AN API FOR IPV6 MULTIHOMING

Isaías Martínez-Yelmo
Alberto García-Martínez
Marcelo Bagnulo Braun
{imyelmo,alberto,marcelo}@it.uc3m.es
Departamento de Telemática
Universidad Carlos III de Madrid
Av. Universidad 30. 28911 Leganés
Madrid (Spain)

Abstract This paper proposes an API for Multihoming in IPv6. This API is based on the *Hash Based Addresses* and *Cryptographically Generated Addresses* approaches, which are being developed by the IETF multi6 Working Group. The support of Multihoming implies several actions such as failure detection procedures, reachability tests, re-homing procedures and exchange of locators. Applications can benefit from transparent access to Multihoming services only if per host Multihoming parameters are defined. However, more benefits could be obtained by applications if they will be able to configure these parameters. The proposed Multihoming API provides different functions to applications which can modify some parameters and invoke some functions related with the Multihoming Layer.

Keywords: API, Multihoming, IPv6

1. Introduction

Networked applications are fundamental tools in our daily lives. So, high available connections and resiliency [Yin and Twist, 2003] are usually required by enterprises and small-users. Providing this high availability is a difficult task because of the fact that network failures can always happen; thus, redundancy techniques such as Multihoming are needed. In IPv4, Multihoming is based on announcing all the prefixes of a site through all the providers using the Border Gateway Protocol (BGP). Therefore, if a connection fails, a new path could be found with another provider. Small-users never use BGP in their equipments because of the scalability problems of the protocol [Huston, 2001] and its complexity. So, Multihoming cannot be supported for small-users unless NAT-based boxes are used [Guo et al., 2004], with all the problems that

NATs present such as need for Application Level Gateways, disruption of end-to-end security, etc.

Some goals for Multihoming in IPv6 have been defined in [Abley et al., 2003], these goals are fault tolerance, load sharing, transport layer survivability and enhanced scalability for small-users. One of the proposed solutions combines Hash Based Addresses (HBA) [Bagnulo, 2004] and Cryptographically Generated Addresses(CGA) [Aura, 2004].

This paper proposes a Multihoming API for the approach based on HBA and CGA. It is structured as follows. In section 2, the Multihoming solution which is being developed actually by the IETF multi6 Working Group is presented. In section 3 a Multihoming API is described; the proposed API allows an ordered access to the Multihoming functionalities shown in the previous section. Future work is considered in section 4. Finally, conclusions about the presented work are exposed.

2. Multihoming in IPv6

Several roles are played by IPv4 addresses and IPv6 addresses:

- *Identifier:* IP addresses are passed to upper layers to be used as identifiers for the local and remote end points of a communication.
- *Locator:* IP addresses reflect the topological location of a host in the Internet
- *Forwarding Label:* Routers forward packets taking into account their destination IP address.

Due to this overload of roles, when a communication path suffers a failure, it is impossible to change the locator of the communication, even if this communication could be continued using another locator, because the identifier would also change and the Transport Level could not identify new packets as belonging to the same flow [Nordmark, 2004].

The proposal of the IETF multi6 Working Group is based on a new Multihoming Layer placed between the IP routing sub-layer and the IP end-point sublayer [Nordmark and Bagnulo, 2005]. This layer manages a set of locators corresponding to a given identifier.

A goal started by RFC 3582 [Abley et al., 2003] is to avoid introducing new vulnerabilities in the deployment of Multihoming. The requirement of mapping different locators for a single identifier enables new vulnerabilities like flooding and hijacking attacks [Nordmark and Li, 2005]. Therefore, it is necessary to prevent these attacks, so the solution should provide a secure mapping between identifiers and locators. The solution based on HBA and CGA provides this secure mapping; either there is a restriction in the locators to use (HBA approach), or there is a secure way of exchanging new locators based on signatures(CGA approach). Thus, it can be assured that the locators are associated with the identifier which is being used in the communication.

Other functionalities are needed for the Multihoming support; for instance, it is needed to change the actual locators of an ongoing communication, this is called re-homing procedure. Nevertheless, a re-homing procedure should only be needed when a failure has been happened; thus, it is necessary to check the communication through reachability tests. There are two options:

- *Bidirectionally operational address pair:* The locator pair used in the communications is the same in both directions.
- *Unidirectionally operational address pair:* The locator pair used in each direction of the communication is different.

For each case, a reachability test is proposed in [Arkko, 2004].

Finally, if reachability tests are not successful, working locators must be found; reachability tests with the new pairs of locators must be performed until a pair allows a new path to establish the communication. Once a pair of valid locators is found, a re-homing procedure can be executed for those locators.

After presenting the different functions required for Multihoming support, we will explain the CGA and HBA approaches.

2.1 CGA (Cryptographically Based Addresses)

The CGA approach provides the secure mapping between identifier and locators through asymmetric cryptography. A CGA is an IPv6 address which can be used as a predefined valid locator, and its interface ID [Hinden and Deering, 2003] is related to a public key. This relation is due to the fact that the interface ID is a hash (SHA-1 hash algorithm [SHA-1, 1995]) of the data structure showed in the Fig.1. The data structure is called *CGA Parameters Data Structure* and contains a modifier, a subnet prefix, a public key and an optional field; only the leftmost 64 bits of the hash of the structure forms the interface ID. Furthermore, there should be a CGA per each subnet prefix owned by the host if we want the information about the public key to be accessible from any subnet prefix.

The establishment of a communication could be done through the DNS service, since a CGA is a valid IPv6 address and it would be probably mapped with a name. If new locators are to be added for its later use, an exchange between the multihomed host and the other side of the communication must be performed. This exchange includes the new locators, the CGA Parameters Data Structure and a signature of the locators with the private key associated with the public key contained in the CGA Parameters Data Structure. The security of this process relies on the fact that the other side of the communication checks that the hash of the received CGA Parameters Data Structures matches with the interface ID of the other host; this means that the public key inside the received CGA Parameters Data Structure belongs to the other host and the signature of the locators can be checked. Thus, if the check of the signature is successful, it implies that the locators are valid.

Note that an attacker requires bruce force methods to obtain a new pair of private/public keys to obtain a hash of the CGA Parameters Data Structure equal to the interface ID of the host which is being attacked. If [Aura, 2004] is read carefully, we can see that the complexity of the attack is $O\left(2^{59}\right)$ and additional security can be added to the CGA by means of the Sec parameter. The *Sec* parameter is embedded in the interface ID (the 3 leftmost bits of the ID) and requires that an adjustable size part of the leftmost 112 bits of another hash (again a SHA-1 algorithm), named hash2, equals to zero. With this requirement, the complexity of a bruce-force attack is $O\left(2^{59+16*Sec}\right)$. It has to be taken into account the fact that this extra security is not without a cost, since hosts have to make $O\left(2^{16*Sec}\right)$ hash2 operations until a CGA Data Structure fits with the Sec parameter condition.

Nevertheless, CGA have a problem due to the fact that the computational cost of asymmetric public operations for signing the locators with the private key is very large.

2.2 HBA (Hash Based Addresses)

The idea of HBA [Bagnulo, 2004] is to avoid the computational cost of CGA. This is due to the fact that HBA does not need asymmetric key cryptography for the exchange of locators. The solution proposed in HBA is to include in a HBA Parameters Data Structure all the known subnet prefixes by a host. As in CGA, a hash operation is applied to the HBA Parameters Data Structure, but in this case information about the known prefixes by the host is included. The leftmost 64 bits are again the interface ID which will be added to the subnet prefix. This process must be applied to all subnet prefixes contained in the HBA Parameters Data Structure; thus, after a HBA set of addresses is obtained, only these addresses could be used as locators in a Multihoming environment.

The interface ID is now a container of information about the different subnet prefixes of the host and the verification process of the HBA is similar to the CGA case, but in this case the verified information is the subnet prefixes owned by the host and not a public key. Due to this last fact, only prefixes of the host are known. When a packet with an unknown source address is received, it must be checked that the prefix belongs to the set of prefixes contained in the HBA Parameters Data Structure, and that the locator address belongs to the HBA set generated from the same HBA Parameters Data Structure. This means making a hash of the HBA Parameters Data Structure with a subnet prefix equal to the

Modifier (16 octets)
Subnet Prefix (8 octets)
Collision Count (1 octet)
Public Key (Variable Length)
Extension Fields (Optional, Variable Length)

Figure 1. CGA Parameters Data Structure

Type	Length	P	Reserved
Prefix[1]			
Prefix[2]			
...			
Prefix[n]			

Figure 2. Multi-Prefix extension for CGA

received. If the leftmost 64 bits are equal to the interface ID, the new address belongs to the HBA set and it can be accepted as a valid locator.

The bruce-force attack complexity is the same as in CGA approach and a Sec parameter is also used to improve the security.

2.3 Solution based on HBA and CGA

The use of HBA has several advantages over CGA: public key cryptography is not needed, so a great amount of computational complexity is avoided and it is not needed to check any signature of the locators. However, the HBA approach has a drawback, recalculation of the HBA set is necessary if a new subnet prefix wants to be added. Usually, a host knows a priori the prefixes which must be managed by it, but in some environments, such as in mobility, prefixes are only known a posteriori.

Nevertheless, HBA and CGA can use a common format for compatibility reasons [Bagnulo, 2004]. This can be done including each assigned Prefix/64 as an extension of the CGA Parameters Data Structure; this is done using an extension for CGA and it is called Multi-Prefix extension (see Fig.2). The public key field of the CGA Parameters Data Structure can be a random number or a public key; so, if the HBA includes a public key, asymmetric cryptography can be used for the exchange of new locators as occurs for CGA.

3. API proposal for Multihoming in IPv6

Nowadays, there are hundreds of applications running on the Internet and it is not desirable to change them if a new network service is introduced. So, the Multihoming Layer should be transparent to legacy applications while providing extended functions to the IPv6 API. An approach to provide the needed transparency could be to rely on predetermined parameters for old applications

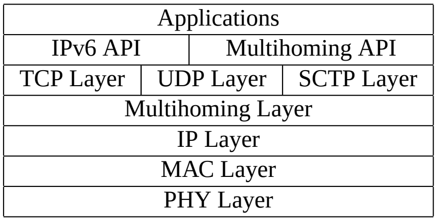

Figure 3. Proposed Multihoming API scheme

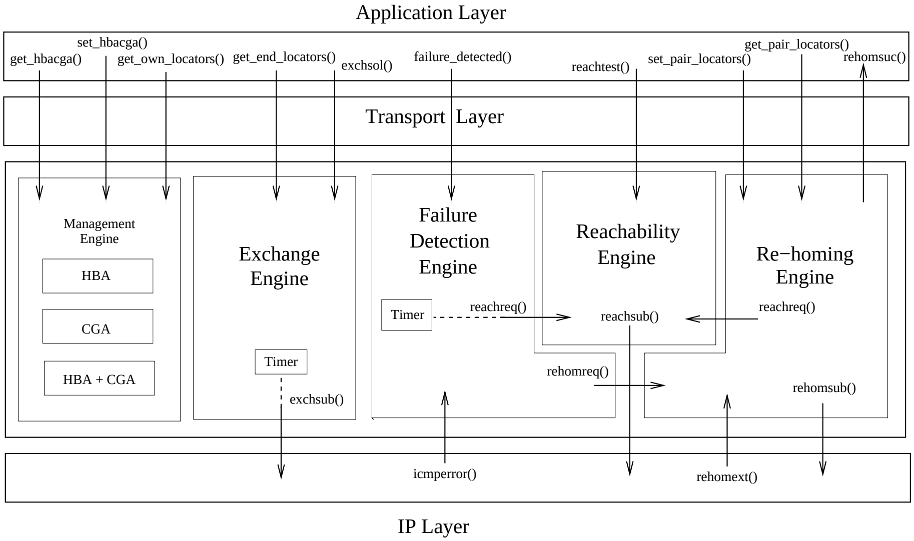

Figure 4. Multihoming API Scheme

and allow new applications to use the Multihoming API to get enhanced features.

A scheme of the proposal is shown in Fig.3. This scheme allows new applications to make use of new features provided by the Multihoming Layer. This approach is inspired in [Komu, 2004].

3.1 API Scheme

The API scheme proposed can be seen in Fig.4. In the figure we can see the different HBAs, CGAs and HBAs+CGAs which are managed by the Multihoming Layer. The HBA can only manage subnet prefixes due to the fact that the authentication is always performed doing the hash test from the extended CGA Parameters Data Structure and comparing the result with the interface ID of the HBA. CGAs and CGAs+HBAs can manage locators, not only prefixes as consequence of the signature used for the exchange of locators. Also there are a Failure Detection Engine, a Reachability Engine, a Re-homing Engine and a Exchange Engine.

The socket interface is usually used in Operative Systems to manage ongoing connections; so, the Multihoming API should provide most of its features through this socket interface.

3.2 Address Generation

First of all, it is needed the generation of HBAs and CGAs. These functions are not related with any outgoing communication, so socket interface can not be used and an external library must be provided. The proposed functions are:

void *create_cga(uint64_t prefix, void *extfields, int extlength, uint8_t sec, void *pk, int pklength). This function creates a CGA and its input parameters are:

prefix: The subnet prefix associated with the CGA which is introduced in the CGA Parameters Data Structure.
extfields: Optional extension fields can be passed with this parameter.
extlength: Length of the extension fields.
sec: This parameter specifies the value of the Sec parameter in the CGA which sets the level of security against bruce force attacks for impersonating a given interface identifier.
pk: With this parameter, public key of the CGA Parameters Data Structure is specified. If pk is equal to zero, an asymmetric key pair must be generated.
pklength: Length in bytes of the public key.

This function selects a random number for the modifier, create the CGA Parameters Data Structure and finally generates the CGA according to the requirements of the Sec parameter. So, the output parameters will be the *CGA Parameters Data Structure*, the *CGA* itself and the *private key* if *pk* was equal to zero.

void *create_hba(void *multiprefix, int multilenght, void *extfields, int extlength, uint8_t sec, void *pk, int pklength). This function creates a set of HBAs which are generated with the same extended CGA Parameters Data Structure. Input parameters are:

multiprefix: It is the Multi-Prefix extension for CGA. It contains the different subnet prefixes which will be used to generate the HBA set.
multilength: Length in bytes of the Multi-prefix extension.
extfields: Extensions fields can be included which are different from the Multi-Prefix extension.
extlength: Length in bytes of the extension fields.
sec: Again, the level of security is selected with this parameter.
pk: The public key of the HBA if it wants to be used the functionality of CGA+HBA.

pklength: Length in bytes of the public key.

The function must select a random number for the modifier and another random number if the pk parameter is equal to zero. Finally, a set of HBAs must be generated according to the proposed generation process in [Bagnulo, 2004] which implies that the hash2 must apply with the Sec parameter. A subnet prefix is not needed because a subnet prefix from the Multi-prefix-extension is used for each HBA. The output parameters will be the *set of CGAs* and the *extended CGA Parameters Data Structure*.

3.3 Management Engine

Applications can benefit from the identifiers and locators could be needed for applications. The selection of an addressing model, if several are available in a host, implies how the alternative locators are communicated to other hosts. There are four possibilities:

- **CGA:** Exchange of locators protected by asymmetric key. A large computational complexity is needed to perform this approach. The authenticity of the public key is checked with two Hash operations. Locators can be notified to other hosts at any time.

- **HBA:** Exchange of locators is based on Hash operations. Only the generated set of HBA allows re-homing procedures.

- **HBA+CGA:** HBAs Data structure is an extension of the CGA data structure, so both functionalities can be provided at the same time.

- **None:** The host is upgraded with Multihoming support but HBAs or CGAs are not available. Only Multihoming functions will be performed if the other host of the communication can manage HBAs and/or CGAs.

A mobile node should select CGA or HBA+CGA, or a web server should select HBA because it needs to perform a lot of tasks per minute and the use of CGA with asymmetric cryptography operations to sign the locators would reduce its performance.

Actual IPv6 API allows to obtain the different IPv6 addresses which have been assigned to the different interfaces in a host. Nevertheless, it is impossible to know if an IPv6 address is a HBA or a CGA by simple inspection. The Multihoming Layer knows this information, so the Multihoming API could provide selection mechanisms and ways to access to the type of addresses that is being used by a given communication. The functions which provide these features will be:

void *get_hbacga(int typeaddr). This function allows applications to retrieve the addresses managed by the Multihoming Layer. The *typeaddr* parameter indicates the type of wanted addresses (HBA, CGA or HBA + CGA). The output will be a list of different addresses with the selected type if they exists.

It could be thought that an address selection mechanism is needed, but this is not necessary because the IPv6 API already supports this feature with the function bind [Stevens and Thomas, 1998].

Furthermore, it is needed to inform at the Mutlihoming Layer about the different available HBAs and CGAs in the host which must be managed by it:

int set_hbacga(void ∗hbacga, int typeaddr). This function should be used by a superuser program to configure the HBAs and CGAs at the Multihoming Layer. This function could be executed in the starting sequence of a host. Input parameters are:

∗hbacga: A pointer to a list with the HBAs, CGAs or HBAs+CGAs.
typeaddr: Type of addresses in the list.

The output value will be an integer and if it equals to zero, the function will have finished successfully.

Locators are managed by the Multihoming Layer and they are not all known a priori; nevertheless, it could be useful to know the available locators for a connection in some situations such as debugging, administration and management or getting traces:

void *get_own_locators(int fdsocket). This function obtains the locators that the host can manage with the socket fdsocket. This is necessary because locators can change if the host is in a mobile environment. The input parameter is *fdsocket* which is the file descriptor of the socket used for the communication. The output parameter will be the list of locators.

3.4 Exchange Engine

Short connections could not benefit from re-homing capabilities; for instance, a request-reply communication in HTTP 1.0. Thus, a timer is defined and when this timer expires, locators are exchanged. This exchange is necessary before a failure interrupts the communication, because if locators are not exchanged, re-homing can not be done. Nevertheless, the application could know in advance the time which will be employed in the communication; so, it will be interesting to force a exchange of locators if a large connection is going to be used, or to prevent the exchange if the communication is going to be short:

int exchsol(int fdsocket, int time) This function of the proposed Multihoming API allows to force a exchange of locator. It has the *time* parameter and there are three possibilities:

minus than zero: The exchange is prevented.
equals to zero: The exchange of locators is done immediately.
minor than internal timer: The internal timer is set to the value of time.

Due to the fact that this engine manages the exchange of locators with the other side of the communication, it should provide this information to upper layers:

void *get_end_locators(int fdsocket) This function shows the locators of the other host in the communication and its parameters are the same as that the *getownlocators* function.

3.5 Failure Detection Engine

A Failure Detection Engine is necessary for detecting failures in communications. Three mechanisms are under study in the IETF multi6 Working Group: a failure may exists if an ICMP Destination Unreachable message is received, packets are not received in the interval of an inactivity TCP timer or applications notify about problems in their communications. So, it is necessary that the Multihoming API provides a function to inform the Multihoming layer about problems in upper layers:

int failure_detected(int fdsocket). This function allows upper layers to notify about problems in the communication and a reachability test must be requested (*reachreq()*). The output parameter will be an integer and if it equals to zero, the function will have finished successfully.

If a failure is detected, the Failure Detection Engine has to send a notification to the Re-homing Engine (*rehomreq()*).

3.6 Reachability Engine

The Reachability Engine performs reachability tests. If a notification is received, it verifies the communication with a reachability test that consists of sending a probe packet with a particular locator pair to confirm that a path is valid if a response is received.

int reachtest(int fdsocket, struct sockaddr my_addr, struct sockaddr end_addr, socklen_t addrlen). This function performs a reachability test and its parameters are:

fdsocket: File descriptor of the socket used for the communication.
my_loc: Source locator.
end_loc: Destination locator.
loclen: Length of locator.

The locators used for the reachability test are *my_loc* and *end_loc*. The output parameter will be an integer and if it equals to zero, the function will have finished successfully.

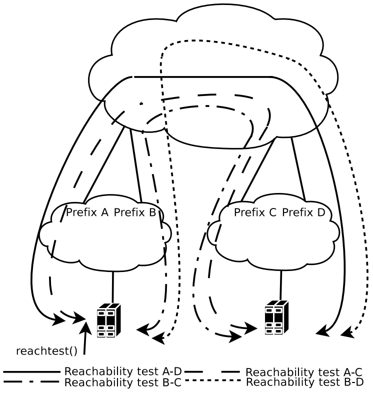

Figure 5. Reachability tests that can be performed

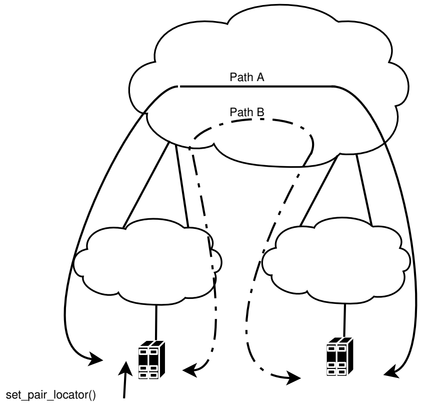

Figure 6. Rehoming of the communication between path A and B

Fig.5 shows the different paths which can be tested taking into account the pair of locators that can be formed. Only it is needed that one path provides connectivity.

3.7 Re-homing Engine

The Re-homing Engine manages re-homing procedures. It could be interesting for applications to obtain information about the occurrence of re-homing procedures. For instance, if an UDP application implements a slow-start algorithm as TCP, it could be interesting for the application to perform a slow-start algorithm after a re-homing procedure and try to obtain a fast adaptation to the new bandwidth.

rehomsuc() A signal is sent to applications to inform that a re-homing procedure has been performed by the other side of the communication.

int get_pairlocators(int fdsocket, struct sockaddr ∗my_loc, struct sockaddr ∗end_loc, socklen_t ∗loclen). This function obtains the pair of locators used in the ongoing communication. This request is solved by the Re-homing because last pair of locators used in a communication are always known after a re-homing procedure. Input parameters are:

fdsocket: File descriptor of the socket used for the communication.
∗*my_loc:* Pointer where the source locator will be saved.
∗*end_loc:* Pointer where the destination will be saved.
∗*loclen:* Pointer with space reserved to the locators. The function modifies its value to the length of locators.

The output parameter will be an integer and if it equals to zero, the function will have finished successfully.

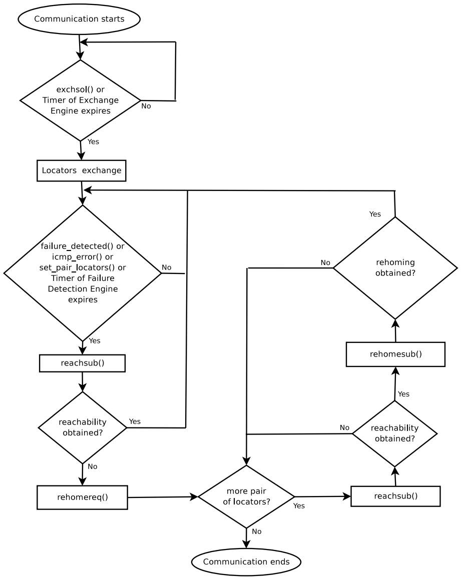

Figure 7. Basic organizational chart of Multihoming Layer

int set_pair_locator(int fdsocket, struct sockaddr my_addr, struct sockaddr end_addr, socklen_t addrlen). This function sets the locators for the ongoing communication; so, it is like a re-homing solicitation. Its structure is the same as the *reachtest* function. If my_loc and end_loc are equal to zero the locators will be selected by the Re-homing Engine. It must be noted that a reachability test should be performed before a re-homing procedure; thus, the Re-homing Engine has to submit a solicitation for a reachability test to the Reachability Engine (*reachreq()*). This function can be used to force a re-homing if, for example, the QoS obtained by an application is not enough; an example could be a videoconference where many frames are dropped.

Fig.6 illustrates a rehoming procedure. After the locators have been exchanged and a new pair of locators has been selected after performing a reachability test. Then, the communication path A can be changed to path B without breaking the established connection.

3.8 Organizational chart of Multihoming Layer

In Fig.7 a basic organizational chart of the Multihoming Layer is shown. The figure shows the basic functions that must be performed to obtain Multihoming support. First of all, it is necessary the exchange of locators. If other locators are not known, it is impossible to find an alternative path. A rehoming procedure can be performed if the ongoing communication has some type of problem (lost of connectivity, low quality, ...). At this point a new path must be found, so a reachability test is used to allow us to test the connectivity between locators of different hosts. Finally, if an alternative pair of locators is found, a rehoming procedure can be tried.

All the proposed functions in this section are necessary to provide a enhanced Multihoming Service to the applications and the needed tools for administration and management.

4. Future Work

Different tasks can be performed in the future taking into account the proposal of this paper. It is necessary a deep study of the default parameters which must be configured in the Multihoming Layer to provide reliable Multihoming support to legacy applications. Also, it should be interesting a comparation of the performance between legacy applications, legacy applications making use of Multihoming in transparent mode and applications using the Multihoming API. Furthermore, a large work in close relation with the multi6 Working Group should be done until an Multihoming API can be provided in the future.

5. Conclusions

HBA and CGA provide a secure mapping between identifiers and locators. In this way, stable identifiers are shown to the Transport Layers while the locators used in the IP datagrams can be changed without disrupting the communication. CGA has a much bigger computational complexity due to the fact that asymmetric key operations are needed. Conversely, HBAs have a low computational cost due to the fact that they only use hash operations. Nevertheless, in HBA the different prefixes which would be used by a host must be known in advance because this information is included in the interface ID of IPv6 addresses through a hash operation. CGA has not this limitation because the locator is authenticated through an asymmetric cryptographic scheme. New locators which are not known a priori can be sent because they are signed with the private key of the Host.

HBA and CGA have different features which can be used at the same time by different applications. The API proposed in this paper manages the mapping method between identifiers and locators. Because of this feature, applications can select HBA, CGA or CGA + HBA depending on their needs.

Furthermore, other functions are added to improve the failure detection and the re-homing procedure. With these functions, applications can inform about failures in the communications or a re-homing procedure can be requested by the application (for example, if the received QoS is not enough). These mechanisms can provide faster response if applications detect failures sooner than lower layers. Other functionalities are also added, such as the solicitation of an exchange of locators or the possibility for applications to be informed about changes of the locators related with the ongoing communication. Legacy applications can obtain Multihoming support if there exists a default configuration which configures the minimal needed parameters.

Acknowledgement

The authors would like to thank to Carlos J. Bernardos for the ideas provided. This work has been partly supported by the E-Next Project FP6506869 and OPTINET6 project TIC-2003-09042-C03-01.

References

Abley, J., Black, B., and Gill, V. (2003). *RFC 3582: Goals for IPv6 Site-Multihoming Architectures*.

Arkko, J. (2004). *Failure Detection and Locator Selection in Multi6*. Internet Draft draft-arkko-multi6dt-failure detection-00.txt (work in progress).

Aura, T. (2004). *Cryptographically Generated Addresses (CGA)*. Internet Draft draft-ietf-send-cga-06.txt (work in progress).

Bagnulo, M. (2004). *Hash Based Addresses (HBA)*. Internet Draft draft-ietf-multi6-hba-00.txt (work in progress).

Guo, Fanglu, Chen, Jiawu, Li, Wei, and Chiueh, Tzicker (2004). Experiences in building a multihoming load balancing system. In *IEEE INFOCOM '04, Hong Kong, China. Volume: 2*, pages 1241–1251.

Hinden, R. and Deering, S. (2003). *RFC 3513: Internet Protocol Version 6 (IPv6) Addressing Architecture*.

Huston, G. (2001). *RFC 3221: Commentary on Inter-Domain Routing in the Internet*.

Komu, M. (2004). Application programming interfaces for the host identity protocol. Master's thesis, Helsinky Universitiy of Technology.

Nordmark, E. (2004). *Multi6 Application Referral Issues*. Internet Draft draft-nordmark-multi6dt-refer-00.txt (work in progress).

Nordmark, E. and Bagnulo, M. (2005). *Multihoming L3 Shim Approach*. Internet Draft draft-ietf-multi6-l3shim-00.txt (work in progress).

Nordmark, E. and Li, T. (2005). *Threats relating to IPv6 multihoming solutions*. Internet Draft draft-ietf-multi6-multihoming-threats-03.txt (work in progress).

SHA-1 (1995). *Secure Hash Standard*. Federal Information Processing Standards. Publication 180-1.

Stevens, W. and Thomas, M. (1998). *RFC 2292: Advanced Sockets API for IPv6*.

Yin, S. and Twist, K. (2003). The coming era of absolute availability.