendpointinfo` resolver internally uses the `endpoint_hip` data structure that was presented in section 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 peer identifiers. The HI member is used when resolving the HIs of the local host.

5.2 Kernelspace Components

The kernelspace 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 kernelspace component, the kernel module, is divided into two conceptual subcomponents. 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 with the `PF_HIP` socket family. The socket handler is a "wrapper" to the IPv6 socket handler because its main purpose is to translate EDs to HITs.

The rest of the Linux networking stack cannot be easily divided 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 networking stack and HIP related hooks. 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 is equipped with "hooks" both in the output and input side. When the endpoints are HIs, the hooks bypass normal networking stack control flow and transfer the control to the HIP module.

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

The layer below the transport layer involves both routing and IPsec handling. At this layer, the hooks are responsible of the conversion of the source and destination HITs into IPv6 addresses (`hip_handle_output`, `hip_get_saddr` and `hip_get_addr`). The base exchange is triggered by the `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 would be appealing to convert the EDs 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 pseudo header 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 EDs straight to IPv6 addresses, we would need separate hooks for the native HIP API.

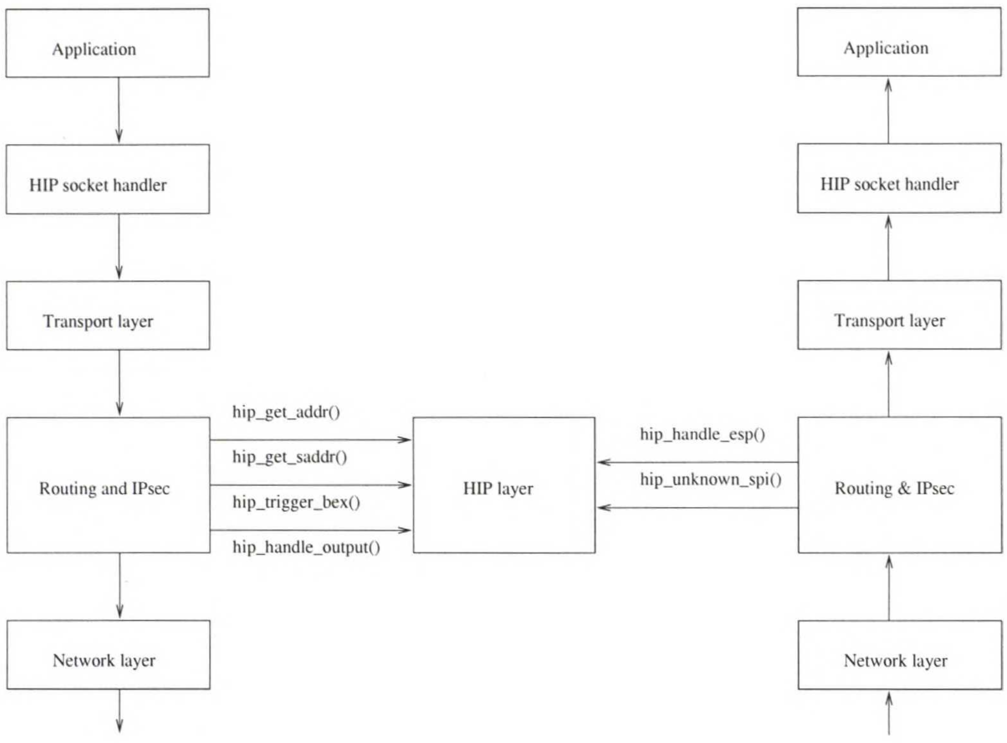

Figure 5.1: The output and input interfaces of the networking stack

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 various socket handlers in the kernel.

The HIP socket handler has two data structures that are used for storing local and peer ED values. The ED value is associated with the ownership information of the ED entry. It guarantees the confidentiality and integrity of the ED related information. The ownership information consists 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

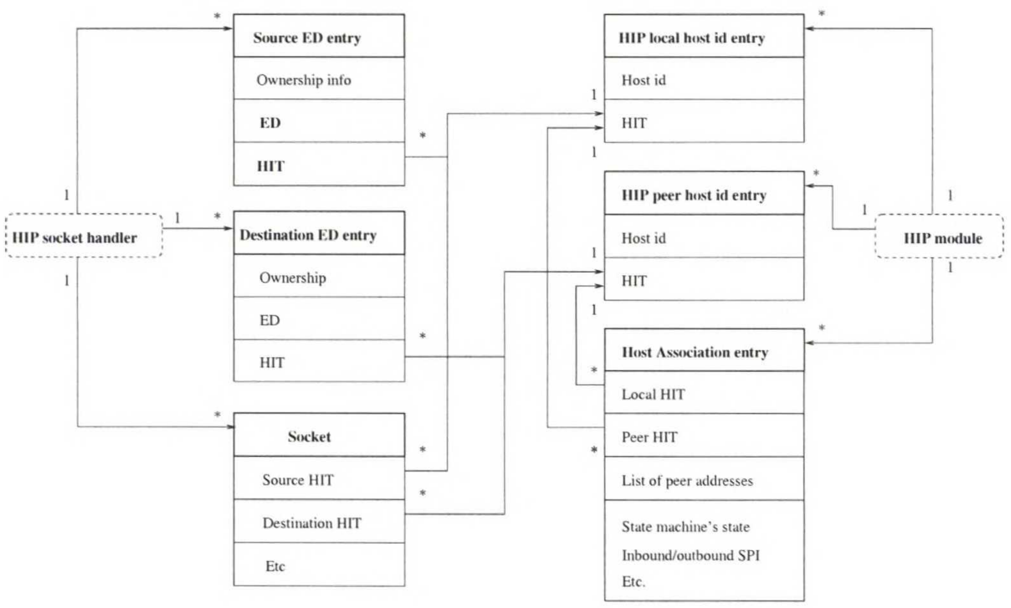

Figure 5.2: HIP kernel module data structures and their relationships

socket handler ED 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 ED data structures could have been merged into one structure if the legacy API compatibility restriction was not present.

5.5 Interaction between the Components

The interaction between the components is illustrated using sequence diagrams. The sequence diagrams show the control flow through different function calls from the userspace to kernelspace. We do not show the full execution trace but instead focus on the most relevant functions. The reader should not become confused by the 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 ED 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 and, for simplicity, the sockets are assumed to be blocking.

5.5.1 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 the `hip_create_socket` 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` resolver. The resolver queries the necessary HIs and locators and calls `setlocaled` to return a local ED corresponding to the HI and locator information. The `setlocaled` call sends the information to the HIP module. The HIP module generates a local ED, stores it to the local ED data structure described in the HIP module and returns the ED to the resolver. The resolver returns the ED to the application.

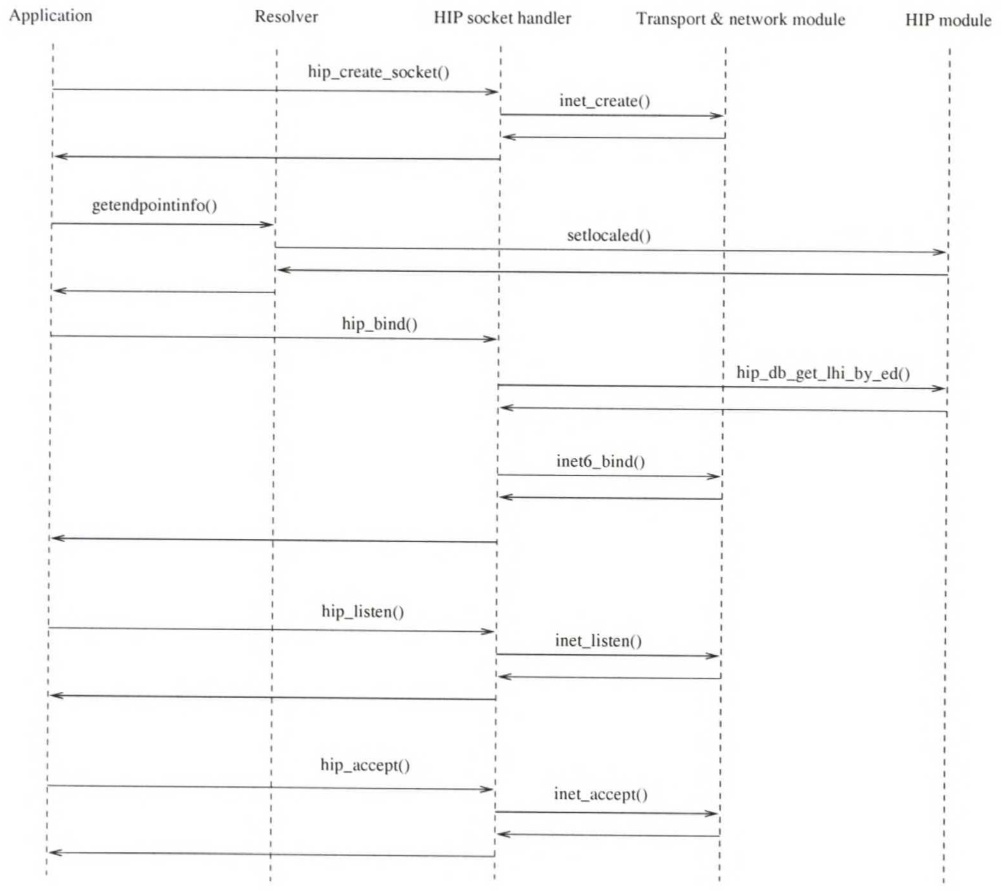

Figure 5.3: Bind sequence diagram

The server application is now ready to bind to the socket using the ED and the port number. The `bind` call eventually translates to a `hip_bind` call in the HIP socket hander. The socket handler maps the ED to a local HIT using `hip_db_get_lhi_by_ed` 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 execution is resumed.

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 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 ED for the peer, stores it into the peer ED data structure, and returns the ED.

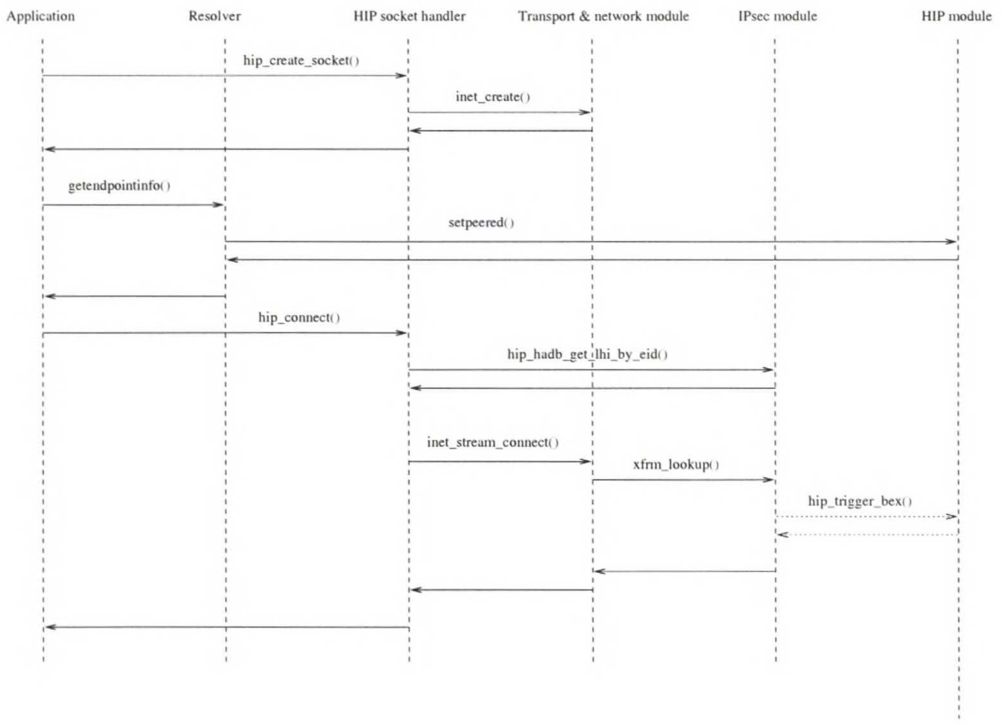

Figure 5.4: Connect sequence diagram

The client initiates the TCP connection using `connect`, which eventually calls `hip_connect` in the HIP socket handler. The socket handler maps the peer ED value to the peer HIT with `hip_hadb_get_lhi_by_ed`. The handler stores the HIT into the `socket` structure. The socket handler calls the `inet_stream_connect` in the transport/network module to initiate TCP handshake. The transport/network module generates 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 ED is bound to a default HI of the host and the corresponding locator set is assigned to any interface available on the host.

The server application was being blocked by the `accept` call. It continues execution and it can now receive data from the client application.

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 ED to HIT conversion here as the corresponding `socket` structure has already been configured to use the peer HIT as the destination IPv6 address.

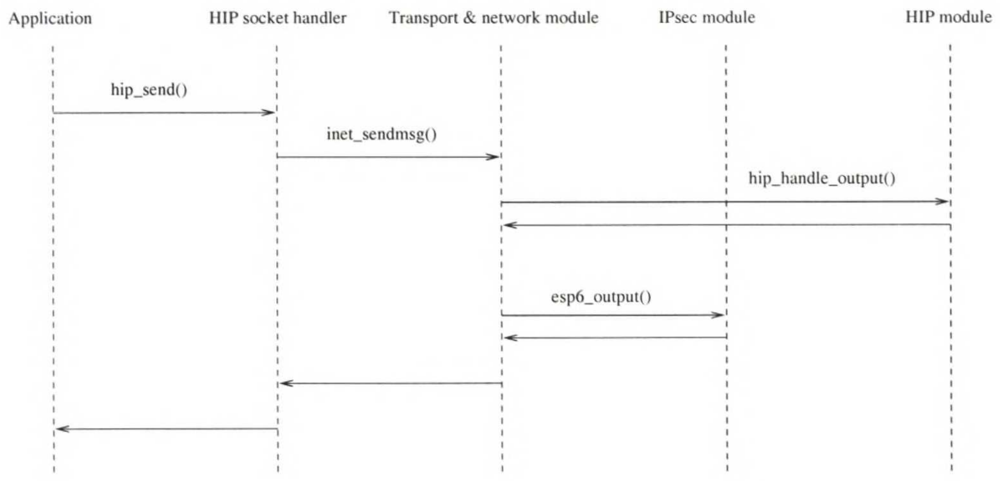

Figure 5.5: Send diagram

The socket handler calls `inet_sendmsg` in the transport/network module to transmit the data into the network. The global SP takes care of encapsulating the data into ESP envelopes. Before the the packet to the network, `hip_handle_output` hook intercepts the packet and notices that the destination address is a HIT, instead of an IPv6 address. The hook replaces the HIT with an IPv6 addresses. Finally, the hook returns control of the flow to the transport/network module that transmits the ESP encapsulated packet to the network [1].

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.

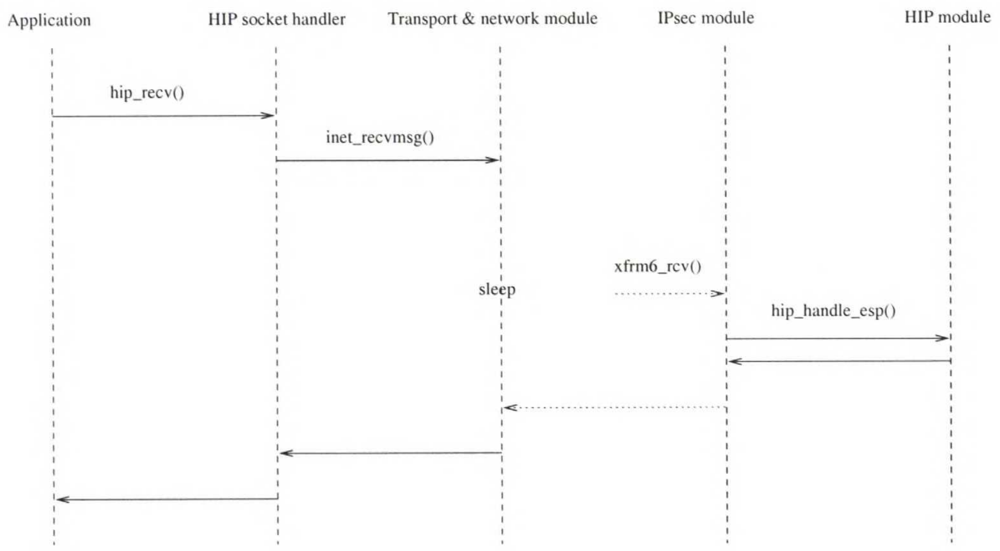

Figure 5.6: Recv diagram

5.6 HIP Enabled Telnet

An IPv6 enabled telnet client and a telnet daemon from [46] were ported to use the native HIP API. The native HIP API was configured as a compile time option into the code.

[1]There is a minor caveat related to the global SP that matches to any HIT. The `hip_handle_-output` is called first to replace the source and destination HITs with IPv6 addresses. The `esp_-output` is called after that to check for a match on the global SP based on HITs. It works because the decision to use ESP is made before calling the `hip_handle_output` function

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 native HIP API based test applications can be found from Appendix A. They are similar to the client and server applications used in the example scenario in this chapter.

Chapter 6

Analysis

In this section, we analyze the design of the native HIP API on a conceptual level. The analysis includes evaluation of the requirements against the design and descriptions 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 in unexpected behavior. This was recorded as a backwards compatibility requirement in section 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 local host is detected when trying to create a HIP socket or when resolving local host HIs. Basically, the HIP capability of the peer is detected when the resolver is called, unless the peer does not have its HIs in the DNS for some reason. For example, it is important that a HIP aware application can detect that it is running on a non-HIP host. This way, the application can make a clear exit or fall back to plain TCP/IP.

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 [40] 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 Descriptor

The ED represents the endpoint for an application. The ED can be associated with both fixed sized identifiers (HITs) and variable sized identifiers (HIs), as stated in the representation requirements in section 3.2.1. Further, the identifiers are dynamically associated with 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 ED and a socket is many-to-one. This way, it is possible to reuse the same ED for multiple sockets. For example, consider a web browser that opens multiple sockets and lets the user specify his own HI. The `getlocaled` call needs to be called only once because the same ED can reused for multiple sockets.

The need for different representations of HIs, such as the HIT, is diminished in the application layer because the ED replaces them in most cases. Consequently, the current 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 ED value is stored in the `sockaddr_ed` 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 ED value is an integer similar to the socket descriptor, there is a chance for the novice application developer to mistake it for a socket descriptor. It is safer to keep the ED value inside the socket address structure.

One might argue that an ED socket address structure is purely related to HIP layer and therefore should not contain a port number. However, the 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 `getlocaled` or `getpeered` function to the HIP module. The problem is that the HIP module cannot unambiguously associate the port number along with the associated ED value to the corresponding socket. This problem could be solved by adding the socket descriptor to the arguments of the ED setting function. This would require the socket descriptor argument to be added also to the resolver, because the resolver uses the same ED 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 `ed_val` field in the `sockaddr_ed` structure, thus requiring no `getlocaled` call. If the macros were defined as constant integers, it would increase the complexity of the HIP socket handler, because it would have to handle ED values that are global to all applications. The complexity was avoided by defining the macros as calls to the `getlocaled` call with NULL arguments and with the appropriate flag set. In this way, the special handling for the constant ED value is not needed at the kernel side.

It should be noted that there is a minor benefit from the exclusion of the socket descriptor in the declarations of the `getlocaled` and `getpeered` 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 ED. As consequence, the resolver call is also independent of the socket instantion even though the resolver uses the `getlocaled` and `getpeered` functions internally.

6.1.3 Application Specified Identifiers

The requirement for the application specified identifiers is met both in the design and implementation. The `hip_endpoint_load_pem` function can be used to create an application specified endpoint with the help of the `getlocaled` or `getpeered` functions. The application specified identifier scheme was tested with the test application described in Appendix A.

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 returns a set of EDs, which the application can pass directly to the socket functions, such as `bind` and `connect`. Still, the real representation of the HIs and locators behind the ED can be revealed with the `getlocaledinfo` or `getpeeredinfo` functions, when needed.

The interfaces are preferred for the local host case instead of IP 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. Nevertheless, IP addresses must be supported too in the local host case. Otherwise unambiguous selection of a specific IP address on a specific interface is not possible.

The interface of the native HIP API resolver function along with its companion data structure, `getendpointinfo` and `endpointinfo`, closely resemble 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 have been 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 current and the native HIP API resolvers, and separated 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 current resolver API.

6.2 Design Alternatives

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

6.2.1 An 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 endpoint identifiers. 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.

Such transparency would be 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" into using an IP address even though it will actually 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 forward compatibility. 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 ED approach has a few cons when compared to the HIT and HI approaches. The ED approach does preserve forward compatibility as opposed to the HIT approach. ED is 100 % collision free, but HIT is not. The size of an ED is constant and small, so it can be used more effectively than a full HI in the sockets API.

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

6.2.3 A Shared Data Structure for Identifiers and Locators

The ED 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 ED is an opaque handle to the HI and associated locators. The ED 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 ED base model where there is no need for the ED concept. In this model, the information referenced by the ED, the HI and the locators, are stored directly in a socket address structure as shown in Figure 6.1. The model is similar to [23].

```
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 ED socket address structure

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 `getlocaled` or `getpeered`, before the socket structure can be used. The resolver just returns `sockaddr_hip` structures but it does not need to call any mapping function. Only when the application specifies its own identifier, does 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 too large to fit directly into the socket address. 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 easily reflected to the application. The locators are not necessary valid as they will be never updated. The locator union may also be more difficult to handle from a usability point of view. Having a variable, albeit limited, number of locators is unnatural for a socket address structure in the sockets API, as there 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 An Endpoint 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 EDs. The TLI pair would then consists of source ED, source port, destination ED and destination port. This extension would allow a more elaborate kind of binding model, which is illustrated in the example connection association below:

$$\begin{aligned} 1 &: \{ED_A^{src}, PORT_A^{src}, ED_B^{dst}, PORT_B^{dst}\} \\ 2 &: \{ED_C^{src}, PORT_A^{src}, ED_B^{dst}, PORT_B^{dst}\} \end{aligned}$$

The associations are within a single host. The source EDs, ED_A^{src} and ED_C^{src}, are distinct EDs, but are associated with the same host identity HI_A^{src}. The destination ED, ED_B^{dst}, is associated with 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 EDs 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 ED unambiguously because the HI can be mapped to multiple EDs.

6.2.5 An Alternative Resolver Model

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

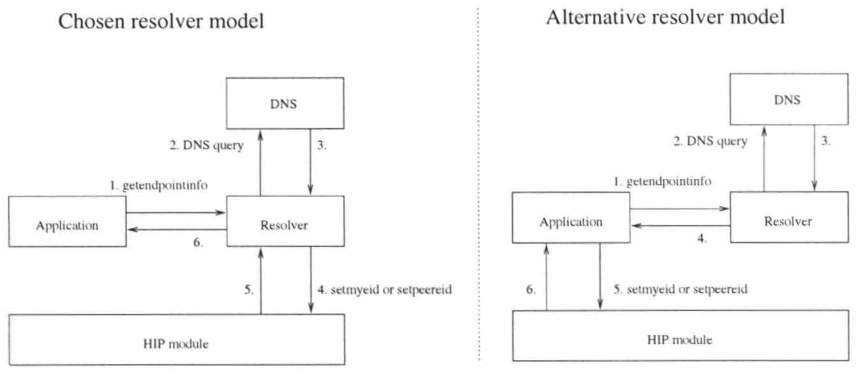

Figure 6.2: The resolver is responsible for the mapping the HI and locators to an ED in the chosen resolver model. In the alternative model, the responsibility is delegated to the application.

In this alternative model, the application must always call the `getlocaled` and `getpeered` functions. To achieve this, the resolver always explicitly returns the HIs and locators to the application. The application selects a HI and the associated locators and passes them to the `getlocaled` or `getpeered` function to receive the corresponding ED. 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 ED. 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 always requires the `getlocaled` or `getpeered`, 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 Descriptor

A couple of ED related functions are needed to make the API more complete. Comparison of ED values needs a new system function, because the design does not give any guarantees about the numerical properties of the ED values. Passing an ED to another process requires a new system function, as well the duplication of an ED within the same process. If the ED was a real file descriptor, the duplication could be implemented with the existing `dup` [40] 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 ED and HI is dynamic instead of static, i.e., the relationship between ED and HI becomes many-to-many over time. The application has a stable ED but the identifier associated with the ED is allowed to change transparently from the application. The benefit of such a HI mobility feature is questionable, but the ED concept would ease the development such as a feature as it adds a layer of indirection.

The ED concept could also be useful to other mobility related protocols, especially to those based on the 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 ED concept might be generic enough to be used in other protocols too, but this requires further analysis and experimentation.

7.1.2 Host Identifiers

The definitions of the `endpoint_hip` structure and `hip_endpoint_load_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 Digital Signature Algorithm (DSA) keys, but other types of keys were not considered because the implementation supports only DSA. Loading of public or private keys could be supported in other formats than Privacy Enhanced Mail (PEM), such as "SSH Public Key File Format" [6].

7.1.3 Locators

A FQDN of a host is associated with 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.

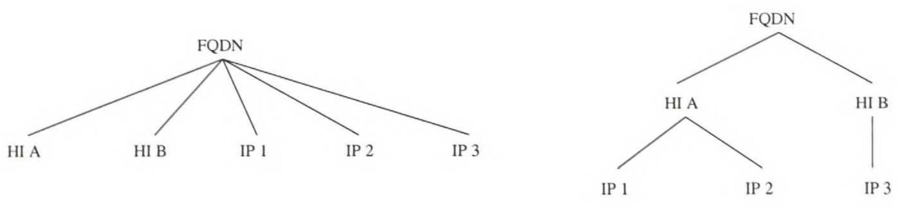

Figure 7.1: In the current DNS RR model (left), the FQDN is associated with a set of HIs and locators, and it is not possible attach a certain HI to specific locators. The right side of the figure illustrates the case where this restraint would not exist.

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 ED cannot be used directly as a referral, and it is not suitable for passing to an FTP application. If an FTP application were to use the native HIP API, it would have to query the host for the HIs and locators corresponding to the ED, and use them for a referral. This area of applications still needs further work.

7.1.5 HIP Proxy, Rendezvous Server and NAT

[4] specifies a proxy model for communication between HIP-hosts and non-HIP hosts. In some scenarios, it is better to use this kind of proxy server. In other scenarios, the option of falling back to plain TCP/IP networking is more appealing.

The fall back option may also be desirable during the transition period to HIP. During that time, all NAT boxes may not be HIP aware. A host may end up behind such an NAT box, and falling back to plain TCP/IP is necessary in order to contact other nodes outside the NAT.

It may 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 `getpeered` function. The analysis of this scheme is not complete, but some thoughts are given below.

If the base exchange would be triggered from the `getpeered` 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 `getpeered` when resolving peer HIs.

There may be some benefit from triggering the base exchange before the application sends any data to the network. For example, the application could trigger base exchange the before it is going to send data to the peer application. This would save some round trips when the application is ready to send data to the peer. This feature could be, however, implemented by the application itself by sending dummy data to the peer host.

Evaluation of the compatibility with other APIs, such as LIN6 [23] and SCTP [41], 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 [41] 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 a 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 [41]. 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" [42] 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, similar to 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 EDs.

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 based on e.g. PF_KEY [24] or NETLINK [37]. Alternatively, each implementation could have a separate native HIP API library that needs to 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 ED data structures described in section 5.4. Some functionality could have been implemented in an easier way, if the related data structures were store in the process context. For example, if the key material passed to the `getlocaled` function is probably easier to deallocate in the HIP module when a process exits. However, as much as possible, we tried to avoid modifying the existing networking stack, and all of the data structures were implemented in the HIP module.

Some requirements were not implemented. The ownership permission checking of EDs was implemented only partially. The HIP module discards the interfaces for local EDs. 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 local host resolving and HITs for peer resolving. Interface selection by specifying the address family was not implemented as the implementation supported only IPv6 addresses. The `getlocaledinfo` and `getpeeredinfo` functions were not implemented either.

Chapter 8

Conclusion

The presented design of the native HIP API meets the given 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 ED concept provides the means to conceal details of the HIss and ephemeral locators. An ED is an opaque handle to a HI and it can be used directly in the sockets API function calls. The ED 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 most important 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 functionality is implemented in the resolver library. It provides a new *endpoint resolver*. For ease of use, the resolver is syntactically almost identical to the sockets API resolver. 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 but the ED approach guarantees forward compatibility with HIP. Also, the transition to IPv6 is more transparent as the ED hides the details of IP addresses. The binding model used for the ED is quite flexible, because it allows the reuse of an ED 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 ED 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] Hari Balakrishnan, Karthik Lakshminarayanan, Sylvia Ratnasamy, Scott Shenker, Ion Stoica, and Michael Walfish. A layered naming architecture for the internet. In *ACM SIGCOMM 2004, Portland, OR*, September 2004.

[2] 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/`.

[3] J. Noel Chiappa. *Endpoints and Endpoint Names: A Proposed Enhancement to the Internet Architecture*, 1999. `http://users.exis.net/~jnc/tech/endpoints.txt`.

[4] 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`.

[5] P. Florissi, Y. Yemini, and D. Florissi. *QoSockets: a New Extension to the Sockets API for End-to-End Application QoS Management*, May 1999.

[6] 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`.

[7] 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`.

[8] 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`.

[9] 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`.

[10] 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`.

[11] Troels Walsted Hansen. *Multihoming with Internet Protocol Version 6*, December 1999. `http://www.vermicelli.pasta.cs.uit.no/ipv6/students/troels/thesis.pdf`.

[12] 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`.

[13] Thomas Henderson, Jeff Ahrenholz, and et al. Boeing's unpublished hip implementation.

[14] Thomas R. Henderson. Host mobility for IP networks: A comparison. *IEEE Network Magazine*, 17(6):18–26, November 2003.

[15] Hip for bsd implementation. `http://www.hip4inter.net/`.

[16] The HIPL Group. *Host Identity Protocol for Linux.* `http://www.gaijin.iki/hipl/`.

[17] 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`.

[18] Miika Komu. Host identity payload in home networks. Seminar paper, Helsinki University of Technology, Espoo, Finland, April 2002.

[19] Julien Laganier. Julien laganier's unpublished hip implementation.

[20] 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`.

[21] 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`.

[22] Jukka Manner and Markku Kojo. *RFC 3753: Mobility Related Terminology.* Internet Engineering Task Force, June 2004. `http://www.ietf.org/rfc/rfc3753.txt`.

[23] Arifumi Matsumoto. *LIN6 Multihoming API.* Internet Engineering Task Force, January 2004. [Expired Internet Draft].

[24] 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`.

[25] A. McGregor. Pyhip release 18 march 2003. `http://www.sharemation.com/adm01bass/pyhip-2003-03-18.tar.bz2`.

[26] 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`.

[27] 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`.

[28] 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`.

[29] 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`.

[30] 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.`

[31] Pekka Nikander. A case for host identity payload: An architecture for multihomed mobile hosts, February 2002. unpublished manuscript.

[32] 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-02.txt`.

[33] 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`.

[34] Erik Nordmark. *Multihoming without IP Identifiers.* IETF, October 2003. [Expired Internet Draft].

[35] C. Perkins. *RFC 3344: IP Mobility Support for IPv4.* Internet Engineering Task Force, August 2002. `http://www.ietf.org/rfc/rfc3344.txt`.

[36] Charles Perkins. *RFC 2002: IP Mobility Support.* Internet Engineering Task Force, October 1996. `http://www.ietf.org/rfc/rfc2002.txt`.

[37] Netlink S.a.s. Netlink - communication between kernel and user. `//www.netlink.it/`.

[38] Kristian Slavov. *Implementing HIP Algorithms in Linux Kernel.* Helsinki University of Technology, January 2004.

[39] 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`.

[40] W. Richard Stevens. *UNIX Network Programming, Volume 1: Networking APIs: Sockets and XTI.* Prentice Hall, Upper Saddle River, New Jersey, 2nd edition, 1997.

[41] 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`.

[42] Randall R. Stewart and Xiaobing Xie. *Stream Control Transmission Protocol (SCTP).* Addison-Wesley, November 2001.

[43] Martin Stiemerling and Juergen Quittek. *Problem Statement: HIP operation over Network Address Translators.* Internet Engineering Task Force, July 2004. [Internet Draft] `http://www.ietf.org/internet-drafts/draft-stiemerling-hip-nat-01.txt`.

[44] 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`.

[45] 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.

[46] USAGI project - linux IPv6 development project. `http://www.linux-ipv6.org/`.

[47] John Viega, Matt Messier, and Pravir Chandra. *Networking Security with OpenSSL.* O'Reilly, jun 2002.

[48] Brian Wellington. *RFC 3007: Secure Domain Name System (DNS) Dynamic Update.* Internet Engineering Task Force, November 2000. `http://www.ietf.org/rfc/rfc3007.txt`.

[49] Joel M. Winett. *RFC 0147: The Definition of a Socket.* Internet Engineering Task Force, May 1971. `http://www.ietf.org/rfc/rfc0147.txt`.

[50] 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/Opodis02-Ylitalo-et-al.pdf`.

[51] 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
 * conntest-client-native.
 */
#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 <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_ed peer_ed;
  char *port_name;
  char mylovemostdata[IP_MAXPACKET];
  int recvnum, sendnum;
  int serversock = 0, sockfd = 0;
  int err = 0;
  int socktype;
  socklen_t peer_ed_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: uknown 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_ed,
      &peer_ed_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_ed,
       &peer_ed_len) > 0) {
mylovemostdata[recvnum] = '\0';
printf("%s", mylovemostdata);
if (recvnum == 0) {
  break;
}

/* send reply */
sendnum = sendto(sockfd, mylovemostdata, recvnum, 0,
 (struct sockaddr *) &peer_ed, peer_ed_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 conntest-server-xx.
 *
 * usage: ./conntest-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_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;
}

/* 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, gepi_strerror(err));
    goto out;
  }

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

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

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

  while ((ch = fgetc(stdin)) != EOF && (datalen < IP_MAXPACKET)) {
    mylovemostdata[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, mylovemostdata + 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_ed *my_ed = 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 = 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;
}

my_ed = getlocaled(endpoint, NULL, NULL, NULL);
if (!my_ed) {
  printf("Failed to set up my ED (%d)\n");
  err = 1;
  goto out;
}

/* We have to bind to the ED to use it. */
err = bind(sockfd, (struct sockaddr *) &my_ed, sizeof(struct sockaddr_ed));
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 getpeered 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_ed *) res->ei_endpoint)->ed_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, mylovemostdata + 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)
    close(sockfd); // discard errors
  if (res)
     free_endpointinfo(res);
  if (my_ed)
     free(my_ed);

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

  return err;
}
```