Mobile IPv6 Home Agent Service API

Implementation Agreement

February 7, 2005
Revision 1.0

Editors:

Karen Nielsen, Ericsson, karen.e.nielsen@ericsson.com
Erik B. Pedersen, Ericsson, erik.b.pedersen@ericsson.com

THIS DOCUMENT AND THE INFORMATION CONTAINED HEREIN IS PROVIDED ON AN "AS IS" BASIS WITHOUT ANY WARRANTY OF ANY KIND. THE INFORMATION, CONCLUSIONS AND OPINIONS CONTAINED IN THE DOCUMENT ARE THOSE OF THE AUTHORS, AND NOT THOSE OF NPF. THE NPF DOES NOT WARRANT THE INFORMATION IN THIS DOCUMENT IS ACCURATE OR CORRECT. THE NPF DISCLAIMS ALL WARRANTIES, WHETHER EXPRESS, IMPLIED OR STATUTORY, INCLUDING BUT NOT LIMITED THE IMPLIED LIMITED WARRANTIES OF MERCHANTABILITY, TITLE OR FITNESS FOR A PARTICULAR PURPOSE AND NON-INFRINGEMENT OF THIRD PARTY RIGHTS.

The words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", “MAY", and "OPTIONAL" in the remainder of this document are to be interpreted as described in the NPF Software API Conventions Implementation Agreement revision 1.0.

For additional information contact:
The Network Processing Forum, 39355 California Street,
Suite 307, Fremont, CA 94538
+1 510 608-5990 phone ✦ info@npforum.org

Table of Contents

List of Figures

Glossary

API	Applications Programming Interface
BC	Binding Cache
CoA	Care-of-Address
CP	Control Plane
DAD	Duplicate Address Detection
FAPI	NPF Functional API
FIB	Forwarding Information Base
HA	Home Agent
HAO	Home Address Destination Option
HoA	Home Address (of mobile node)
IETF	Internet Engineering Task Force
IKE	Internet Key Exchange
IP	Internet Protocol
IPSec	IP Security
IPv6	Internet Protocol version 6
LFB	Logical Functional Block
MIPv6	Mobile IPv6
MIPv6 HA	Mobile IPv6 Home Agent
MH	MIPv6 Mobility header
MN	Mobile Node
MTU	Maximum Transmission Unit
NE	Network Element
ND	Neighbor Discovery
NP	Network Processor
NPE	Network Processing Element
NPF	Network Processing Forum
PMTU	Path Maximum Transmission Unit
RFC	Request For Comments (IETF standard)
SA	Security Association
SAPI	NPF Service API
SL	Service ILayer
IM API	NPF Interface Management API
PH API	NPF Packet idr API

1. Revision History

Revision	Date	Reason for Changes
1.0	2/7/2005	Rev. 1.0 of the Mobile IPv6 Home Agent Service API Implementation Agreement. Source: npf2004.071.06.

2. Introduction

This document defines a NPF Service API for the Mobile IPv6 Home Agent function (MIPv6 HA for short). The MIPv6 HA Service API serves to configure and manage some specific MIPv6 HA forwarding path functions.

The primary data plane task performed by a MIPv6 HA is to tunnel packets destined to mobile nodes (MNs for short), identified with their home addresses – denoted the HoA, to their current location in the IPv6 internet, the Care-of-Address - denoted the CoA, as well as conversely, to accept and decapsulate reversely tunneled packet from the mobile nodes and forward those onto the internet for further delivery.

On the MIPv6 HA to/from MN path, the packets are tunneled using IPv6-in-IPv6, possibly with the addition of IP Security ESP depending on the nature of the payload. In the first case the MIPv6 HA decapsulation and encapsulation function should be applied to the packets. In the latter case, the packets are processed by the corresponding IP Security decryption and encryption functions.

Mobile nodes use MIPv6 signaling to register for the MIPv6 HA service as well as to notify their Home Agents about their current location. Mobile node registration data, the HoA, the CoA as well as a number of auxiliary parameters, lifetime and more, are cached in the conceptual Binding Cache of the MIPv6 HA.

In an architecture with separation of forwarding and control elements, the (MIPv6) signaling processing is normally handled in the control plane, whereas payload handling functions such as IP forwarding and interface processing functions, and for the MIPv6 HA node function specifically, the MIPv6 HA tunnel encapsulation and decapsulation functions, are implemented in the forwarding plane.

The MIPv6 HA Service API provides the means for a HA control plane entity to manage the MIPv6 HA encapsulation and decapsulation tunneling function of the forwarding plane as well as to manage one auxiliary MIPv6 HA forwarding path function, the MIPv6 HA Proxy Neighbor Discovery function.

The anticipated user of the MIPv6 HA Service API is a conceptual MIPv6 HA Service Layer Module responsible for the processing of the MIPv6 signaling messages, for the maintenance of the Binding Cache in accordance with the MIPv6 signaling messages received and for the push down and retrieval of the appropriate Binding Cache attributes to the afore mentioned MIPv6 HA specific forwarding path functions.

The conceptual MIPv6HA Service Layer module may also be responsible for the instantiation of other forwarding path functions required for the MIPv6 HA function, this either directly via the respective Service APIs or indirectly via mediation with the Service Layer Users of the respective Service APIs

2.1 NPF MIPv6 HA Framework

The MIPv6 HA Service API is designed in accordance with how the MIPv6 HA node function is envisaged split in control and forwarding path functions as well as in accordance with how the management of the various forwarding path functions of a MIPv6 HA node then should be distributed over the suite of Service APIs of the NP Forum

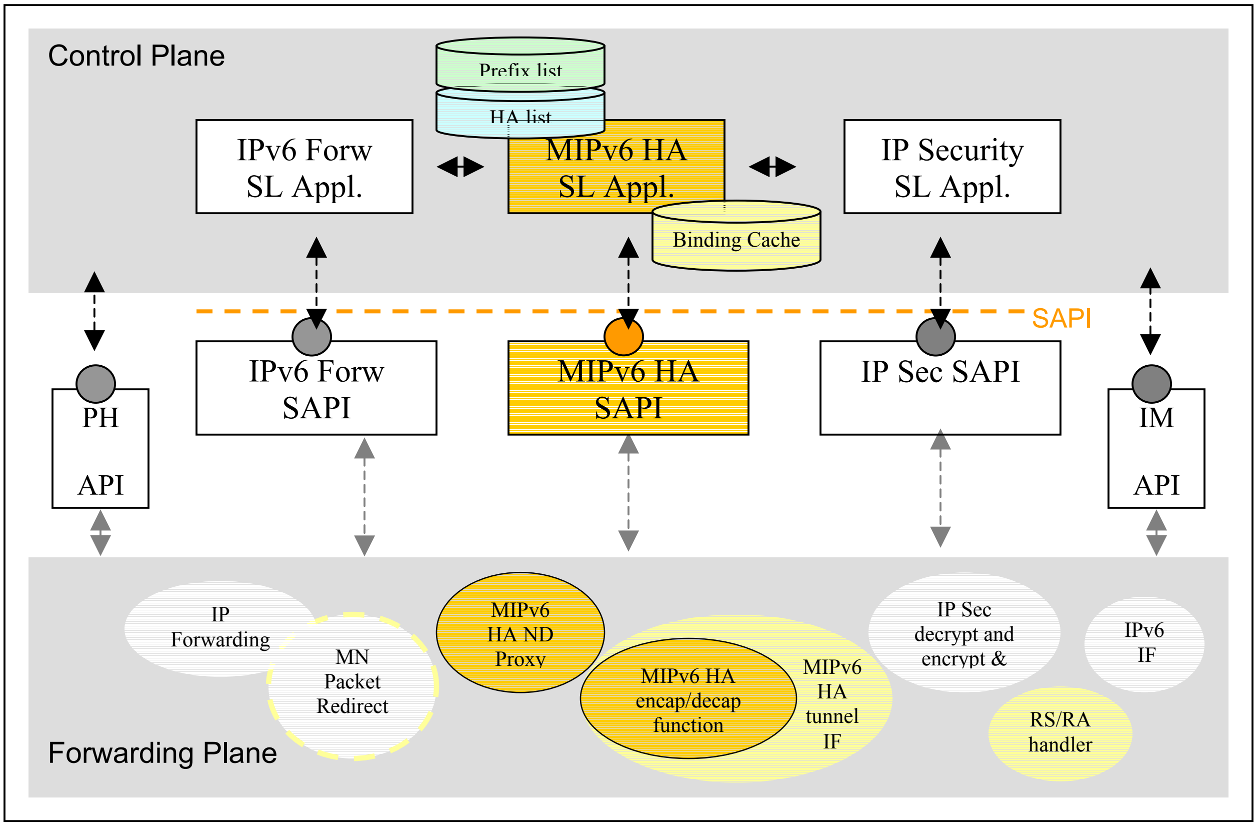

Figure 1 MIPv6 HA framework

2.1.1 Control Plane Functions

It is assumed that all functions associated with the processing of MIPv6 signaling messages reside in the Control Plane. This in particular means that the following data structures related to HA and MN Mobile IPv6 protocol signaling are assumed maintained and validated by the MIPv6 HA Service Layer module:

- Binding Cache
- HA lists
- Mobile Prefix Lists

The Binding Cache contains CoA and HoA address binding information[1] for the MNs currently served by the MIPv6 HA. Binding cache entries are created, modified and deleted on the basis of MIPv6 signaling messages interchanged in between the MIPv6 HA and mobile nodes.
The Mobile IPv6 Service API does not address the functions associated with the pure control plane functions associated with HA and Mobile Prefix list creation.

A Mobile IPv6 Home Agent naturally relies on a number of additional, generic IP router control plane functions such as Interface, Routing Table and IP Security Management in particular.

2.1.2 Forwarding Plane Functions

The MIPv6 HA SAPI addresses only a subset of the forwarding path functions needed in a MIPv6 HA node (see Figure 1). In general, the forwarding path functions on which the MIPv6 HA node function depends can be divided into the following two categories:

- Generic forwarding path functions such as Unicast Forwarding functions, Interface Management functions as well as IP Security decryption and encryption functions.
- Particular MIPv6 HA forwarding path functions and MIPv6 HA enhancements of generic forwarding path functions.

Generic forwarding path functions are handled via their respective Service APIs. Forwarding Path functions falling into the second category are the following:

- Functions that should be performed within the forwarding plane:
    - Redirect of packets destined for MNs
    - Reverse decapsulation and forward encapsulation of traffic from/to MNs
    - MIPv6 HA interface processing functions
- Functions that may be invoked in the forwarding plane or in the control plane depending on the implementation, the optional off load of which it is considered a MUST for the API framework to support:
    - MIPv6 HA ND proxy functions for MN packet intercept
    - Transmittal of IPv6 Router Advertisements with the MIPv6 HA specific attributes and options

Of these, the MIPv6 HA decapsulation and encapsulation function and the, optional, MIPv6 HA ND proxy function are managed via the specific MIPv6 HA Service API. Whereas the other functions are envisaged managed via other Service APIs; the MN packet redirect function is thought integrated with the unicast forwarding function and thus handled via the Unicast Forwarding Service API ([2]), the specific RS/RA options and attributes needed for the MIPv6 HA particular usage of the IPv6 Router Advertisement function is thought integrated with the generic IPv6 Router Advertisement function handled via the Interface Management API ([4]),

[1] In addition to the HoA and CoA binding information, each Binding Cache entry contains service attributes and status information including link-local address compatibility on/off, Key Management Mobility Capability on/off, sequence number of last BU received and lifetime of binding.

and finally, a specific MIPv6 HA tunnel interface is envisaged instantiated and managed via the Interface Management API ([4]). The latter interface is closely related to the MIPv6 HA encapsulation and decapsulation function.

The reader is referred to Appendix B for a full description of the MIPv6 HA software framework of the NP forum; This include a description of the split up of the MIPv6 HA node function in forwarding plane and control plane functions, the operation of the individual forwarding path functions and the overall managing of these functions over the Service API boundary.

Appendix B is informational. It is included as background information to put the MIPv6 HA Service API in its right context and present the rationale behind the particular design of the MIPv6 HA SAPI.

2.2 Assumptions and prerequisites

1. The design of the MIPv6 Service API is fundamentally based on the following split in functionality in between the Mobile IPv6 Home Agent Service API and the Interface Management API ([4]):
    i. Data structures related to the Mobile IPv6 Home Agent operation on link local/interface level which are *dynamic* in nature, that is, which reflect run-time behavior, are managed and instantiated by the MIPv6 HA Service Layer module by means of function calls within the Mobile IPv6 Home Agent SAPI.
    ii. Data structures related to the Mobile IPv6 Home Agent function on an interface which are static in nature, that is, which are associated with the enabling/disabling and general configuration of the Mobile IPv6 Home Agent function on an interface, is managed and instantiated by means of (an extension of) the Interface Management API.
2. Interface binding of data structures specifically related to Mobile IPv6 Home Agent function and managed by the Mobile IPv6 Home Agent Service Layer module, is realized via function calls in the Mobile IPv6 Home Agent SAPI (by passing down respective interface handles). It is by purpose not realized by passing down handles of these specific data structures through the Interface Management API. This mirrors the design decision made for the IP Security Service API ([3]).

2.3 Dependencies

The document depends on the NPF Software Implementation Agreement - Software API Conventions (Revision 2, September 2003) document ([1]) for basic type definitions.

The document depends on the NPF Software Implementation Agreement – Interface Management API document ([4]) for the definition of NPF_IfHandle_t and the encompassing definition of IPv6 interfaces and other functions to manage IPv6 interfaces, the IPv6“home link interface” in particular.

The document depends on the definition of a MIPv6 HA IPv6-in-IPv6 tunnel interface in an extension of the Interface Management API.

The MIPv6 HA Service Layer function as such, opposed to the MIPv6 HA Service *API*, depend on a number of functions of the forwarding plane which are instantiated by means of other APIs; forwarding and MN packet redirect functions instantiated via the IPv6 Unicast Forwarding

Service API ([2]), IP Security functions instantiated via the IP Security Service API ([3]) and MIPv6 HA Router Advertisement functions instantiated via an extension of the Interface Management API ([4]). Further the MIPv6 HA Service Layer function depends on APIs for transferal of packets from the control plane to the forwarding plane and vise-versa.

2.4 Scope

The Service API serves the Mobile IPv6 Home Agent function only. Other Mobile IPv6 node functions such as the Mobile Node and Corresponding Node functions are not covered.

The MIPv6 HA functionality is defined in [5] and [6].

2.5 Miscellaneous

With the aim of homogeneity over the various APIs of the NPF, the event rate limiting control function of the MIPv6 HA Service API has been modeled in a similar fashion as the corresponding function of the IP Security Service API.

3. API usage Model

The MIPv6 HA Service API is fundamentally based on the assumption that a “home link” IPv6 interface and a MIPv6 HA tunnel interface will be instantiated via the Interface Management API.

The “home link” interface is a logical IPv6 interface on which the IPv6 address of the MIPv6 HA is configured.

The MIPv6 HA tunnel interface is an open-ended IPv6-in-IPv6 tunnel interface which in turn is using the IPv6 “home link” interface as IP bearer interface with anchoring on the MIPv6 HA address. The MIPv6 HA tunnel interface should be thought of as the envelopment, in the IPv6 interface realm, of the individual, conceptual, tunnels in between the MIPv6 HA and the MNs it serves.

Henceforth, the conceptual tunnels in between the MIPv6 HA and its MNs are denoted “MIPv6 HA MN sub-tunnels” or simply “sub-tunnels”.

The creation via the Interface Management API of a MIPv6 HA tunnel interface anchored on an IPv6 address of an underlying IPv6 interface effectively represent the enabling of the MIPv6 HA function on the underlying IPv6 interface.

3.1 MIPv6 HA - MN tunnel handling functions

The function calls, data types and structures of the MIPv6 HA Service API related to the MIPv6 HA tunnel encapsulation and decapsulation functions are in this specification prefixed with the term “BC” for “Binding Cache”. Thereby illustrating the fact that the MIPv6 HA sub-tunnel functions of the forwarding plane instantiates the binding registration data administratively maintained in the conceptual Binding Cache on the MIPv6 HA.[2]

The attributes of an individual MIPv6 HA MN sub-tunnel/forwarding path Binding Cache entry are the following:

- Home Address of MN
- Care-of-Address of MN
- Path MTU value
- Lifetime Mode
- Binding Lifetime
- Remaining Binding Lifetime

- plus, allocated by the implementation :

- A sub-tunnel handle, NPF_MIPv6HA_subtunnel_t

Of these only the Home Address, the Care-of-Address, the Lifetime mode and (possibly) the Binding Lifetime are settable attributes.

[2] The MIPv6 HA module also uses the Binding Cache information when sending and receiving MIPv6 signaling messages to and from mobile nodes. The MIPv6 HA Service API does not address this usage of the Binding cache information as these transmit and receive functions are assumed residing in the control plane.

The Lifetime Mode governs whether the lifetime of the sub-tunnel/the binding cache entry is controlled by the forwarding path implementation or by the control plane. When controlled by the implementation, the Binding Cache entry is deleted by the implementation when expired. The remaining lifetime of a Binding Cache entry can optionally be queried using the BC query function, NPF_Mipv6HA_BC_EntryAttrGet.

Each sub-tunnel/binding cache entry is via the API function calls bound to the corresponding enveloping MIPv6 HA tunnel interface. The latter identified with its interface handle, NPF_IfHandle_t.

The settings, via the Interface Management API, on the enveloping MIPv6 HA tunnel interface governs part of the sub-tunnel functionality such as IPv6 source address (the MIPv6 HA address), the values of the max hops, the DSCP and the IPv6 flow label fields in the outer IPv6 header for sending. Further the PATH MTU mode and the possible static MTU value set on the MIPv6 HA tunnel interface governs the MTU handling on the sub-tunnels and whether PMTU should be performed or not. When PMTU is performed, the PMTU values of the individual sub-tunnels are retrieved via the BC query function call, NPF_MIPv6HA_BC_EntryAttrGet. Support for per sub-tunnel PMTU is optional.

The MIPv6 HA Service API allow for the retrieval of packet statistics per sub-tunnel entity. Collective packet statistics per MIPv6 HA entity can be retrieved using the corresponding functions of the Interface Management API.

The individual sub-tunnels are over the API identified via a sub-tunnel handle, NPF_MIPv6HA_subtunnelHandle_t, provided by the implementation, by the MN HoA address or by a structure including both; the NPF_MIPv6HA_subtunnelIdentifiers_t.

The MIPv6 HA Service API provides the following function calls to manage the MIPv6 HA sub-tunnels/the Binding Cache entry information of the forwarding plane.

- NPF_Mipv6HA_BC_EntryAdd
- NPF_Mipv6HA_BC_EntryDelete
- NPF_Mipv6HA_BC_EntryAttrGet
- NPF_Mipv6HA_BC_EntryStatsGet
- NPF_Mipv6HA_BC_GetAll
- NPF_Mipv6HA_BC_Flush
- NPF_Mipv6HA_BC_AttrGet

For further details on the individual function calls, the reader is referred to Section 5.

3.2 MIPv6 HA ND proxy function

The “home link” interface is the MIPv6 HAs link to the home network of the mobile nodes. Due to standard routing, packets destined for the mobile nodes home addresses (HoA) are routed to the home network regardless of whether the mobile nodes are at home or not. The MIPv6 ND proxy function serves to make sure that packets that arrives on the home network, and which are destined for mobile nodes away from home, locally are destined (in terms of last hop lower layer addressing) for the MIPv6 HA.

An integral part of the MIPv6 ND Proxy function is that it must perform Duplicate Address Detection (DAD) on the addresses for which it is ND proxy'ing. DAD is performed in accordance with the generic IPv6 settings (NPF_IfIPv6DAD_Transmits_t) on the home link interface.

The MIPv6 HA Service API supports off load of the MIPv6HA ND Proxy function to the forwarding plane. Support for this function is however optional. The MIPv6HA ND Proxy function can be managed via the following, optional, API calls:

- NPF_Mipv6HA_ProxyND_AddressAdd
- NPF_Mipv6HA_ProxyND_AddressDelete
- NPF_Mipv6HA_ProxyND_AddrStateGet
- NPF_Mipv6HA_ProxyND_GetAll
- NPF_Mipv6HA_ProxyND_Flush
- NPF_Mipv6HA_ProxyND_AttrGet

The API calls take the latter only or both of the following attributes:

- The address(es) of the Mobile Node(s) for which ND proxy'ing should be performed
- The home link interface handle

For the calls that return the address status after DAD validation, the following address states are applicable:

- valid
- probing
- invalid

When "valid", the address has passed DAD validation and will be ND proxy'ed until deleted. When "invalid", the address failed DAD validation and must be deleted (this also means that the MIPv6 registration message for the address must be declined). When "probing", the address is undergoing DAD validation.

The NPF_Mipv6HA_ProxyND_AddressAdd function call will complete prior to DAD taking place and the completion callback will not include the address state. The result of the DAD validation is communicated in a subsequent event, the NPF_MIPv6HA_PROXYND_DAD event, which returns the state of the address after DAD validation.

Multiple MIPv6 HA instances may be running on the same node, such instances should ideally be anchored on different "home link" interfaces of the node but may, in principle, also merely be anchored on different addresses configured on the same logical IPv6 interface.

Only one instance on the MIPv6 HA Service API is required for management of the forwarding path functions of the multiple MIPv6 HA instances. It should be noted however, that in the case of multiple MIPv6 HA instances being anchored on the same logical interface, then it will be indistinguishable from the ND_Proxy function calls and attributes themselves which addresses are accredited to which MIPv6 HA instance.

For further details on the individual function calls, the reader is referred to Section 5.

4. Data Types

4.1 MIPv6 SAPI Data Types

4.1.1 MIPv6HA subtunnel handle: NPF_MIPv6HA_SubtunnelHandle_t

```
/*
* A unique identifier selected by the implementation
*/
typdedef NPF_unit32_t   NPF_MIPv6HA_SubtunnelHandle_t;
```

4.1.2 MIPv6HA subtunnel identifiers: NPF_MIPv6HA_SubtunnelIdentifiers_t

```
/*
*     Subtunnel Identifier
*
*/
typedef struct {
      NPF_MIPv6HA_SubtunnelHandle_t       subtunnelHandle;
      NPF_IPv6Address_t                   HoA;
} NPF_MIPv6HA_SubtunnelIdentifiers_t;
```

4.1.3 MIPv6HA subtunnel identifiers Array : NPF_MIPv6HA_SubtunnelIdentifiersArray_t

```
/*
*     Subtunnel Identifier Array
*
*/
typedef struct {
      NPF_uint32_t                        nCount;
      NPF_MIPv6HA_SubtunnelIdentifiers_t  *subtunnelIdentifiersArray;
} NPF_MIPv6HA_SubtunnelIdentifiersArray_t;
```

4.1.4 MIPv6HA Binding Cache Entry: NPF_MIPv6HA_BC_Entry_t

```
/*
*  MIPv6HA Binding Cache Entry
*/
typedef struct {
           NPF_MIPv6HA_SubtunnelHandle_t       subtunnelHandle;
           NPF_IPv6Address_t                   HoA;
           NPF_IPv6Address_t                   CoA;
           NPF_uint16_t                        PMTU;
           NPF_MIPv6HA_LifetimeMode_t          lifetimeMode;
           NPF_unit32_t                        bindingLifetime;
           NPF_unit32_t                        remainingBindingLifetime;
} NPF_MIPv6HA_BC_Entry_t
```

Comments:

- The subtunnelhandle is selected by the implementation. It is a read only attribute.
- PMTU is a read only attribute. Per tunnel PMTU is only performed when PMTU mode of associated MIPv6 HA tunnel interface is enabled. When PMTU mode is disabled the

value of this field should be ignored by the application, though an implementation may choose to indicate the MTU of the associated MIPv6 HA tunnel interface in this field.

- The lifetime mode indicates whether the BC lifetime should be controlled by the implementation. When the Lifetime mode is set as ON, the remaining lifetime will be decremented by the implementation. When the remaining lifetime reaches zero the BC entry will be deleted by the implementation (issuing a BC entry lifetime expired event). When the Lifetime mode is set as OFF all BC entries are considered by the implementation to have infinite lifetime. BC cache entries of infinite lifetime must be controlled by the application.

```
/*
* BC entry Lifetime Mode
*/
typedef enum {
NPF_MIPv6HA_LIFETIME_ON  =1, /* Lifetime (secs) monitoring on */
NPF_MIPv6HA_LIFETIME_OFF =2 /* Lifetime (secs) monitoring off */
} NPF_MIPv6HA_LifetimeMode_t
```

- Binding Lifetime is set by the application. When the Lifetime mode is not set, this field will be ignored by the implementation and the remaining lifetime will not be controlled for such entries.

- Remaining lifetime value is a read only attribute.

Support for Lifetime control in the implementation is an **optional** feature.

4.1.5 MIPv6HA MN Statistics: NPF_MIPv6HA_BC_EntryStats_t

```
/*
* BC Entry Packet Statistics
*/
typdedef struct {
NPF_MIPv6HA_SubtunnelHandle_t     subtunnelHandle;
NPF_IPv6Address_t                 HoA;
NPF_unit64_t                      mipv6NodeHCInOctets;
NPF_unit64_t                      mipv6NodeHCInPkts;
NPF_unit64_t                      mipv6NodeHCOutOctets;
NPF_unit64_t                      mipv6NodeHCOutPkts;
NPF_unit64_t                      dropRxOctets;
NPF_unit64_t                      dropRxPkts;
NPF_unit64_t                      dropTxOctets;
NPF_unit64_t                      dropTxPkts;
} NPF_MIPv6HA_BC_EntryStats_t
```

4.1.6 MIPv6HA Proxy ND Address entry: NPF_MIPv6HA_ProxyND_Entry_t

```
/*
* MIPv6HA Proxy ND address entry
*/
typdedef struct {
      NPF_IPv6Address_t                            Addrs;
      NPF_IPv6AddrState_t                     AddrsState;
} NPF_MIPv6HA_ProxyND_Entry_t
```

Comments:

- NPF_IPv6AddrState_t is a read only attribute. NPF_IPv6AddrState is defined in the IM API, but included here for completeness:

  ```
  /*
  * IPv6 address state
  */
  typedef enum {
  NPF_IPv6_ADDR_VALID = 1,
  NPF_IPv6_ADDR_PROBING = 2,
  NPF_IPv6_ADDR_INVALID = 3
  } NPF_IPv6AddrState_t
  ```

  An address that is added to the MIPv6HA Proxy ND Address Entry table on the home link interface will initially be in probing state until DAD has concluded. After that it will be in valid state if DAD succeeded and in invalid state if a DAD collision occurred. An invalid address must be deleted by the application. DAD is performed is accordance with the NPF_IfIPv6DAD_Transmits_t value set on the home link interface.

- For any address that is added to an interfaces MIPv6HA_Proxy ND structure the implementation will attempt to perform DAD and further if DAD succeeds if will multicast an unsolicited NA for the address with the overwrite bit set. After that and until the address entry is removed the implementation will defend the address using the standard Proxy ND mechanisms. In case the interface's operational or administrative status is down for a period of time, all addresses in the MIPv6HA_Proxy ND structure of the interface must be DAD validated anew when the interface comes up again.

4.1.7 IPv6 address Array: NPF_IPv6AddressArray_t

```
/*
*      Ipv6 address array
*
*/
typedef struct {
      NPF_uint32_t                        nCount;
      NPF_IPv6Address_t                  *Ipv6AddressArray;
} NPF_IPv6AddressArray_t;
```

4.2 Data Structures for Completion Callbacks

This section defines the control structures needed for a Completion Callback, which provides the response information to the application that invoked an asynchronous function call. Although an asynchronous function call may request the execution of a single operation, most of the asynchronous call functions of the MIPv6 HA SAPI have the ability to request the execution of multiple operations. The implementation may invoke the completion callback one or more times in order to provide responses for the total number of operations requested.

4.2.1 Completion Callback Types

```
/*
* Mipv6 HA completion callback types
*/
typedef enum {
      NPF_MIPV6HA_BC_ENTRY_ADD             =1,
      NPF_MIPV6HA_BC_ENTRY_DELETE          =2,
      NPF_MIPV6HA_BC_FLUSH                 =3,
      NPF_MIPV6HA_BC_ENTRY_ATTR_GET        =4,
      NPF_MIPV6HA_BC_ENTRY_STATS_GET       =5,
      NPF_MIPV6HA_PROXYND_ADDR_ADD         =6,
      NPF_MIPV6HA_PROXYND_ADDR_DELETE      =7,
      NPF_MIPV6HA_PROXYND_FLUSH            =8,
      NPF_MIPV6HA_PROXYND_ADDR_STATE_GET   =9,
      NPF_MIPV6HA_BC_TABLE_SPACE_GET      =10,
      NPF_MIPV6HA_BC_GET_ALL              =11,
      NPF_MIPV6HA_PROXYND_TABLE_SPACE_GET =12,
      NPF_MIPV6HA_PROXYND_GET_ALL         =13,
      NPF_MIPv6HA_RATE_LIMIT_EVENTS       =14
} NPF_MIPv6HA_CallbackType_t;
```

4.2.2 Completion Callback Data Structure

Each completion callback provides the NPF_MIPv6HA_CallbackData_t structure, whose members will have particular values depending on the invoking function, whether or not a single operation was requested and whether the operations were successful or not.

```
/*
* MIPv6HA Completion Callback Data
*/
typedef struct {
      NPF_MIPv6HA_CallbackType_t   type;
      NPF_boolean_t               allOK;
      NPF_uint32_t                numResp;
      NPF_MIPv6HA_AsyncResponse_t *resp;
} NPF_MIPv6HA_CallbackData_t;
```

Comments:

- type – Refers to the function invocation that led to the response.
- allOK – This field and the numResp field provide a flexible means of providing information regarding the number of responses in this callback and their status. The specific details for these

fields are provided below.

- numResp – This field and the allOK field provide a flexible means of providing information regarding the number of responses in this callback and their status. The specific details for these fields are provided below
- resp – A pointer to an array of response elements or the NULL pointer. Each array element contains a return code, indicating the completion status of the request element, and possibly may contain other information specific to the type of request.

Depending on the number of request invoked by the application the above field have different meaning. The specific details are the following:

Single operation request:

- If allOK = TRUE, then numResp = 0 and the “resp” pointer is NULL. This indicates the operation completed successfully and there is no other additional response data to return.
- If allOK = FALSE, then numResp = 1 and the “resp” pointer points to a response structure. If the returnCode field indicates NPF_NO_ERROR, the operation completed successfully and there is additional response data in the structure. Otherwise, the operation failed and the reason is indicated by the returnCode.

Multiple operations request:

- If all operations completed successfully at the same time and there is no additional response data to provide, then allOK = TRUE, numResp = 0 and the “resp” pointer is NULL.
- If all operations completed successfully at the same time, but there is additional response data to provide, then allOK = FALSE, numResp indicates the total number of requested operations and the “resp” pointer points to an array of response structures. The returnCode field will indicate NPF_NO_ERROR.
- If some operations completed succesfully, but not all, then:
  - allOK = FALSE, numResp = the number of request operations completed.
  - The “resp” pointer will point to an array of response structures, each one containing one element for each completed request. For operations that completed successfully, the returnCode field will indicate NPF_NO_ERROR and additional response data may be present, depending on the type of function invocation. For operations that failed, the reason is indicated by the returnCode field, again additional response data may be present, depending on the type of function invocation.

  Callback function invocations are repeated in this fashion until all requests are complete. Responses are not repeated for request elements already indicated as complete in earlier callback function invocations.

4.2.3 Asynchronous Response Data Structure

One or more of the following structures may be provided to the callback function in the response array within the NPF_MIPv6HA_callbackData_t structure.

```
typedef struct {
      NPF_MIPv6HA_errorType_t                       errorCode;
      union{
           NPF_IfHandle_t                          ifHandle;
           NPF_uint32_t                            unused;
      } u1;
      union {
           NPF_MIPv6HA_SubtunnelIdentifiers_t       subtunnelIdentifiers;
           NPF_MIPv6HA_BC_Entry_t                   *BCentry;
           NPF_MIPv6HA_BC_EntryStats_t              *BCentrystats;
           NPF_IPv6Address_t                        proxyNDAddress;
           NPF_MIPv6HA_ProxyND_Entry_t              proxyNDEntry;
           NPF_MIPv6HA_SubtunnelIdentifiersArray_t  *subtunnelIdentifiers;
           NPF_Ipv6AddressArray_t                   *Ipv6Address;
           NPF_uint32_t                             entryspaceRemaining;
           NPF_MIPv6HA_Event_t                      eventType
           NPF_uint32_t                             unused;
      } u2;
} NPF_MIPv6HA_AsyncResponse_t;
```

Comments:

- The error code indicates an error or the success of a particular request operation.
- Embedded within the u1 structure, the asyncResponse structure contains the interface handle, if any, given in the function that invoked the response. The following table shows the usage of this structure and the meaning of the NPF_ifHandle_t when used for the various callbacks.

Completion Callback Type Code	**u1 and IfHandle meaning in NPF_MIPv6HA_AsyncResponse_t**
NPF_MIPV6HA_BC_FLUSH,	IfHandle of MIPv6HA tunnel interface
NPF_MIPV6HA_BC_ENTRY_ADD,	IfHandle of MIPv6HA tunnel interface
NPF_MIPV6HA_BC_ENTRY_DELETE	IfHandle of MIPv6HA tunnel interface
NPF_MIPV6HA_BC_ENTRY_ATTR_GET	IfHandle of MIPv6HA tunnel interface
NPF_MIPV6HA_BC_ENTRY_STATS_GET	IfHandle of MIPv6HA tunnel interface
NPF_MIPV6HA_PROXYND_FLUSH	IfHandle of home link interface
NPF_MIPV6HA_PROXYND_ADDR_ADD	IfHandle of home link interface
NPF_MIPV6HA_PROXYND_ADDR_DELETE	IfHandle of home link interface
NPF_MIPV6HA_PROXYND_ADDR_STATE_GET	IfHandle of home link interface
NPF_MIPV6HA_BC_TABLE_SPACE_GET	IfHandle of MIPv6HA tunnel interface
NPF_MIPV6HA_BC_GET_ALL	IfHandle of MIPv6HA tunnel interface
NPF_MIPV6HA_PROXYND_TABLE_SPACE_GET	IfHandle of home link interface
NPF_MIPV6HA_PROXYND_GET_ALL	IIfHandle of home link interface
NPF_MIPv6HA_RATE_LIMIT_EVENTS	unused

- Embedded within the u2 structure, the asyncResponse structure contains potential information requested by the operation.

  The following table summarizes the u2 information returned in the completion callback data structure by each function call in the MIPv6 HA SAPI.

Function Name	**Completion Callback Type Code**	**Structure Returned in u2 of**

		NPF_MIPv6HA_AsyncEsponse_t
NPF_Mipv6HA_BC_Flush	NPF_MIPV6HA_BC_FLUSH,	unused
NPF_Mipv6HA_BC_EntryAdd	NPF_MIPV6HA_BC_ENTRY_ADD	NPF_MIPv6HA_SubtunnelIdentifiers_t
NPF_Mipv6HA_BC_EntryDelete	NPF_MIPV6HA_BC_ENTRY_DELETE	NPF_MIPv6HA_SubtunnelIdentifiers_t
NPF_Mipv6HA_BC_EntryAttrGet	NPF_MIPV6HA_BC_ENTRY_ATTR_GET	*NPF_MIPv6HA_BC_Entry_t
NPF_Mipv6HA_BC_EntryStatsGet	NPF_MIPV6HA_BC_ENTRY_STATS_GET	*NPF_MIPv6HA_BC_EntryStats_t
NPF_Mipv6HA_ProxyND_Flush	NPF_MIPV6HA_PROXYND_FLUSH	Unused
NPF_Mipv6HA_ProxyND_AddressAdd	NPF_MIPV6HA_PROXYND_ADDR_ADD	NPF_IPv6Address_t
NPF_Mipv6HA_ProxyND_AddressDelete	NPF_MIPV6HA_PROXYND_ADDR_DELETE	NPF_IPv6Address_t
NPF_Mipv6HA_ProxyND_AddrStateGet	NPF_MIPV6HA_PROXYND_ADDR_STATE_GET	NPF_MIPv6HA_ProxyND_Entry_t
NPF_Mipv6HA_BC_TableSpaceGet	NPF_MIPV6HA_BC_TABLE_SPACE_GET	uint32
NPF_Mipv6HA_BC_GetAll	NPF_MIPV6HA_BC_GET_ALL	*NPF_MIPv6HA_SubtunnelIdentifiersArray_t
NPF_Mipv6HA_ProxyND_TableSpaceGet	NPF_MIPV6HA_PROXYND_TABLE_SPACE_GET	uint32
NPF_Mipv6HA_ProxyND_GetAll	NPF_MIPV6HA_PROXYND_GET_ALL	*NPF_Ipv6AddressArray_t
NPF_MIPv6HA_RateLimitEvents	NPF_MIPv6HA_RATE_LIMIT_EVENTS	NPF_MIPv6HA_Event_t

4.3 Data Structures for Event Notifications

4.3.1 Mipv6HA Event Type: NPF_MIPv6HA_Event_t

```
/*
*     MIPv6 HA Event Type
*/
typedef enum MIPv6 HA Event {
      NPF_MIPv6HA_PROXYND_DAD                  =1,
      NPF_MIPv6HA_BINDING_LIFETIME_EXPIRED    =2,
      NPF_MIPv6HA_BC_ENTRY_MISS              =3,
      NPF_MIPv6HA_SUBTUNNEL_ENDPOINT_AUTH_FAILED =4
} NPF_MIPv6HA_Event_t;
```

4.3.2 Event Notification Structures:

```
/*
*     MIPv6 HA Event Data Structure
*/
typedef struct NPF_MIPv6HA_EventData {
      NPF_MIPv6HA_Event_t                               eventType;
      union {
      NPF_MIPv6HA_ProxyND_DAD_t                         proxyND_DAD;
      NPF_MIPv6HA_BindingLifetimeExpired_t              bindingLftExp;
      NPF_MIPv6HA_BC_EntryMiss_t                        BC_entryMiss;
      NPF_MIPv6HA_SubtunnelEndpointAuthFailed_t         subtnlEndpAuthFail;
      } u;
} NPF_MIPv6HA_EventData_t;
```

```
/*
*     MIPv6 HA Event Array
*/
typedef struct NPF_MIPv6HA_EventArray {
      NPF_uint16_t                  n_data;
      NPF_MIPv6HA_EventData_t      *eventData;
} NPF_MIPv6HA_EventArray_t;
```

4.3.2.1 MIPv6HA Proxy ND DAD event: NPF_MIPv6HA_ProxyND_DAD_t

```
typedef struct {
      NPF_IfHandle_t                    homelinkInterfaceHandle;
      NPF_IPv6Address_t                 ucAddrs;
      NPF_IPv6AddrState_t               ucAddrsState;
} NPF_ MIPv6HA_ProxyND_DAD_t
```

Comments:

- This event is generated whenever DAD concludes for an address added to the MIPv6HA NP Proxy structures on a home link interface. The address state included in the event data structure indicates whether DAD was successful or not.

4.3.2.2 MIPv6HA Binding lifetime expired Event: NPF_MIPv6HA_BindingLifetimeExpired_t

```
typdedef struct {
      NPF_IfHandle_t               Mipv6HA_tunnelInterfaceHandle;
      NPF_MIPv6HA_subtunnelHandle_t  subtunnelHandle;
      NPF_IPv6Address_t            HoA;
} NPF_MIPv6HA_BindingLifetimeExpired_t
```

Comments:

- The event is generated when a BC entry has been deleted by the implementation due to lifetime expiration.

4.3.2.3 MIPv6HA no BC Entry found: NPF_MIPv6HA_BC_EntryMiss_t

```
typedef struct {
      NPF_Ipv6Address_t          IP destination address of packet;
} NPF_MIPv6HA_BC_EntryMiss_t
```

Comments:

- This event is generated when a packet has arrived for encapsulation at the tunnel module implementation for which no corresponding subtunnel can be located, i.e., destination address is not among HoA addresses in binding cache entries.

4.3.2.4 MIPv6HA endpoint authentication check failed: NPF_MIPv6HA_SubtunnelEndpointAuthFailed_t

```
typedef struct {
   NPF_ifHandle_t                     Mipv6HA_tunnelInterfaceHandle;
   NPF_MIPv6HA_SubtunnelHandle_t      subtunnelHandle;
   NPF_Ipv6Address_t                  HoA;
   NPF_Ipv6Address_t                  IP source address of inner packet;
   NPF_Ipv6Address_t                  IP destination address of inner packet;
   NPF_uint8_t                        repetition count;
} NPF_MIPv6HA_SubtunnelEndpointAuthFailed_t
```

Comments:

- This event is generated when an incoming encapsulated packet fails the (HoA, CoA) source address check. The subtunnelHandle is the handle of the binding cache entry pointed out by the address (the CoA) in the IPv6 source address of the outer IPv6 header.

- The repetition count is used when rate limitation is in effect; it provides the number of packets that would have triggered the same event on the same subtunnelHandle in case rate limitation were not in effect. When the repetition count is greater than one, IP address fields become ambiguous since they may vary between the offending packets; the rule that implementations should adhere to is to copy these fields from the last of the offending packets currently received.

4.3.3 MIPv6HA Event Mask : NPF_MIPv6HA_EventMask_t

```
/*
*     MIPv6 HA event bitmask used in the event registration call.
*/
typedef NPF_uint32_t NPF_MIPv6HA_EventMask_t;

/*
* The following values can be set for the MIPv6HAEventMask
*/

#define     NPF_IPSEC_EVENT_ALL_DISABLE                   (0) /* disable all */
#define     NPF_MIPv6HA_PROXYND_DAD_CONCLUDED             (1 << 0)
#define     NPF_MIPv6HA_BINDING_LIFETIME_EXPIRED          (1 << 1)
#define     NPF_MIPv6HA_BC_ENTRY_MISS                     (1 << 2)
#define     NPF_MIPv6HA_SUBTUNNEL_ENDPOINT_AUTH_FAILED    (1 << 3)
#define     NPF_MIPv6HA_EVENT_ALL_ENABLE                  0xFFFFFFFF
```

4.3.4 Rate Limiting Events: NPF_MIPv6HA_EventLimit_t

```
/*
*     MIPv6 HA Rate Limiting Events
*/
typedef enum {
      NPF_MIPv6HA_EVENT_LIMIT_TIME=1,      /* Time base limiting */
      NPF_MIPv6HA_EVENT_LIMIT_COUNT=2      /* Counter base limiting */
} NPF_MIPv6HA_EventLimitType_t;

typedef struct {
      NPF_MIPv6HA_Event_t               eventid;      /* Event HANDLE */
      NPF_MIPv6HA_EventLimitType_t      limitType;    /* Limit type */
      union{
            NPF_uint32_t numPerSec;     /* Event frequency in time */
            NPF_uint32_t nCount;        /* Generate 1 event for */
                                        /* every nCount encounters */
     }u;
} NPF_MIPv6HA_EventLimit_t;
```

Comments:

- NPF _MIPv6HA_SUBTUNNEL_ENDPOINT_AUTH_FAILED event type.

4.4 Error Codes

```
/*
*     Asynchronous error codes (returned in function callbacks)
*/

/*
*     MIPv6HA reserved error codes in relation to other NPF APIs
*     Note** The maximum range is 100
*/
#define NPF_MIPv6HA_BASE_ERR XXX /* Base value of XXX wrt other NPF codes */

/* Optional feature not supported */
#define NPF_MIPv6HA_E_OPTIONAL_FEATURE_NOT_SUPPORTED \
      ((NPF_MIPv6HA_ErrorType_t) NPF_MIPv6HA_BASE_ERR + 1)

/* System was unable to allocate sufficient memory to complete operation */
#define NPF_MIPv6HA_E_NOMEMORY      ((NPF_MIPv6HA_ErrorType_t)
NPF_MIPv6HA_BASE_ERR+2)

/* The Interface handle provided was not recognized as being valid */
#define NPF_MIPv6HA_E_INVALID_IF_HANDLE \
      ((NPF_MIPv6HA_ErrorType_t) NPF_MIPv6HA_BASE_ERR + 3)

/* Ipv6 ND Proxy Address not found*/
#define NPF_MIPv6HA_E_PROXYND_ADDR_UNKNOWN\
      ((NPF_MIPv6HA_ErrorType_t) NPF_MIPv6HA_BASE_ERR + 4)

/* Invalid parameter */
#define NPF_MIPv6HA_E_INVALID_PARAM\
      ((NPF_MIPv6HA_ErrorType_t) NPF_MIPv6HA_BASE_ERR + 5)

/* Duplicate HoA*/
#define NPF_MIPv6HA_E_DUPLICATE_HOA \
      ((NPF_MIPv6HA_ErrorType_t) NPF_MIPv6HA_BASE_ERR + 6)


/* Invalid or unknown tunnel handle - */
#define NPF_MIPv6HA_E_INVALID_SUBTUNNEL_HANDLE \
      ((NPF_MIPv6HA_ErrorType_t) NPF_MIPv6HA_BASE_ERR + 7)

/* Duplicate event id */
#define NPF_MIPv6HA_E_DUPLICATE_EVENT_ID \
      ((NPF_MIPv6HA_ErrorType_t) NPF_MIPv6HA_BASE_ERR + 8)

/* Invalid or unknown event ID  - */
```

```
#define NPF_MIPv6HA_E_INVALID_EVENT_ID \
      ((NPF_MIPv6HA_ErrorType_t) NPF_MIPv6HA_BASE_ERR + 9)
```

5. Functions

5.1 Completion Callbacks and Error Returns

Each of the functions defined in the MIPv6HA API can return an immediate error and most makes asynchronous callbacks.

Error codes eligible for immediate return are those defined in the NPF Software Conventions document ([1]) plus for some functions additional MIPv6 HA SAPI specific return codes. The usage of MIPv6 HA specific synchronous return codes for each API function is defined with each function description. The MIPv6 HA error codes that may be returned synchronously for certain functions are the following:

- `NPF_MIPv6HA_E_NOMEMORY` – The system is unable to allocate sufficient memory to complete this operation.
- `NPF_MIPv6HA_E_OPTIONAL_FEATURE_NOT_SUPPORTED`: An attempt was made to leverage an optional feature within the API, which is not supported by this implementation.
- `NPF_MIPv6HA_E_DUPLICATE_HOA`: Duplicate HoA addresses when attempting to create Binding Cache entries.

All other error codes of Section 4.4 must be returned in an asynchronous callback response. The usage of those is defined with each function description.

5.2 Completion Callback

5.2.1 Completion Callback Function

```
typedef void (*NPF_MIPv6HA_CallBackFunc_t)(
      NPF_IN NPF_userContext_t             userContext,
      NPF_IN NPF_correlator_t        correlator,
      NPF_IN NPF_MIPv6HA_CallbackData_t    MIPv6HA_CallbackData);
```

Description

The application registers this asynchronous response handling routine to the API implementation. The callback function is intended to be implemented by the application, and be registered to the NPF MIPv6HA Service API implementation through NPF_MIPv6HA_Register( ) function.

Input Parameters

- userContext
  The context item that was supplied by the application when the completion callback function was registered.
- correlator
  The correlator item that was supplied by the application when the API function call was made. The correlator is used by the application mainly to distinguish between multiple invocations of the same function.
- MIPv6HA_CallbackData

Pointer to a structure containing an array of response information related to the API function call. Contains information that are common among all functions, as well as information that are specific to a particular function. See NPF_MIPv6HA_callbackData_t definition for details.

Output Parameters

None.

Return Value

None.

5.2.2 Completion Callback Registration Function

```
NPF_error_t NPF_MIPv6HA_Register(
   NPF_IN NPF_userContext_t                   userContext,
   NPF_IN NPF_MIPv6HA_CallbackFunc_t          MIPv6HACallbackFunc,
   NPF_OUT NPF_callbackHandle_t              *MIPv6HA_CallbackHandle);
```

Description

This function is used by an application to register its completion callback function for receiving asynchronous responses related to API function calls. Application may register multiple callback functions using this function. The callback function is identified by the pair of userContext and MIPv6HA_CallbackFunc, and for each individual pair, a unique MIPv6HA_callbackid_t will be assigned for future reference. Since the callback function is identified by both userContext and MIPv6HA_CallbackFunc, duplicate registration of same callback function with different userContext is allowed. Also, same userContext can be shared among different callback functions. Duplicate registration of the same userContext and MIPv6HA_CallbackFunc pair has no effect, and will output a handle that is already assigned to the pair, and will return NPF_E_CALLBACK_ALREADY_REGISTERED.

Note : NPF_MIPv6HA_Register( ) is a synchronous function and has no completion callback associated with it.

In Parameters

- userContext
  A context item for uniquely identifying the context of the application registering the completion callback function. The exact value will be provided back to the registered completion callback function as its 1st parameter when it is called. Application can assign any value to the userContext and the value is completely opaque to the NPF MIPv6HA Service API implementation.
- MIPv6HA_CallbackFunc_t
  The pointer to the completion callback function to be registered.

Out Parameters

- NPF_callbackHandle_t

A unique identifier assigned for the registered userContext and MIPv6HA_CallbackFunc pair. This handle will be used by the application to specify which callback function to be called when invoking asynchronous NPF MIPv6HA Service API functions. It will also be used when de-registering the userContext and MIPv6HA_CallbackFunc pair.

Return Values

- NPF_NO_ERROR: The registration completed successfully.
- NPF_E_BAD_CALLBACK_FUNCTION: MIPv6HA_CallbackFunc is NULL.
- NPF_E_ALREADY_REGISTERED: No new registration was made since the userContext and MIPv6HA_CallbackFunc pair was already registered.
  Note: Whether this should be treated as an error or not is dependent on the application.

5.2.3 Completion Callback Deregistration

```
NPF_error_t NPF_MIPv6HA_Deregister(
     NPF_IN NPF__callbackHandle_t  MIPv6HA_CallbackHandle);
```

Description

This function is used by an application to de-register a pair of user context and callback function.
Note: If there are any outstanding calls related to the de-registered callback function, the callback function may be called for those outstanding calls even after de-registration.
Note: NPF_MIPv6HA_EventRegister( ) is a synchronous function and has no completion callback associated with it.

In Parameters

- NPF_callbackHandle_t
  The unique identifier representing the pair of user context and callback function to be de-registered.

Output Parameters

None.

Return Values

- NPF_NO_ERROR: The de-registration completed successfully.
- NPF_E_BAD_CALLBACK_HANDLE: The API implementation does not recognize the callback handle. There is no effect to the registered callback functions.

5.3 Event Notification

5.3.1 Event Notification Signature

```
typedef void (*NPF_MIPv6HA_EventCallFunc_t)(
      NPF_IN NPF_userContext_t          userContext,
      NPF_IN NPF_MIPv6HA_EventArray_t   eventArray);
```

Description

This handler function is for the application to register an event handling routine to the API implementation. One or more events can be notified to the application through a single invocation of this event handler function. Information on each event is represented in an array in the eventArray structure so that the application can traverse through the array and process each of the events. This event handler function is intended to be implemented by the application, and be registered to the API implementation through NPF_MIPv6HA_EventRegister() function.

Note: This function may be called any time after NPF_MIPv6HA_EventRegister() is called for it.

Input Parameters

- userContext: The context item that was supplied by the application when the event handler function was registered.
- eventArray: Data structure that contains an array of event information. See NPF_MIPv6HA_EventArray_t definition for details.

Output Parameters

None.

Return Codes

None.

5.3.2 Event Notification Registration

```
NPF_error_t NPF_MIPv6HA_EventRegister(
      NPF_IN NPF_userContext_t             userContext,
      NPF_IN NPF_MIPv6HA_EventCallFunc_t   eventCallFunc,
      NPF_IN NPF_MIPv6HA_EventMask         eventMask,
      NPF_OUT NPF_callHandle_t             *eventCallHandle);
```

Description

This function is used by an application to register its event handling routine for receiving notifications of MIPv6HA SAPI events. Applications MAY register multiple event handling routines using this function. The event handling routine is identified by the pair of userContext and eventCallFunc, and for each individual pair, a unique eventCallHandle will be assigned for future reference.

Since the event handling routine is identified by both userContext and eventCallFunc, duplicate registration of the same event handling routine with a different userContext is allowed. Also, the same userContext can be shared among different event handling routines. Duplicate registration of the same userContext and eventCallFunc pair has no effect, and will output a handle that is already assigned to the pair, and will return NPF_E_CALLBACK_ALREADY_REGISTERED.
This function also enables notifications for the events selected by the bits that are set in the eventMask parameter. A mask with all bits set selects all events of this SAPI. If the application wishes to change the selection of events, it may call the event registration function again with the same userContext and eventCallFunc, but with a different event selection mask. The events enabled are those whose bits were set in the most recent registration function call for a particular userContext and eventCallFunc pair.

Notes: Besides registering a handler function, this call enables events. The handler function could be called at any time following the invocation of `NPF_MIPv6HA_EventRegister().` `NPF_MIPv6HA_EventRegister()` is a synchronous function and has no completion callback associated with it.

Input Parameters

- `userContext`: A context item for uniquely identifying the context of the application registering the event handler function. The exact value will be provided back to the registered event handler function as its 1st parameter when it is called. Application can assign any value to the userContext and the value is completely opaque to the API implementation.
- `eventCallFunc`: Pointer to the event handler function to be registered.
- `eventMask`: a bitmask defining the events to enable for this callback

Output Parameters

- `eventCallHandle`: A unique identifier assigned for the registered `userContext` and `eventCallFunc` pair. This handle will be used by the application de- registering the `userContext` and `eventCallFunc` pair.

Return Codes

- `NPF_NO_ERROR`: The registration completed successfully.
- `NPF_E_BAD_CALLBACK_FUNCTION: eventCallFunc` is `NULL` or not recognized.
- `NPF_E_ALREADY_REGISTERED`: No new registration was made since the `userContext` and `eventCallFunc` pair was already registered.
- `NPF_MIPv6HA_E_OPTIONAL_FEATURE_NOT_SUPPORTED`: An attempt was made to leverage an optional feature within the API, which is not supported by this implementation (some events are optional).

5.3.3 Event Notification Deregistration

```
NPF_error_t NPF_MIPv6HA_EventDeregister(
            NPF_IN NPF_callHandle_t);
```

Description

This function is used by an application to de-register a pair of user context and event handler function.

Input Parameters

- `eventCallHandle`: The unique identifier representing the pair of user context and event handler function to be de-registered.

Output Parameters

None.

Return Codes

- `NPF_NO_ERROR`: The de-registration completed successfully.
- `NPF_E_BAD_CALLBACK_HANDLE`: The API implementation does not recognize the event handler handle. There is no effect to the registered event handler functions.

5.3.4 MIPv6HA Control Event Frequency

```
NPF_error_t NPF_MIPv6HA_RateLimitEvents (
      NPF_IN NPF_callbackhandle_t           cbHandle,
      NPF_IN NPF_correlator_t             cbCorrelator,
      NPF_IN NPF_errorReporting_t             errorReporting,
      NPF_IN NPF_uint32_t                     countEventData,
      NPF_IN NPF_MIPv6HA_EventLimit_t        *eventLimitArray);
```

Description

This function allows control over the number of events generated for each event type. Rate limiting may be set based on time or the accumulation of multiple events of the same type into a single event to the client application. Currently this function is only supported for the NPF_MIPv6HA_subtunnelEndpointAuthFailed_t event type. **This is an optional function.**

Input Parameters

- `cbHandle`: the registered callback handle.
- `cbCorrelator`: the application's context for this call.
- `errorReporting`: the desired level of feedback.
- `countEventData`: the number of events being configured
- `eventLimitArray`: rate limiting data associated with each event

Output Parameters

None.

Synchronous Return Codes

- NPF_NO_ERROR: The registration completed successfully.
- NPF_E_BAD_CALLBACK_HANDLE: callback handle is NULL or not recognized.
- NPF_E_FUNCTION_NOT_SUPPORTED: The function is not supported by the implementation.
- NPF_MIPv6HA_E_DUPLICATE_EVENT_ID – The event ID was duplicated in the event array

Asynchronous response

A single callback of type NPF_MIPv6HA_RATE_LIMIT_EVENT is generated in response to this function call. The following status codes may be returned.

- NPF_NO_ERROR – The operation was successful.
- NPF_MIPv6HA_E_NOMEMORY – The system was unable to allocate sufficient memory to complete this operation.
- NPF_MIPv6HA_E_INVALID_EVENT_ID – The event ID specified was not recognized as being valid.
- NPF_MIPv6HA_E_DUPLICATE_EVENT_ID – The event ID was duplicated in the event array

5.4 MIPv6 HA Service API

5.4.1 NPF_Mipv6HA_BC_EntryAdd

```
NPF_error_t NPF_ Mipv6HA_BC_Entry_Add (
          NPF_IN NPF_callbackHandle_t        cbHandle,
          NPF_IN NPF_correlator_t            cbCorrelator,
          NPF_IN NPF_errorReporting_t        errorReporting,
          NPF_IN NPF_ifHandle_t              Mipv6HA_tunnelInterfaceHandle,
          NPF_IN NPF_uint32_t                n_entries,
          NPF_IN NPF_MIPv6HA_BC_Entry_t      *BCentryarray);
```

Description

This function allows for the addition of a single or multiple BC entries to one MIPv6HA tunnel indicated by the NPF_ifHandle_t interface handle. The addition of a BC entry with an already existing HoA will result in a replacement of the BC entry parameters associated with that BC entry.

Input Parameters

- cbhandle: the registered callback handle.
- cbCorrelator: the application's context for this call.
- errorReporting: the desired level of feedback.
- Mipv6HA_tunnelInterfaceHandle: Ifhandle of MIPv6HA tunnel interface
- n_entries: the number of BC entries being configured
- BCentryarray: array of BC entries

Synchronous Return Codes

- NPF_NO_ERROR – The operation is in progress.
- NPF_E_UNKNOWN – The operation could not be completed due to problems encountered when handling the input parameters.
- NPF_E_BAD_CALLBACK_HANDLE – The callback handle is not valid.
- NPF_MIPv6HA_E_NOMEMORY - The system is unable to allocate sufficient memory to complete this operation.
- NPF_MIPv6HA_E_DUPLICATE_HOA - The HoA was duplicated in the BCentryarray.

Asynchronous response

A callback of type NPF_MIPV6HA_BC_ENTRY_ADD is generated in response to this function call. A total of n_entries asynchronous responses (NPF_MIPv6HA_AsyncResponse_t) will be passed to the callback function, in one or more invocations. Each response contains the interface handle of the MIPv6HA tunnel interface, the subtunnelhandle and HoA of the BCentry and the associated status code. The HoA is returned in order to allow for mapping of the subtunnelhandle to the BC entry created by the application. The following status codes may be returned.

- NPF_NO_ERROR – The operation was successful.

- NPF_E_UNKNOWN – The operation could not be completed. Reason unspecified.
- NPF_MIPv6_E_NOMEMORY – The system was unable to allocate sufficient memory to complete this operation.
- NPF_MIPv6HA_E_INVALID_IF_HANDLE - Handle is null, invalid or is not a MIPv6HA tunnel interface.
- NPF_MIPv6HA_E_INVALID_PARAM – Invalid parameters given in BC entry structure (e.g. invalid CoA).
- NPF_MIPv6HA_E_NOMEMORY – The system is unable to allocate sufficient memory to complete this operation.
- NPF_MIPv6HA_E_DUPLICATE_HOA - The HoA was duplicated in the BCentryarray.
- NPF_MIPv6HA_E_OPTIONAL_FEATURE_NOT_SUPPORTED: An attempt was made to leverage the support of binding cache lifetime and the implementation does not support this function.

5.4.2 NPF_Mipv6HA_BC_EntryDelete

```
NPF_error_t NPF_ Mipv6HA_BC_Entry_Delete (
      NPF_IN NPF_callbackHandle_t            cbHandle,
      NPF_IN NPF_correlator_t                cbCorrelator,
      NPF_IN NPF_errorReporting_t            errorReporting,
      NPF_IN NPF_ifHandle_t              Mipv6HA_tunnelInterfaceHandle,
      NPF_IN NPF_uint32_t                    n_entries,
      NPF_IN NPF_MIPv6HA_SubtunnelHandle_t   *subtunnelHandleArray);
```

Description

This function allows for the deletion of a single or multiple BC entries on one MIPv6HA tunnel identified by the NPF_ifHandle_t interface handle.

Input Parameters

- cbHandle: the registered callback handle.
- cbCorrelator: the application's context for this call.
- errorReporting: the desired level of feedback.
- Mipv6HA_tunnelInterfaceHandle: Ifhandle of MIPv6HA tunnel interface
- n_entries: the number of BC entries being deleted
- *subtunnelHandleArray: array of subtunnel handles to be deleted

Synchronous Return Codes

- NPF_NO_ERROR – The operation is in progress.
- NPF_E_UNKNOWN – The operation could not be completed due to problems encountered when handling the input parameters.
- NPF_E_BAD_CALLBACK_HANDLE – The callback handle is not valid.

Asynchronous response

A callback of type NPF_MIPV6HA_BC_ENTRY_DELETE is generated in response to this function call. A total of n_entries asynchronous responses (NPF_MIPv6HA_AsyncResponse_t) will be passed to the callback function, in one or more invocations. Each response contains the interface handle of the MIPv6HA tunnel interface, the subtunnelidentifiers of the BCentry and the associated status code. The following status codes may be returned.

- NPF_NO_ERROR – The operation was successful.
- NPF_E_UNKNOWN – The operation could not be completed. Reason unspecified.
- NPF_MIPv6HA_E_INVALID_SUBTUNNEL_HANDLE – Invalid or unknown subtunnelhandle
- NPF_MIPv6HA_E_INVALID_IF_HANDLE – Interface handle is null, invalid or is not a MIPv6HA tunnel interface.

5.4.3 NPF_Mipv6HA_BC_Flush

```
NPF_error_t NPF_ Mipv6HA_BC_Flush (
      NPF_IN NPF_callbackHandle_t         cbHandle,
      NPF_IN NPF_correlator_t             cbCorrelator,
      NPF_IN NPF_errorReporting_t         errorReporting,
      NPF_IN NPF_ifHandle_t             Mipv6HA_tunnelInterfaceHandle);
```

Description

This function allows for the deletion of all BC entries associated with a MIPv6HA tunnel interface indicated by the NPF_ifHandle_t interface handle.

Input Parameters

- cbHandle: the registered callback handle.
- cbCorrelator: the application's context for this call.
- errorReporting: the desired level of feedback.
- Mipv6HA_tunnelInterfaceHandle: Ifhandle of MIPv6HA tunnel interface

Synchronous Return Codes

- NPF_NO_ERROR – The operation is in progress.
- NPF_E_UNKNOWN – The operation could not be completed due to problems encountered when handling the input parameters.
- NPF_E_BAD_CALLBACK_HANDLE – The callback handle is not valid.

Asynchronous response

A callback of type NPF_MIPV6HA_BC_FLUSH is generated in response to this function call. A single asynchronous response, NPF_MIPv6HA_AsyncResponse_t, will be passed to the callback function containing the status code and the interface handle of the MIPv6HA tunnel interface. The following status codes may be returned.

- NPF_NO_ERROR – The operation was successful.
- NPF_E_UNKNOWN – The operation could not be completed. Reason unspecified.

- NPF_MIPV6HA_E_INVALID_IF_HANDLE: Ifhandle is null or invalid, or is not a Mipv6HA tunnel interface.

5.4.4 NPF_Mipv6HA_BC_EntryAttrGet

```
NPF_error_t NPF_ Mipv6HA_BC_EntryAttrGet (
      NPF_IN NPF_callbackHandle_t        cbHandle,
      NPF_IN NPF_correlator_t            cbCorrelator,
      NPF_IN NPF_errorReporting_t        errorReporting,
      NPF_IN NPF_ifHandle_t              Mipv6HA_tunnelInterfaceHandle,
      NPF_IN NPF_uint32_t                n_entries,
      NPF_IN NPF_MIPv6HA_SubtunnelHandle_t   *subtunnelHandleArray);
```

Description

This function allows for the retrieval of the attributes of a single or multiple BC entries on one MIPv6HA tunnel identified by the NPF_ifHandle_t interface handle. **This is an optional function.**

Input Parameters

- cbHandle: the registered callback handle.
- cbCorrelator: the application's context for this call.
- errorReporting: the desired level of feedback.
- Mipv6HA_tunnelInterfaceHandle: Ifhandle of MIPv6HA tunnel interface
- n_entries: the number of BC entries queried
- *subtunnelHandleArray: array of subtunnel handles to be queried

Synchronous Return Codes

- NPF_NO_ERROR – The operation is in progress.
- NPF_E_UNKNOWN – The operation could not be completed due to problems encountered when handling the input parameters.
- NPF_E_BAD_CALLBACK_HANDLE – The callback handle is not valid.
- NPF_E_FUNCTION_NOT_SUPPORTED: The function is not supported by the implementation.

Asynchronous response

A callback of type NPF_MIPV6HA_BC_ENTRY_ATTR_GET is generated in response to this function call. A total of n_entries asynchronous responses (NPF_MIPv6HA_AsyncResponse_t) will be passed to the callback function, in one or more invocations. Each response contains the interface handle of the MIPv6HA tunnel interface and the associated status code. If the error code indicates success, the union in the callback response structure contains a pointer to the NPF_MIPv6HA_BC_Entry_t structure for the BC entry.

The following status codes may be returned.

- NPF_NO_ERROR – The operation was successful.
- NPF_E_UNKNOWN – The operation could not be completed. Reason unspecified.
- NPF_MIPv6HA_E_INVALID_SUBTUNNEL_HANDLE - Invalid or unknown subtunnelHandle

- NPF_MIPV6HA_E_INVALID_IF_HANDLE: Ifhandle is null or invalid, or is not a Mipv6HA tunnel interface.

5.4.5 NPF_Mipv6HA_BC_EntryStatsGet

```
NPF_error_t NPF_ Mipv6HA_BC_EntryStatsGet (
      NPF_IN NPF_callbackHandle_t        cbHandle,
      NPF_IN NPF_correlator_t            cbCorrelator,
      NPF_IN NPF_errorReporting_t        errorReporting,
      NPF_IN NPF_ifHandle_t              Mipv6HA_tunnelInterfaceHandle,
      NPF_IN NPF_uint32_t                n_entries,
      NPF_IN NPF_MIPv6HA_SubtunnelHandle_t  *subtunnelHandleArray);
```

Description

This function allows for the retrieval of the packet statistics of a single or multiple BC entries on one MIPv6HA tunnel identified by the NPF_ifHandle_t interface handle.

Input Parameters

- cbHandle: the registered callback handle.
- cbCorrelator: the application's context for this call.
- errorReporting: the desired level of feedback.
- Mipv6HA_tunnelInterfaceHandle: Ifhandle of MIPv6HA tunnel interface
- n_entries: the number of BC entries queried
- *subtunnelHandleArray: array of subtunnel handles to be queried

Synchronous Return Codes

- NPF_NO_ERROR – The operation is in progress.
- NPF_E_UNKNOWN – The operation could not be completed due to problems encountered when handling the input parameters.
- NPF_E_BAD_CALLBACK_HANDLE – The callback handle is not valid.

Asynchronous response

A callback of type NPF_MIPV6HA_BC_ENTRY_STATS_GET is generated in response to this function call. A total of n_entries asynchronous responses (NPF_MIPv6HA_AsyncResponse_t) will be passed to the callback function, in one or more invocations. Each response contains the interface handle of the MIPv6HA tunnel interface and the associated status code. If the error code indicates success, the union in the callback response structure contains a pointer to the NPF_MIPv6HA_BC_Entry_Stats_t structure for the BC entry.

The following status codes may be returned.

- NPF_NO_ERROR – The operation was successful.
- NPF_E_UNKNOWN – The operation could not be completed. Reason unspecified.
- NPF_MIPv6HA_E_INVALID_SUBTUNNEL_HANDLE - Invalid or unknown tunnelhandle
- NPF_MIPV6HA_E_INVALID_IF_HANDLE: Ifhandle is null or invalid, or is not a Mipv6HA tunnel interface.

5.4.6 NPF_Mipv6HA_ProxyND_AddressAdd

```
NPF_error_t NPF_ Mipv6HA_ProxyND_AddressAdd (
      NPF_IN NPF_callbackHandle_t        cbHandle,
      NPF_IN NPF_correlator_t            cbCorrelator,
      NPF_IN NPF_errorReporting_t        errorReporting,
      NPF_IN NPF_ifHandle_t              homelinkInterfaceHandle,
      NPF_IN NPF_uint32_t                n_addresses,
      NPF_IN NPF_Ipv6Address_t           *Ipv6AddressArray);
```

Description

This function allows for the addition of a single or multiple addresses to the MIPv6HA ND Proxy function on a home link interface indicated by the `NPF_ifHandle_t` interface handle. The addition of an already existing address will result in a reset of the ProxyND functions performed on the address. In particular the status of the address will be set to PROBE and DAD will be initiated anew. **This is an optional function.**

Input Parameters

- `cbHandle`: the registered callback handle.
- `cbCorrelator`: the application's context for this call.
- `errorReporting`: the desired level of feedback.
- `homelinkInterfaceHandle:` Ifhandle of home link interface
- `n_adresses`: the number of addresses being configured
- `Ipv6AddressArray:` array of addresses entries

Synchronous Return Codes

- `NPF_NO_ERROR` – The operation is in progress.
- `NPF_E_UNKNOWN` – The operation could not be completed due to problems encountered when handling the input parameters.
- `NPF_E_BAD_CALLBACK_HANDLE` – The callback handle is not valid.
- `NPF_MIPv6HA_E_NOMEMORY` – The system was unable to allocate sufficient memory to complete this operation.
- `NPF_E_FUNCTION_NOT_SUPPORTED`: The function is not supported by the implementation.

Asynchronous response

A callback of type NPF_MIPV6HA_ProxyND_ADDR_ADD is generated in response to this function call. A total of n_entries asynchronous responses (NPF_MIPv6HA_AsyncResponse_t) will be passed to the callback function, in one or more invocations. Each response contains the interface handle of the home link interface, the ipv6address and the associated status code. The following status codes may be returned.

- `NPF_NO_ERROR` – The operation was successful.
- `NPF_E_UNKNOWN` – The operation could not be completed. Reason unspecified.
- `NPF_MIPv6_E_NOMEMORY` – The system was unable to allocate sufficient memory to complete this operation.

- NPF_MIPv6HA_E_INVALID_PARAM – Invalid IPv6 address parameters
- NPF_MIPV6HA_E_INVALID_IF_HANDLE: Ifhandle is null or invalid

5.4.7 NPF_Mipv6HA_ProxyND_AddressDelete

```
NPF_error_t NPF_ Mipv6HA_ProxyND_AddressDelete (
      NPF_IN NPF_callbackHandle_t        cbHandle,
      NPF_IN NPF_correlator_t            cbCorrelator,
      NPF_IN NPF_errorReporting_t        errorReporting,
      NPF_IN NPF_ifHandle_t              homelinkInterfaceHandle,
      NPF_IN NPF_uint32_t                n_addresses,
      NPF_IN NPF_Ipv6Address_t           *Ipv6AddressArray);
```

Description

This function allows for the deletion of a single or multiple addresses from the MIPv6HA ND Proxy function on a home link interface indicated by the NPF_ifHandle_t interface handle. **This is an optional function.**

Input Parameters

- cbHandle: the registered callback handle.
- cbCorrelator: the application's context for this call.
- errorReporting: the desired level of feedback.
- homelinkInterfaceHandle: Ifhandle of home link interface
- n_adresses: the number of addresses being deleted
- Ipv6AddressArray: array of addresses entries

Synchronous Return Codes

- NPF_NO_ERROR – The operation is in progress.
- NPF_E_UNKNOWN – The operation could not be completed due to problems encountered when handling the input parameters.
- NPF_E_BAD_CALLBACK_HANDLE – The callback handle is not valid.
- NPF_E_FUNCTION_NOT_SUPPORTED: The function is not supported by the implementation.

Asynchronous response

A callback of type NPF_MIPV6HA_BC_ProxyND_ADDR_DELETE is generated in response to this function call. A total of n_entries asynchronous responses (NPF_MIPv6HA_AsyncResponse_t) will be passed to the callback function, in one or more invocations. Each response contains the interface handle of the home link interface, the ipv6address and the associated status code. The following status codes may be returned.

- NPF_NO_ERROR – The operation was successful.
- NPF_E_UNKNOWN – The operation could not be completed. Reason unspecified.
- NPF_MIPv6HA_E_PROXYND_ADDR_UNKNOWN – Address not known

- NPF_MIPV6HA_E_INVALID_IF_HANDLE: Ifhandle is null or invalid

5.4.8 NPF_Mipv6HA_ProxyND_Flush

```
NPF_error_t NPF_ Mipv6HA_ProxyND_Flush(
      NPF_IN NPF_callbackHandle_t         cbHandle,
      NPF_IN NPF_correlator_t             cbCorrelator,
      NPF_IN NPF_errorReporting_t         errorReporting,
      NPF_IN NPF_ifhandle_t               homelinkInterfaceHandle);
```

Description

This function allows for the deletion of all addresses from the MIPv6HA ND Proxy function on a home link interface indicated by the NPF_ifHandle_t interface handle. **This is an optional function.**

Input Parameters

- cbHandle: the registered callback handle.
- cbCorrelator: the application's context for this call.
- errorReporting: the desired level of feedback.
- homelinkInterfaceHandle: Ifhandle of home link interface

Synchronous Return Codes

- NPF_NO_ERROR – The operation is in progress.
- NPF_E_UNKNOWN – The operation could not be completed due to problems encountered when handling the input parameters.
- NPF_E_BAD_CALLBACK_HANDLE – The callback handle is not valid.
- NPF_E_FUNCTION_NOT_SUPPORTED: The function is not supported by the implementation.

Asynchronous response

A callback of type NPF_MIPV6HA_BC_PROXYND_FLUSH is generated in response to this function call. A single asynchronous response (NPF_MIPv6HA_AsyncResponse_t) will be passed to the callback function. The response contains the interface handle of the home link interface and the associated status code. The following status codes may be returned.

- NPF_NO_ERROR – The operation was successful.
- NPF_E_UNKNOWN – The operation could not be completed. Reason unspecified.
- NPF_MIPV6HA_E_INVALID_IF_HANDLE: Ifhandle is null or invalid

5.4.9 NPF_Mipv6HA_ProxyND_AddrStateGet

```
NPF_error_t NPF_ Mipv6HA_ProxyND_AddrStateGet(
      NPF_IN NPF_callbackHandle_t         cbHandle,
      NPF_IN NPF_correlator_t             cbCorrelator,
      NPF_IN NPF_errorReporting_t         errorReporting,
      NPF_IN NPF_ifHandle_t               homelinkInterfaceHandle,
      NPF_IN NPF_uint32_t                 n_addresses,
      NPF_IN NPF_Ipv6Address_t            *Ipv6AddressArray);
```

Description

This function allows for the retrieval of the address state of a single or multiple addresses from the MIPv6HA ND Proxy function on a home link interface indicated by the `NPF_ifHandle_t` interface handle. **This is an optional function.**

Input Parameters

- `cbHandle`: the registered callback handle.
- `cbCorrelator`: the application's context for this call.
- `errorReporting`: the desired level of feedback.
- `homelinkInterfaceHandle`: Ifhandle of home link interface
- `n_adresses`: the number of addresses being queried
- `Ipv6AddressArray`: array of addresses entries

Synchronous Return Codes

- `NPF_NO_ERROR` – The operation is in progress.
- `NPF_E_UNKNOWN` – The operation could not be completed due to problems encountered when handling the input parameters.
- `NPF_E_BAD_CALLBACK_HANDLE` – The callback handle is not valid.
- `NPF_E_FUNCTION_NOT_SUPPORTED`: The function is not supported by the implementation.

Asynchronous response

A callback of type NPF_MIPV6HA_PROXYND_ADDR_STATE_GET is generated in response to this function call. A total of n_entries asynchronous responses (NPF_MIPv6HA_AsyncResponse_t) will be passed to the callback function, in one or more invocations. Each response contains the interface handle of the home link interface, the ProxyND entry, NPF_MIPv6HA_proxyND_entry_t, and the associated status code. The following status codes may be returned.

- `NPF_NO_ERROR` – The operation was successful.
- `NPF_E_UNKNOWN` – The operation could not be completed. Reason unspecified.
- `NPF_MIPv6HA_E_PROXYND_ADDR_UNKNOWN – Address not known`
- `NPF_MIPV6HA_E_INVALID_IF_HANDLE`: Ifhandle is null or invalid

5.4.10 NPF_Mipv6HA_BC_TableSpaceGet

```
NPF_error_t NPF_Mipv6HA_BC_TableSpaceGet(
      NPF_IN NPF_callbackHandle_t         cbHandle,
      NPF_IN NPF_correlator_t             cbCorrelator,
      NPF_IN NPF_errorReporting_t         errorReporting,
      NPF_IN NPF_ifhandle_t             Mipv6HA_tunnelInterfaceHandle);
```

Description

This function allows the application to retrieve an estimate of the remaining space available for BC entries on a MIPv6HA tunnel interface indicated by the NPF_ifHandle_t interface handle. The remaining space is estimated in terms of estimated number of entries. **This is an optional function.**

Input Parameters

- cbHandle: the registered callback handle.
- cbCorrelator: the application's context for this call.
- errorReporting: the desired level of feedback.
- Mipv6HA_tunnelInterfaceHandle: Ifhandle of Mipv6HA tunnel interface

Synchronous Return Codes

- NPF_NO_ERROR – The operation is in progress.
- NPF_E_UNKNOWN – The operation could not be completed due to problems encountered when handling the input parameters.
- NPF_E_BAD_CALLBACK_HANDLE – The callback handle is not valid.
- NPF_E_FUNCTION_NOT_SUPPORTED: The function is not supported by the implementation.

Asynchronous response

A callback of type NPF_MIPV6HA_BC_TABLE_SPACE_GET is generated in response to this function call. A single asynchronous response (NPF_MIPv6HA_AsyncResponse_t) will be passed to the callback function containing the status code, the interface handle and if successful the estimated number of how many BC entries that can yet be added to the MIPv6HA tunnel interface. The following status codes may be returned.

- NPF_NO_ERROR - The operation was successful.
- NPF_E_UNKNOWN - The operation could not be completed. Reason unspecified
- NPF_MIPV6HA_E_INVALID_IF_HANDLE: Ifhandle is null or invalid

5.4.11 NPF_Mipv6HA_BC_GetAll

```
NPF_error_t NPF_ Mipv6HA_BC_GetAll(
      NPF_IN NPF_callbackHandle_t         cbHandle,
      NPF_IN NPF_correlator_t             cbCorrelator,
      NPF_IN NPF_errorReporting_t         errorReporting,
      NPF_IN NPF_ifhandle_t             Mipv6HA_tunnelInterfaceHandle);
```

Description

This function allows for the retrieval of all tunnel identifiers from the BC structure set on a MIPv6HA tunnel interface indicated by the NPF_ifHandle_t interface handle. **This is an optional function.**

Input Parameters

- `cbHandle`: the registered callback handle.
- `cbCorrelator`: the application's context for this call.
- `errorReporting`: the desired level of feedback.
- `Mipv6HA_tunnelInterfaceHandle:` Ifhandle of Mipv6HA tunnel interface

Synchronous Return Codes

- `NPF_NO_ERROR` – The operation is in progress.
- `NPF_E_UNKNOWN` – The operation could not be completed due to problems encountered when handling the input parameters.
- `NPF_E_BAD_CALLBACK_HANDLE` – The callback handle is not valid.
- `NPF_E_FUNCTION_NOT_SUPPORTED`: The function is not supported by the implementation.

Asynchronous response

A callback of type `NPF_MIPV6HA_BC_GET_ALL` is generated in response to this function call. A single asynchronous response (NPF_MIPv6HA_AsyncResponse_t) will be passed to the callback function containing the status code, the interface handle and if successful a pointer to the NPF_MIPv6HA_tunnelHandle_array_t detailing the subtunnelidentifiers of the BC entries associated with the interface. The following status codes may be returned.

- `NPF_NO_ERROR -` The operation was successful.
- `NPF_E_UNKNOWN -` The operation could not be completed. Reason unspecified
- `NPF_MIPV6HA_E_INVALID_IF_HANDLE`: Ifhandle is null or invalid

5.4.12 NPF_Mipv6HA_ProxyND_TableSpaceGet

```
NPF_error_t NPF_Mipv6HA_ProxyND_TableSpaceGet(
      NPF_IN NPF_callbackHandle_t         cbHandle,
      NPF_IN NPF_correlator_t             cbCorrelator,
      NPF_IN NPF_errorReporting_t         errorReporting,
      NPF_IN NPF_ifhandle_t               homelinkInterfaceHandle);
```

Description

This function allows the application to retrieve an estimate of the remaining space available for ND proxy address entries on a home link interface indicated by the `NPF_ifHandle_t` interface handle. The remaining space is estimated in terms of estimated number of address entries. **This is an optional function.**

Input Parameters

- `cbHandle`: the registered callback handle.
- `cbCorrelator`: the application's context for this call.
- `errorReporting`: the desired level of feedback.
- `HomelinkInterfaceHandle: Ifhandle of home link interface`

Synchronous Return Codes

- NPF_NO_ERROR – The operation is in progress.
- NPF_E_UNKNOWN – The operation could not be completed due to problems encountered when handling the input parameters.
- NPF_E_BAD_CALLBACK_HANDLE – The callback handle is not valid.
- NPF_E_FUNCTION_NOT_SUPPORTED: The function is not supported by the implementation.

Asynchronous response

A callback of type NPF_MIPV6HA_NDPROXY_TABLE_SPACE_GET is generated in response to this function call. A single asynchronous response (NPF_MIPv6HA_AsyncResponse_t) will be passed to the callback function containing the status code, the interface handle and if successful the estimated number of how many address entries that can yet be added to the ND proxy function on the home link interface. The following status codes may be returned.

- NPF_NO_ERROR – The operation was successful.
- NPF_E_UNKNOWN – The operation could not be completed. Reason unspecified
- NPF_MIPV6HA_E_INVALID_IF_HANDLE: Ifhandle is null or invalid

5.4.13 NPF_Mipv6HA_ProxyND_GetAll

```
NPF_error_t NPF_ Mipv6HA_ProxyND_GetAll(
      NPF_IN NPF_callbackHandle_t         cbHandle,
      NPF_IN NPF_correlator_t             cbCorrelator,
      NPF_IN NPF_errorReporting_t         errorReporting,
      NPF_IN NPF_ifhandle_t               homelinkInterfaceHandle);
```

Description

This function allows for the retrieval of all addresses from the MIPv6HA ND Proxy function on a home link interface indicated by the NPF_ifHandle_t interface handle. **This is an optional function.**

Input Parameters

- cbHandle: the registered callback handle.
- cbCorrelator: the application's context for this call.
- errorReporting: the desired level of feedback.
- homelinkInterfaceHandle: Ifhandle of home link interface

Synchronous Return Codes

- NPF_NO_ERROR – The operation is in progress.
- NPF_E_UNKNOWN – The operation could not be completed due to problems encountered when handling the input parameters.
- NPF_E_BAD_CALLBACK_HANDLE – The callback handle is not valid.
- NPF_E_FUNCTION_NOT_SUPPORTED: The function is not supported by the implementation.

Asynchronous response

A callback of type `NPF_MIPV6HA_PROXYND_GET_ALL` is generated in response to this function call. A single asynchronous response (NPF_MIPv6HA_AsyncResponse_t) will be passed to the callback function containing the status code, the interface handle and if successful a pointer to the NPF_Ipv6AddressArray_t   detailing the addresses set in the Proxy ND structure of the interface. The following status codes may be returned.

- `NPF_NO_ERROR` – The operation was successful.
- `NPF_E_UNKNOWN -` The operation could not be completed. Reason unspecified
- `NPF_MIPV6HA_E_INVALID_IF_HANDLE`: Ifhandle is null or invalid

5.5 Order of Operations

Creation of the respective home link and MIPv6 HA tunnel interfaces (via Interface Management API) must precede the respective ND Proxy and BC function calls of the Mipv6 HA Service API. Similarly deletion (flushing) of ND proxy and BC entry structures must preceded deletion of the home link and the MIPv6 HA tunnel interface.

It should be noted that the DAD validation process can only complete when the home link interface is in operational status on.

6. References

[1]	NPF Software Implementation Agreement - Software API Conventions (Revision 2, September 2003)
[2]	IPv6 Unicast Forwarding Service API Implementation Agreement (Revision 2, June 2004)
[3]	IP Security Service API (August 2004)
[4]	Interface Management API (under revision)
[5]	Mobility Support in IPv6, RFC 3775
[6]	Using IPsec to Protect Mobile IPv6 Signaling Between Mobile Nodes and Home Agents, RFC 3776

7. API Capabilities

Function Name	Function Required
NPF_Mipv6HA_BC_Flush	Yes
NPF_Mipv6HA_BC_EntryAdd	Yes
NPF_Mipv6HA_BC_EntryDelete	Yes
NPF_Mipv6HA_BC_EntryAttrGet	Optional
NPF_Mipv6HA_BC_EntryStatsGet	Yes
NPF_Mipv6HA_ProxyND_Flush	Optional
NPF_Mipv6HA_ProxyND_AddressAdd	Optional
NPF_Mipv6HA_ProxyND_AddressDelete	Optional
NPF_Mipv6HA_ProxyND_AddrStateGet	Optional
NPF_Mipv6HA_BC_TableSpaceGet	Optional
NPF_Mipv6HA_BC_GetAll	Optional
NPF_Mipv6HA_ProxyND_GetAll	Optional
NPF_Mipv6HA_ProxyND_TableSpaceGet	Optional
NPF_MIPv6HA_RateLimitEvents	Optional

Event Name	Event Required
NPF_MIPv6HA_PROXYND_DAD_	Optional
NPF_MIPv6HA_BINDING_LIFETIME_EXPIRED	Optional
NPF_MIPv6HA_BC_ENTRY_MISS	Yes
NPF_MIPv6HA_SUBTUNNEL_ENDPOINT_AUTH_FAILED	Yes

APPENDIX A. NPF_MIPV6.H

```
/*
 * This header file defines typedefs, constants, and functions
 * for the MIPv6 SAPI
 */

#ifndef __NPF_MIPv6_0_H__
#define __NPF_MIPv6_0_H__

#ifdef __cplusplus
extern "C" {
#endif

#include "npf.h"

/*
 * NPF MIPv6 Error type
 */
typedef NPF_uint32_t NPF_MIPv6HA_ErrorType_t;


/*
 * Tunnel handle. A unique identifier selected by the implementation
 */
typedef NPF_uint32_t   NPF_MIPv6HA_SubtunnelHandle_t;

/*
 *        Tunnel Identifiers
 *
 */
typedef struct {
  NPF_MIPv6HA_SubtunnelHandle_t          subtunnelhandle;
  NPF_IPv6Address_t                      HoA;
} NPF_MIPv6HA_SubtunnelIdentifiers_t;

/*
 * Tunnel Identifiers Array
 */
typedef struct {
  NPF_uint32_t                              nCount;
  NPF_MIPv6HA_SubtunnelIdentifiers_t          *subtunnelidentifiersarray;
} NPF_MIPv6HA_SubtunnelIdentifiersArray_t;



/*
```

```
 * BC entry Lifetime Mode
 */
typedef enum {
  NPF_MIPv6HA_LIFETIME_ON  =1, /* Lifetime (secs) monitoring on */
  NPF_MIPv6HA_LIFETIME_OFF =2 /* Lifetime (secs) monitoring off */
} NPF_MIPv6HA_LifetimeMode_t;

/*
 *  MIPv6HA Binding Cache Entry
 */
typedef struct {
  NPF_MIPv6HA_SubtunnelHandle_t        subtunnelhandle;
  NPF_IPv6Address_t                    HoA;
  NPF_IPv6Address_t                    CoA;
  NPF_uint16_t                         PMTU;
  NPF_MIPv6HA_LifetimeMode_t           lifetimemode;
  NPF_uint32_t                         bindinglifetime;
  NPF_uint32_t                         remainingbindinglifetime;
} NPF_MIPv6HA_BC_Entry_t;

/*
 * BC Entry Packet Statistics
 */
typedef struct {
  NPF_MIPv6HA_SubtunnelHandle_t     subtunnelhandle;
  NPF_IPv6Address_t                 HoA;
  NPF_uint64_t                      mipv6NodeHCInOctets;
  NPF_uint64_t                      mipv6NodeHCInPkts;
  NPF_uint64_t                      mipv6NodeHCOutOctets;
  NPF_uint64_t                      mipv6NodeHCOutPkts;
  NPF_uint64_t                      dropRxOctets;
  NPF_uint64_t                      dropRxPkts;
  NPF_uint64_t                      dropTxOctets;
  NPF_uint64_t                      dropTxPkts;
} NPF_MIPv6HA_BC_EntryStats_t;

/*
 * MIPv6HA Proxy ND address entry
 */
typedef struct {
  NPF_IPv6Address_t              Addrs;
  NPF_IPv6AddrState_t            AddrsState;
} NPF_MIPv6HA_ProxyND_Entry_t;

/*
```

```
 *       IPv6 address array
 */
typedef struct {
  NPF_uint32_t                  nCount;
  NPF_IPv6Address_t            *Ipv6AddressArray;
} NPF_IPv6AddressArray_t;

/*
 * Mipv6 HA completion callback types
 */
typedef enum {
  NPF_MIPV6HA_BC_ENTRY_ADD                        =1,
  NPF_MIPV6HA_BC_ENTRY_DELETE                     =2,
  NPF_MIPV6HA_BC_FLUSH                            =3,
  NPF_MIPV6HA_BC_ENTRY_ATTR_GET                   =4,
  NPF_MIPV6HA_BC_ENTRY_STATS_GET                  =5,
  NPF_MIPV6HA_PROXYND_ADDR_ADD                    =6,
  NPF_MIPV6HA_PROXYND_ADDR_DELETE                 =7,
  NPF_MIPV6HA_PROXYND_FLUSH                       =8,
  NPF_MIPV6HA_PROXYND_ADDR_STATE_GET              =9,
  NPF_MIPV6HA_BC_TABLE_SPACE_GET                  = 10,
  NPF_MIPV6HA_BC_GET_ALL                          =11,
  NPF_MIPV6HA_PROXYND_TABLE_SPACE_GET             =12,
  NPF_MIPV6HA_PROXYND_GET_ALL                     =13,
  NPF_MIPv6HA_RATE_LIMIT_EVENTS                   =14
} NPF_MIPv6HA_CallbackType_t;

/*
 *       MIPv6 HA Event Type
 */
typedef enum {
   NPF_MIPv6HA_PROXYND_DAD                          = 1,
   NPF_MIPv6HA_BINDING_LIFETIME_EXPIRED             = 2,
   NPF_MIPv6HA_BC_ENTRY_MISS                        = 3,
   NPF_MIPv6HA_TUNNEL_ENDPOINT_AUTH_FAILED          = 4
} NPF_MIPv6HA_Event_t;

typedef struct {
  NPF_MIPv6HA_ErrorType_t                 errorCode;
  union{
   NPF_IfHandle_t                          ifhandle;
   NPF_uint32_t                            unused;
  } u1;
  union {
   NPF_MIPv6HA_SubtunnelIdentifiers_t     subtunnelIdentifiers;
   NPF_MIPv6HA_BC_Entry_t                  BCentry;
   NPF_MIPv6HA_BC_EntryStats_t             BCentrystats;
```

```
    NPF_IPv6Address_t                        proxyND_Address;
    NPF_MIPv6HA_ProxyND_Entry_t              proxyND_Entry;
    NPF_MIPv6HA_SubtunnelIdentifiersArray_t  *subtunnelIdentifiersArray;
    NPF_IPv6AddressArray_t                   *Ipv6Address;
    NPF_uint32_t                             entryspaceremaining;
    NPF_MIPv6HA_Event_t                      eventtype;
    NPF_uint32_t                             unused;
  } u2;
} NPF_MIPv6HA_AsyncResponse_t;

/*
 * MIPv6HA Completion Callback Data
 */
typedef struct {
  NPF_MIPv6HA_CallbackType_t       type;
  NPF_boolean_t                    allOK;
  NPF_uint32_t                     numResp;
  NPF_MIPv6HA_AsyncResponse_t      *resp;
} NPF_MIPv6HA_CallbackData_t;

typedef struct {
  NPF_IfHandle_t               homelinkInterfaceHandle;
  NPF_IPv6Address_t            ucAddrs;
  NPF_IPv6AddrState_t          ucAddrsState;
} NPF_MIPv6HA_ProxyND_DAD_t;

typedef struct {
  NPF_IfHandle_t                  Mipv6HAinterfacehandle;
  NPF_MIPv6HA_SubtunnelHandle_t   subtunnelHandle;
  NPF_IPv6Address_t               HoA;
} NPF_MIPv6HA_BindingLifetimeExpired_t;

typedef struct {
  NPF_IPv6Address_t        packetAddr;
} NPF_MIPv6HA_BC_EntryMiss_t;

typedef struct {
  NPF_IfHandle_t                  Mipv6HA_tunnelInterfaceHandle;
  NPF_MIPv6HA_SubtunnelHandle_t   subtunnelhandle;
  NPF_IPv6Address_t               HoA;
  NPF_IPv6Address_t               packetSrcAddr;
  NPF_IPv6Address_t               packetDstAddr;
  NPF_uint8_t                     repCount;
} NPF_MIPv6HA_SubtunnelEndpointAuthFailed_t;

/*
 *       MIPv6 HA Event Data Structure
 */
typedef struct NPF_MIPv6HA_EventData {
  NPF_MIPv6HA_Event_t                                 eventType;
  union {
     NPF_MIPv6HA_ProxyND_DAD_t                        proxyND_DAD;
     NPF_MIPv6HA_BindingLifetimeExpired_t             bindingLftExp;
     NPF_MIPv6HA_BC_EntryMiss_t                       BC_entryMiss;
     NPF_MIPv6HA_SubtunnelEndpointAuthFailed_t   subtnlEndpAuthFail;
  } u;
} NPF_MIPv6HA_EventData_t;

/*
 *       MIPv6 HA Event Array
 */
typedef struct NPF_MIPv6HA_EventArray {
  NPF_uint16_t                    n_data;
  NPF_MIPv6HA_EventData_t      *eventdata;
} NPF_MIPv6HA_EventArray_t;

/*
 *       MIPv6 HA event bitmask used in the event registration call
 */
typedef NPF_uint32_t NPF_MIPv6HA_EventMask_t;

/*
 * The following values can be set for the MIPv6HAEventMask
 */

#define NPF_IPSEC_EVENT_ALL_DISABLE                     ( 0) /* disable all */
#defineNPF_MIPv6HA_PROXYND_DAD_CONCLUDED               (1 << 0)
#defineNPF_MIPv6HA_BINDING_LIFETIME_EXPIRED            (1 << 1)
#defineNPF_MIPv6HA_BC_ENTRY_MISS                      (1 << 2)
#defineNPF_MIPv6HA_SUBTUNNEL_ENDPOINT_AUTH_FAILED   (1 << 3)

#define NPF_MIPv6HA_EVENT_ALL_ENABLE                     0xFFFFFFFF


/*
 *       MIPv6 HA Rate Limiting Events
 */
typedef enum {
  NPF_MIPv6HA_EVENT_LIMIT_TIME=1,          /* Time base limiting */
  NPF_MIPv6HA_EVENT_LIMIT_COUNT=2        /* Counter base limiting */
} NPF_MIPv6HA_EventLimitType_t;

typedef struct {
  NPF_MIPv6HA_Event_t                eventid;                /* Event HANDLE */
  NPF_MIPv6HA_EventLimitType_t    limitType;            /* Limit type */
  union
  {
    NPF_uint32_t numPerSec; /* Event frequency in time */
    NPF_uint32_t nCount;    /* Generate 1 event for */
                            /* every nCount encounters */
  }u;
} NPF_MIPv6HA_EventLimit_t;


/*
 *      Asynchronous error codes (returned in function callbacks)
 */

/*
 *      MIPv6HA reserved error codes in relation to other NPF APIs
 *      Note** The maximum range is 100
 */
#define NPF_MIPv6HA_BASE_ERR XXX /* Base value of XXX wrt other NPF codes */

/* Optional feature not supported */
#define NPF_MIPv6HA_E_OPTIONAL_FEATURE_NOT_SUPPORTED \
        ((NPF_MIPv6HA_ErrorType_t) NPF_MIPv6HA_BASE_ERR + 1)

/* System was unable to allocate sufficient memory to complete operation */
#define NPF_MIPv6HA_E_NOMEMORY  ((NPF_MIPv6HA_ErrorType_t)
NPF_MIPv6HA_BASE_ERR+2)

/* The Interface handle provided was not recognized as being valid */
#define NPF_MIPv6HA_E_INVALID_IF_HANDLE \
        ((NPF_MIPv6HA_ErrorType_t) NPF_MIPv6HA_BASE_ERR + 3)

/* Ipv6 ND Proxy Address not found*/
#define NPF_MIPv6HA_E_PROXYND_ADDR_UNKNOWN\
        ((NPF_MIPv6HA_ErrorType_t) NPF_MIPv6HA_BASE_ERR + 4)

/* Invalid parameter */
#define NPF_MIPv6HA_E_INVALID_PARAM\
        ((NPF_MIPv6HA_ErrorType_t) NPF_MIPv6HA_BASE_ERR + 5)

/* Duplicate HoA*/
#define NPF_MIPv6HA_E_DUPLICATE_HOA \
        ((NPF_MIPv6HA_ErrorType_t) NPF_MIPv6HA_BASE_ERR + 6)

/* Invalid or unknown tunnel handle - */
#define NPF_MIPv6HA_E_INVALID_SUBTUNNEL_HANDLE \
        ((NPF_MIPv6HA_ErrorType_t) NPF_MIPv6HA_BASE_ERR + 7)
```

```
/* Duplicate event id */
#define NPF_MIPv6HA_E_DUPLICATE_EVENT_ID \
        ((NPF_MIPv6HA_ErrorType_t) NPF_MIPv6HA_BASE_ERR + 8)

/* Invalid or unknown event ID  - */
#define NPF_MIPv6HA_E_INVALID_EVENT_ID \
        ((NPF_MIPv6HA_ErrorType_t) NPF_MIPv6HA_BASE_ERR + 9)

typedef void (*NPF_MIPv6HA_CallbackFunc_t)(
        NPF_IN NPF_userContext_t                userContext,
        NPF_IN NPF_correlator_t                 correlator,
        NPF_IN NPF_MIPv6HA_CallbackData_t   MIPv6HACallbackData);


/* Callback registration */
NPF_error_t NPF_MIPv6HA_Register(
            NPF_IN NPF_userContext_t                userContext,
            NPF_IN NPF_MIPv6HA_CallbackFunc_t      MIPv6HACallbackFunc,
            NPF_OUT NPF_callbackHandle_t              *MIPv6HA_CallbackHandle);

NPF_error_t NPF_MIPv6HA_Deregister(
            NPF_IN NPF_callbackHandle_t                 MIPv6HACallbackHandle);

typedef void (*NPF_MIPv6HA_EventCallFunc_t)(
                        NPF_IN NPF_userContext_t              userContext,
                        NPF_IN NPF_MIPv6HA_EventArray_t    eventArray);

NPF_error_t NPF_MIPv6HA_EventRegister(
                NPF_IN NPF_userContext_t                      usercontext,
                NPF_IN NPF_MIPv6HA_EventCallFunc_t        eventCallFunc,
                NPF_IN NPF_MIPv6HA_EventMask_t             eventmask,
                NPF_OUT NPF_callbackHandle_t                  *eventCallHandle);

NPF_error_t NPF_MIPv6HA_EventDeregister(
                                        NPF_IN NPF_callbackHandle_t cbHandle);

NPF_error_t NPF_MIPv6HA_RateLimitEvents (
                NPF_IN NPF_callbackHandle_t              cbHandle,
                NPF_IN NPF_correlator_t                  cbCorrelator,
                NPF_IN NPF_errorReporting_t               errorReporting,
                NPF_IN NPF_uint32_t                      countEventData,
                NPF_IN NPF_MIPv6HA_EventLimit_t    *eventLimitArray);


NPF_error_t NPF_MIPv6HA_BC_Entry_Add (
                NPF_IN NPF_callbackHandle_t             cbHandle,
                NPF_IN NPF_correlator_t                 cbCorrelator,
                NPF_IN NPF_errorReporting_t             errorReporting,
                NPF_IN NPF_IfHandle_t                    Mipv6HA_tunnelInterfaceHandle,
                NPF_IN NPF_uint32_t                      n_entries,
                NPF_IN NPF_MIPv6HA_BC_Entry_t            *BCentryarray);


NPF_error_t NPF_MIPv6HA_BC_Entry_Delete (
                NPF_IN NPF_callbackHandle_t                     cbHandle,
                NPF_IN NPF_correlator_t                         cbCorrelator,
                NPF_IN NPF_errorReporting_t                     errorReporting,
                NPF_IN NPF_IfHandle_t
Mipv6HA_tunnelInterfaceHandle,
                NPF_IN NPF_uint32_t                             n_entries,
                NPF_IN    NPF_MIPv6HA_SubtunnelHandle_t    *subtunnelHandleArray);


NPF_error_t NPF_MIPv6HA_BC_Flush (
                 NPF_IN NPF_callbackHandle_t            cbHandle,
                 NPF_IN NPF_correlator_t                cbCorrelator,
                 NPF_IN NPF_errorReporting_t            errorReporting,
                 NPF_IN NPF_IfHandle_t                  Mipv6HA_tunnelInterfaceHandle);

NPF_error_t NPF_MIPv6HA_BC_EntryAttrGet (
                NPF_IN NPF_callbackHandle_t                   cbHandle,
                NPF_IN NPF_correlator_t                        cbCorrelator,
                NPF_IN NPF_errorReporting_t                    errorReporting,
                NPF_IN NPF_IfHandle_t                        Mipv6HA_tunnelInterfaceHandle,
                NPF_IN NPF_uint32_t                            n_entries,
                NPF_IN NPF_MIPv6HA_SubtunnelHandle_t     *subtunnelHandleArray);


NPF_error_t NPF_MIPv6HA_BC_EntryStatsGet (
        NPF_IN NPF_callbackHandle_t                         cbHandle,
        NPF_IN NPF_correlator_t                              cbCorrelator,
        NPF_IN NPF_errorReporting_t                          errorReporting,
        NPF_IN NPF_IfHandle_t                                Mipv6HA_tunnelInterfaceHandle,
        NPF_IN NPF_uint32_t                                  n_entries,
        NPF_IN      NPF_MIPv6HA_SubtunnelHandle_t     *subtunnelHandleArray);


NPF_error_t NPF_MIPv6HA_ProxyND_AddressAdd (
        NPF_IN NPF_callbackHandle_t                cbHandle,
        NPF_IN NPF_correlator_t                    cbCorrelator,
        NPF_IN NPF_errorReporting_t                errorReporting,
        NPF_IN NPF_IfHandle_t                       homelinkInterfaceHandle,
        NPF_IN NPF_uint32_t                         n_addresses,
        NPF_IN NPF_IPv6Address_t                *Ipv6AddressArray);


    NPF_error_t NPF_MIPv6HA_ProxyND_AddressDelete (
                    NPF_IN NPF_callbackHandle_t      cbHandle,
                    NPF_IN NPF_correlator_t          cbCorrelator,
                    NPF_IN NPF_errorReporting_t      errorReporting,
                    NPF_IN NPF_IfHandle_t            homelinkInterfaceHandle,
                    NPF_IN NPF_uint32_t              n_addresses,
                    NPF_IN NPF_IPv6Address_t         *Ipv6AddressArray);


    NPF_error_t NPF_MIPv6HA_ProxyND_Flush(
                NPF_IN NPF_callbackHandle_t          cbHandle,
                NPF_IN NPF_correlator_t              cbCorrelator,
                NPF_IN NPF_errorReporting_t          errorReporting,
                NPF_IN NPF_IfHandle_t                homelinkInterfaceHandle);


    NPF_error_t NPF_MIPv6HA_ProxyND_AddrStateGet(
                NPF_IN NPF_callbackHandle_t    cbHandle,
                NPF_IN NPF_correlator_t        cbCorrelator,
                NPF_IN NPF_errorReporting_t    errorReporting,
                NPF_IN NPF_IfHandle_t          homelinkInterfaceHandle,
                NPF_IN NPF_uint32_t            n_addresses,
                NPF_IN NPF_IPv6Address_t       *Ipv6AddressArray);


    NPF_error_t NPF_MIPv6HA_BC_TableSpaceGet(
                NPF_IN NPF_callbackHandle_t       cbHandle,
                NPF_IN NPF_correlator_t           cbCorrelator,
                NPF_IN NPF_errorReporting_t       errorReporting,
                NPF_IN NPF_IfHandle_t             Mipv6HA_tunnelInterfaceHandle;


    NPF_error_t NPF_MIPv6HA_BC_GetAll(
                 NPF_IN NPF_callbackHandle_t      cbHandle,
                 NPF_IN NPF_correlator_t          cbCorrelator,
                 NPF_IN NPF_errorReporting_t      errorReporting,
                 NPF_IN NPF_IfHandle_t            Mipv6HA_tunnelInterfaceHandle);


    NPF_error_t NPF_MIPv6HA_ProxyND_TableSpaceGet(
                NPF_IN NPF_callbackHandle_t       cbHandle,
                NPF_IN NPF_correlator_t           cbCorrelator,
                NPF_IN NPF_errorReporting_t       errorReporting,
                NPF_IN NPF_IfHandle_t             homelinkInterfaceHandle);
```

```
NPF_error_t NPF_MIPv6HA_ProxyND_GetAll(
                    NPF_IN NPF_callbackHandle_t            cbHandle,
                    NPF_IN NPF_correlator_t                cbCorrelator,
                    NPF_IN NPF_errorReporting_t            errorReporting,
                    NPF_IN NPF_IfHandle_t                  homelinkInterfaceHandle);

#ifdef __cplusplus
}
#endif

#endif
```

APPENDIX B. NPF MIPV6 HA FRAMEWORK

The MIPv6 HA Service API is designed in accordance with how the MIPv6 HA node function is envisaged split in control and forwarding path functions as well as in accordance with how the management of the various forwarding path functions of a MIPv6 HA node are envisaged best distributed over the suite of Service APIs of the NP Forum. This Appendix serves to give an overview of these aspects.

Figure 1 is referred to for illustration. As in the rest of this document the specific MIPv6 HA processing entity (possibly multiple entities) in the Control Plane will here be referred to as the MIPv6 HA Service Layer Module.

B.1 MIPv6 HA Functional Split

B.1.1 Control Plane Functions

It is assumed that all functions associated with the processing of MIPv6 signaling messages reside in the Control Plane. This in particular means that the following data structures related to HA and MN Mobile IPv6 protocol signaling are assumed maintained and validated by the MIPv6 HA Service Layer module:

- Binding Cache
- HA lists
- Mobile Prefix Lists

The Binding Cache contains CoA and HoA address binding information[3] for the MNs currently served by the MIPv6 HA. Binding cache entries are created, modified and deleted on the basis of MIPv6 signaling messages interchanged in between the MIPv6 HA and mobile nodes.

HA and Mobile Prefix list are further discussed in Section B.2.3.

B.1.2 Forwarding Plane Functions

MIPv6 HA forwarding functions are assumed realized in the forwarding data layer of the node. Here by MIPv6 HA forwarding functions we refer to functions associated with payload traffic transiting the node. In particular it is assumed that the following functions are performed within the forwarding plane:

- Redirect of packets destined for MNs
- Reverse decapsulation and forward encapsulation of traffic from/to MNs
- IP interface processing functions

[3] In addition to the HoA and CoA binding information, each Binding Cache entry contains service attributes and status information including link-local address compatibility on/off, Key Management Mobility Capability on/off, sequence number of last BU received and lifetime of binding.

- IPSec decapsulation and encapsulation of traffic from/to MNs

Further it is considered a MUST that the API framework should support off load of the following functions to the forwarding data layer:

- MIPv6 HA ND proxy functions for MN packet intercept
- Transmittal of IPv6 Router Advertisements with the MIPv6 HA specific attributes and options

The overall functionality of these functions and their modeling over the Service API and the IM API boundary is described in Section B.3.

B.1.3 Receive and Transmit of MIPv6 HA signaling messages

A MIPv6 HA functions as a Mobile IPv6 Corresponding Node when transmitting and receiving Mobile IPv6 signaling messages. In order to insert the appropriate Router Headers in such packets, the Binding Cache must be consulted prior to the Forwarding Table on the transmit side and conversely, the Binding Cache Information is required to ensure correct processing of Home Address Destination options on the receive side. The Mobile IPv6 SAPI does not address this usage of the Binding Cache as these receive and transmit functions are assumed to recede in the forwarding plane.

B.2 MIPv6 HA Service Layer and Forwarding Plane Packet Transfer

B.2.1 MIPv6 signaling messages

The data part of Mobile IPv6 signaling messages interchanged in between Home Agents and Mobile Nodes are carried in the data part of ICMP messages or in the Mobility Header (MH) protocol fields of an IPv6 packet. In order to process Mobile IPv6 signaling messages correctly, the Mobile IPv6 Home Agent Service Layer module need not only access to the data part but also to the contents of various fields of the IPv6 header, more specifically to the contents of the IP source address and the (when present) Home Address Destination Option extension header field. Consequently it is vital that this information is conveyed to the MIPv6 HA Service Layer module together with the data parts.

Conversely when sending signaling messages as part of the Mobile IPv6 signaling protocol operation, the MIPv6 HA Service Layer module must specify part of the IP header format, such as in particular the addresses that should be placed in the destination address field and (for certain messages) in the Routing Header of type 2 Destination Option extension header field.

No assumptions are made on how exactly this is achieved as well as no specific API functions are assumed provided in this respect. It should be noted, however, that this may be achieved by using an advanced MIPv6 socket [7] (extension of raw IP socket) on top of an IP stack packetization process which in turn uses the PH API for transfer of packets to and from the forwarding plane.

B.2.2 IP Security Processing

IP Security processing is an integral part of Mobile IPv6. The Mobile IPv6 base specification [5] currently mandate certain signaling messages in between Mobile IPv6 Home Agent and Mobile Nodes to be protected by IP Security Transport mode, although it is acknowledged that alternative protection mechanism not based on IP Security may appear in the future.
It is assumed that the IP Security encryption and decryption processes for originating and terminating traffic generally will be instantiated in the data plane of an NP device. Consequently only clear text packets (and for inbound packets only IP security wise checked packets) are assumed passed in between the PH API and the forwarding plane. It should be emphasized, however, that this assumption has no significance for the NPF APIs as such, simply it have implications for the realization of the forwarding plane and the MIPv6 HA Service Layer module's usage of the PH API.

B.2.3 ICMP RA messages

HA list and Mobile Prefix list information must be sent unsolicited or solicited to mobile nodes at various occasions. The MIPv6 HA function collects the appropriate information maintained in these lists from RAs broadcasted on links on which the MIPv6 HA function is provided.

It is assumed that the functions associated with the collection and maintaining of these list are implemented in their completeness by the MIPv6 HA Service Layer module.

The NPF API framework does not provide any particular support for the retrieval of this information via API function calls apart from the basic packet transfer functionality provided by the PH API. Consequently it is assumed that for the purpose of these functions then the PH API is set up to transfer the RA messages to the Control Plane – to the MIPv6 HA Service Layer directly or, e.g. depending on the implementation, to an ICMP specific module which mediates the required information to the MIPv6 HA Service Layer module. Note that the RA messages may at the same time also be subject to non-MIPv6 processing in the forwarding plane or control plane, e.g., RA monitoring.

B.3 IM API and Service API Boundary Mapping

The following design prerequisites govern the distribution of the management of the various MIPv6 HA forwarding functions over the Service APIs and the Interface management API of the NP Forum, the split of functionality in between the IM API and the MIPv6 HA Service API in particular:

- Generic functions are handled via their generic APIs, this include generic interface functions, IP forwarding functions and IP Security functions.
- MIPv6 HA forwarding path functions, attributes and data structures that do not unequivocally belong within the scope of the IPv6 Unicast Forwarding Service API, the Interface Management API and the IP Security Service API are managed via the Mobile IPv6 Home Agent Service API.
- Data structures related to the Mobile IPv6 Home Agent operation on link local/interface level which are dynamic in nature, that is, which reflect run-time behavior are managed and instantiated by the MIPv6 HA Service Layer module by means of function calls

within the Mobile IPv6 Home Agent SAPI. This include, e.g., data structures governing MIPv6 HA ND proxy functions that must be instantiation on an interface due to the receipt of a valid Binding Updates.

- Data structures related to the Mobile IPv6 Home Agent function on an interface which are static in nature, that is, which are associated with the enabling/disabling and general configuration of the Mobile IPv6 Home Agent function on an interface, is managed and instantiated by means of the IM API.

B.3.1 IM API Modeling

B.3.1.1 Home Link Interface Modeling

An IPv6 interface on which the MIPv6 HA function is provided (here also denoted MIPv6 HA home link interface) is modeled as any other standard IPv6 interface. A MIPv6 HA home link interface must be configured with a global MIPv6 HA address as well as with the MIPv6 HA anycast addresses corresponding to the network prefixes for which the Home Agent function is provided. These addresses are configured in the normal manner.

The fact that a particular IPv6 interface (and IPv6 address respectively) is used as MIPv6 HA home link interface (and MIPv6 HA address respectively) will not be apparent from the interface type (address type respectively). A MIPv6 HA home link interface is identifiable only via is binding as the parent of a MIPv6 HA tunnel interface (see below) and an address of an IPv6 interface is identifiable only as a MIPv6 HA address via it usage as anchor source address in an MIPv6 HA tunnel interface as well as via its announcement as MIPv6 HA address in RAs (see below).

B.3.1.2 Home Agent Router Advertisements

The RAs sent from an IPv6 router serving as a MIPv6 Home Agent convey information relevant for the general IPv6 router functionality as well as particular information relevant for the Mobile IPv6 Home Agent functionality. The latter include the Global MIPv6 HA address and MIPv6 Home Agent preference.

The Mobile IPv6 HA specific attributes are envisaged specified via extensions to the RA function calls of the IM API.

B.3.1.3 MIPv6 Home Agent Tunnel Interface Modeling

The conceptual sub-tunnels in between the MIPv6 HA of a particular MIPv6 HA address and any of the MNs that it is serving (MNs currently away from home) is in the IM API represented by one common, enveloping MIPv6 HA IPv6-in-IPv6 tunnel interface. The tunnel interface is bound as a child to a MIPv6 Home Link interface and anchored with source address on the MIPv6 HA address of the MIPv6 Home Link interface.

The MIPv6 HA tunnel interface is a particular kind of an IPv6-in-IPv6 tunnel interface in that only the local source address (MIPv6 HA address) is specified via the IM API whereas the multiple remote sub-tunnel endpoints (the CoA of the respective MNs) are specified via the MIPv6 HA Service API. Packet processing on a MIPv6 HA tunnel interface must comply with the validation rules specified in [5]. These rules rely on the sub-tunnel information specified via the MIPv6 HA SAPI (the CoA and HoA bindings).

A Router will have a MIPv6 HA tunnel interface for every home link interface and MIPv6 HA address on and with which it is serving as MIPv6 HA.

The explicit modeling of the MIPv6 HA tunnel interface in the IM API serves a number of purposes:

1. In compliance with the existing IPv6 Unicast Forwarding SAPI ([2]), the MIPv6 HA tunnel interface can be used as egress interface in per MN host route entries in the FIB (more on this below)
2. In compliance with the existing IP Security SAPI ([3]), the MIPv6 HA tunnel interface can be used as interface anchor for IP Security Policies that should be in effect on the tunnel link in between the HA and a MN only. For more details see Section B.3.3.
3. The modeling of the MIPv6 HA tunnel interface as a child ip-in-ip tunnel interface of the Home Link interface is consistent with standard interface processing and demultiplexing procedures for incoming packets.
4. (Secondary) the modeling in the IM API allow for the set up of a particular FIB to be used for reverse decapsulated packets using the standard procedures in this respect.

MIPv6 HA tunnel interface common attributes and statistics such as MIPv6 HA address, MTU mode and packet counts are set and managed via the IM API, whereas per BC binding Cache/per MIPv6 HA MN conceptual sub-tunnel specific attributes and statistics such as, e.g., outer destination address per MN (MN CoA address), PMTU to each MN destination and packet statistics per MN destination are set up and managed via the MIPv6 HA SAPI. This reflects the split in between the static and the dynamic nature of these.

The MIPv6 HA functionality relies on the MIPv6 HA tunnel interface being in operational mode on. In case the operator manually deletes the tunnel interface, care must be taken first to delete the associated BC entries.

B.3.2 Service API Boundary Modeling

B.3.2.1 MIPv6 HA Proxy ND for mobile nodes

In order to intercept packets for the MN otherwise destined to the Home link, a Mobile IPv6 HA must perform IPv6 ND proxying for all the MNs that it is serving.

DAD is an integral part of the NP Proxy function. Only after successful instantiation of the ND proxy for the MN Home Address (that is successful DAD on MN Home Address) can a Binding Update from the MN be accepted. Depending on the L-bit set in the initiating BU, ND Proxy (and DAD) should be performed on both the global MN address and the associated canonical link-local address or on the global MN address only.

The MIPv6 HA ND proxy functions require dynamic and real time updating of the set of MN addresses for which ND Proxy should be performed. The MIPv6 HA ND Proxy function operates locally on the MIPv6 HA Home Link interface. Due to dynamic considerations, however, the MIPv6 HA Proxy ND function itself is managed via the MIPv6 Service API. This

also because the only known use case for Proxy ND is the Mobile IPv6 Home Agent functionality.
Binding Cache and MIPv6 HA tunnel handling
The Binding Cache is assumed maintained by the MIPv6 HA Service Layer module and only the Binding Cache attributes which are required for packets processing or which may demand particular hardware support are assumed to propagate down to the forwarding plane. Of the latter only lifetime is considered.

The Binding Cache entries of the forwarding plane are managed and instantiated by the MIPv6 HA Service Layer module using the MIPv6 HA Service API.

The Binding Cache maintained pr MIPv6 HA address in the Service Layer in the forwarding plane maps down to a conceptual Binding Cache table per MIPv6 HA tunnel interface in which each Binding Cache entry represent the conceptual sub-tunnel link in between the MIPv6 HA and the particular MN.

Each Binding Cache entry/ each conceptual MIPv6HA MN sub-tunnel is allocated, and identified by, a 32-bit handle by the implementation. The usage of this handle over the Service API allow for more optimal implementations. Further, future revisions of the IPv6 Unicast Forwarding Service API could allow for the usage of this sub-tunnel handle as next hop reference.

B.3.2.2 Redirect function modeling

Packets destined for MN currently served by the MIPv6 HA should not be send out on the Home link network but must instead be directed to encapsulation by the appropriate MIPv6 HA tunnel interface implementation. This regardless of whether the packets are intercepted using Proxy ND on the Home link network or whether they have arrived at the MIPv6 HA router on a different link.

The tunnel redirect function may be implementing in different ways, e.g.:

- It may be realized as part of the ingress FIB look up by installing per MN address host route entries in the FIB each pointing towards the appropriate MIPv6 HA tunnel interface as the egress interface (this type of next hop entries are supported in [2]).
- It may be realized using egress filters on the Home link interface, thus redirected all the relevant packets to the MIPv6 HA tunnel interface(s).

In the first case, host route entries can be installed into the FIB in (again) different ways:

- The host route entries (MN address and MIPv6 HA tunnel handle) may be created/deleted via the Unicast forwarding SAPI and by the Unicast Forwarding SL module. This prompted by a request from the MIPv6 HA SL module.
- The host route entries may also be installed within the respective FIBs by the forwarding control layer transparently to the Unicast Forwarding SL module. This prompted by

actions taken by the MIPv6 HA SL module via the MIPv6 HA SAPI, e.g. the installment of new binding cache entries.[4]

Of the above, the model assumed is that the intercept function is realized using per MN host route entries in the FIB and that these are installed in the forwarding plane FIBs by the Unicast Forwarding SL module via the Unicast forwarding SAPI.

No explicit assumptions on how the appropriate information is conveyed to the Unicast Forwarding SL module by the MIPv6 HA SL module are made. It could be done by use of an intra-service layer API in between the two modules but this is left to the choice of the providers of the SL modules.

It should be noted that the installment a new per MN host route entry would need to be done not every time a MN changes location (CoA change) but only every time the MN either starts or ceases to be served by the MIPv6 HA.

Possibly the IPv6 Unicast Forwarding SAPI may be extended to support next hop types containing not simply the MIPv6HA tunnel interface handle, but instead the more specialized MIPv6 HA MN sub-tunnel handle provided by the MIPv6 HA SAPI implementation. Whether the usage of the latter is advantageous will depend on the exact implementation of the forwarding path.

In any case, that is, whether using per sub-tunnel handle next hops or per MIPv6 HA tunnel interface next hop are used, the MIPv6 HA tunnel module to which the packets are redirected must perform the encapsulation process. In case the enveloping MIPv6 HA tunnel interface is referenced as next hop, an additional look-up to retrieve the appropriate sub-tunnel (and the appropriate CoA) must be performed by the tunnel module. In case the individual sub-tunnel handles are referenced in the next hop structure of the forwarding table look up, the CoA may be indicated already in the next hop structure. The latter however require the FIB to be updated every time the MN changes its location and not only when it either starts or ceases to be served by the MIPv6 HA.

[4] The model is somewhat doubtful since the forwarding control layer may not possess the required information vis-à-vis virtual router boundaries and VPN boundaries to ensure that the entries are installed in the correct set of FIBs.

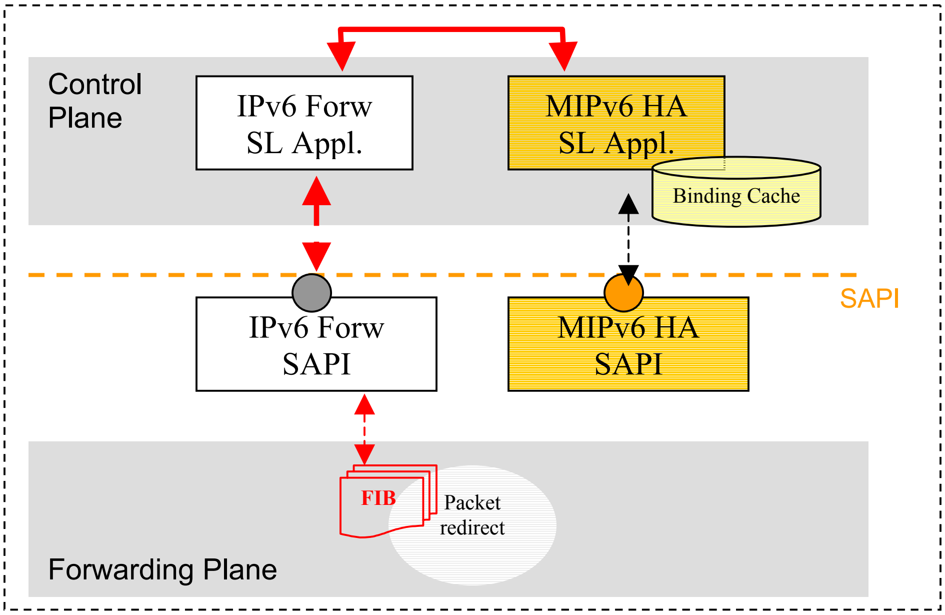

Figure 2 Redirect function modeling

B.3.3 Interaction with the IP Security API framework

Secure communication on the MIPv6 HA=MN tunnel path rely on tunnel IP Security SAs.

The associated IP Security tunnel SAs are anchored on the addresses of the Home Agent and the Home Addresses of the MN but the destination address of the outer header, that is, the present CoA of the MN, must be changed whenever a new BU (new CoA) for the MN in question is received (the latter without necessarily demanding renegotiation of the SA).
The associated security policy does not refer to the outer destination header and need not be changed while the MN moves around (CoA changes) however the policy should only be in effect for MNs with valid binding cache.

It is envisaged that the enabling and disabling of the required security policies as well as the dynamic updating of the involved SAs could be achieved by communication in between the MIPv6 HA SL module and the IP Security SL module (indeed this is the operation suggested by the IETF MIPv6 specifications, [5] and [6]). That is, prompted by the accept of a new BU the MIPv6 HA SL module shall request the IP Security SL module to take the appropriate actions via the IP Security SAPI.

No assumptions are made on how intra service layer communication is achieved not or which parties such communication may involve. Simply it is assumed that IP Security Policies and Security Associations required by the MIPv6 HA function are installed via the IP Security SAPI and managed by the client of the API, i.e., the IP Security SAPI SL module.

In the current IP Security SAPI framework Security Policies are bound to interfaces (via the SPD binding to interfaces).

The MIPv6 HA function relies on a peculiar interaction with IP Security in that it suggests certain IP Security Policies (ESP tunnel mode) to be set up on the individual tunnel interfaces, see [6], and prescribes this to result in that packets, which are captured by those IP Security Policies and which otherwise would have been sent unencrypted but encapsulated to the CoAs of the MNs, must *escape* the tunnel interface encapsulation and instead be sent according to the prescription of the IP Security Policies alone.

It is assumed that an implementation of the IP Security SAPI supporting the MIPv6 HA framework will support the setup of policies on MIPv6 HA tunnel interfaces in compliance with the above-required behavior. In this context the MIPv6 HA tunnel interfaces is referred to in the IP Security SAPI as a standard IPv6 interface.

In addition to IP Security tunnel mode required on the MIPv6HA=MN path, the MIPv6 HA also rely on the use of IP Security transport for direct communication in between the MN and the HA. The transport Security Policies and Security Associations are only related to the HoA of the MNs, not the CoA, and will thus not be directly affected by the moving around of the MNs. The transport security associations and policies are assumed set up via the IP Security SAPI in the standard manner.

Remark

The IP Security Architecture of the IETF is currently undergoing revision. In particular one is moving away from interface specific Security Policies. In an environment without interface specific Security Policies, the interaction with MIPv6 HA tunneling may be achieved using global security policies that filter on the MH protocol type value. Given that the IP Security SAPI of the NPF is enhanced to support policy selection on MH type values (or that the implementation support selection on MH type value by means of the existing SAPI calls of the IP Security SAPI[5]) the interaction with IP Security may be achieved with global policies which MH type specific selectors and without the setup of Security Policies anchored on the MIPv6 HA tunnel interface.

Further, the IETF is working on modifying IKE (IKEv2) so that IKE may be used to renegotiate/update IP Security SAs when the Mobile Nodes moves around. Potentially eliminating the need for interaction in between the MIPv6 HA function and the IP Security Control function for this purpose.

[5] An MH type selector could be specified via the existing NPF Security SAPI using the following semantics: In a NPF-IPSecSeclector with IP Transport Protocol the MH Protocol, the IP source port denotes the MH type whereas the IP destination port should be ignored.

APPENDIX C. ACKNOWLEDGEMENTS

Working Group Chair:

Alex Conta, Transwitch, aconta@txc.com

Task Group Chair:

Karen Nielsen, Ericsson, karen.e.nielsen@ericsson.com

The following individuals are acknowledged for their participation to the IPv6 Task Groups teleconferences, plenary meetings, mailing list, and/or for their NPF contributions used for the development of this Implementation Agreement. This list may not be all-inclusive since only names supplied by member companies for inclusion here will be listed. The NPF wishes to thank all active participants to this Implementation Agreement, whether listed here or not.

The list is in alphabetical order of last names:

Alex Conta, Transwitch

Suresh Krishnan, Ericsson

Karen E. Nielsen, Ericsson

Erik B. Pedersen, Ericsson

John Renwick, Agere Systems

Chirayu Shah, Ericsson

APPENDIX D. LIST OF COMPANIES BELONGING TO NPF DURING APPROVAL PROCESS

Agere Systems

Altera

AMCC

Analog Devices

Avici Systems

Cypress Semiconductor

Enigma Semiconductor

Ericsson

Erlang Technologies

ETRI

EZ Chip

Flextronics

HCL Technologies

Hifn

IBM

IDT

Infineon Technologies AG

Intel

IP Fabrics

IP Infusion

Kawasaki LSI

Motorola

NetLogic

Nokia

Nortel Networks

NTT Electronics

PMC Sierra

Seaway Networks

Sensory Networks

Sun Microsystems

Teja Technologies

TranSwitch

U4EA Group

Xelerated

Xilinx