IKEv2 Mobility and Multihoming
(mobike)
Internet-Draft
Expires: September 6, 2006

U. Schilcher
Universitaet Klagenfurt
H. Tschofenig
F. Muenz
Siemens AG
S. Sugimoto
Ericsson

March 5, 2006

Application Programming Interface to a Trigger Database for MOBIKE
draft-schilcher-mobike-trigger-api-03.txt

Status of this Memo

Copyright Notice

Abstract

MOBIKE is a protocol which adds mobility and multihome support for IKEv2. MOBIKE peers continuously exchange a list of available IP addresses each other. A MOBIKE peer should have some information about the status of each address and interface in order to execute

the respective actions. Examples may comprise switching from one address or interface to another. This information, which will be referred as trigger, is distributed over a number of protocol daemons at an end host. To make this information available to a MOBIKE daemon, it is necessary to store it centrally at the host (called trigger database) and to enable the protocols to insert the triggers and to allow MOBIKE to obtain timely information.

Table of Contents

1. Introduction

When a MOBIKE daemon [1] is started it first has to build a set of all available addresses (or a subset of them for policy reasons; see [4]) before communicating with another peer. From these addresses, it has to select one of the addresses as the preferred address that will be used as the source address in the communication with the MOBIKE peer.

This address set together with the preferred address may change during operation because of several reasons, e.g. an interface is disconnected, a handover between two different link layer technologies takes place or the communication path becomes unavailable due to router failure. Many of the events, which cause the change of the address set, are out of the scope of the MOBIKE protocol itself but need an interaction with other protocols daemons locally at the end host.

In order to make MOBIKE working properly, it is really important to know about the status of the available addresses for making reasonable decisions. A number of other protocols running on the end host might have various information necessary to determine whether to switch from one preferred address to another or whether it is necessary to modify the peer address set.

An example is the IEEE 802.21 Media Independent Handover (MIH) standard [5], which is currently under development. The MIH is defined as a shim layer in the mobility-management protocol stack of both, the mobile node and the network elements, that provide mobility support. The MIH Function provides abstracted services to higher layers about the status and performance of any link layer technology.

To benefit from this information on higher layers, however, the MIH services must be combined with information from upper layers in order to facilitate a basis for decisions above network layer.

In this document, we therefore suggest to define an API that allows protocol daemons to insert information (triggers) about addresses and interfaces into a "database" that can later be made available to the MOBIKE daemon (or other protocols). Although, there have been some mechanisms [6][7] that can fulfill some of these requirements, we still need a flexible framework that can handle asynchronous event that takes place at L3 or below, including MIH events. The API will provide similar services to the MOBIKE daemon like MIH does for layer 3 and above. It is based on the BSD routing socket API in a similar fashion as PF_KEY [2] extends the same API for generic key management usage.

This document therefore heavily focuses on the functionality offered by the PF_KEY specification and uses the MIH Function as an example for retrieving necessary information for a decision making process.

Please note that the authors use the term 'database' in a symbolic way. It is a container for storing information about events. Information about the status of interfaces and addresses might not even be stored directly in this database and could well be implemented using a collection of pointers to the respective information.

2. Terminology

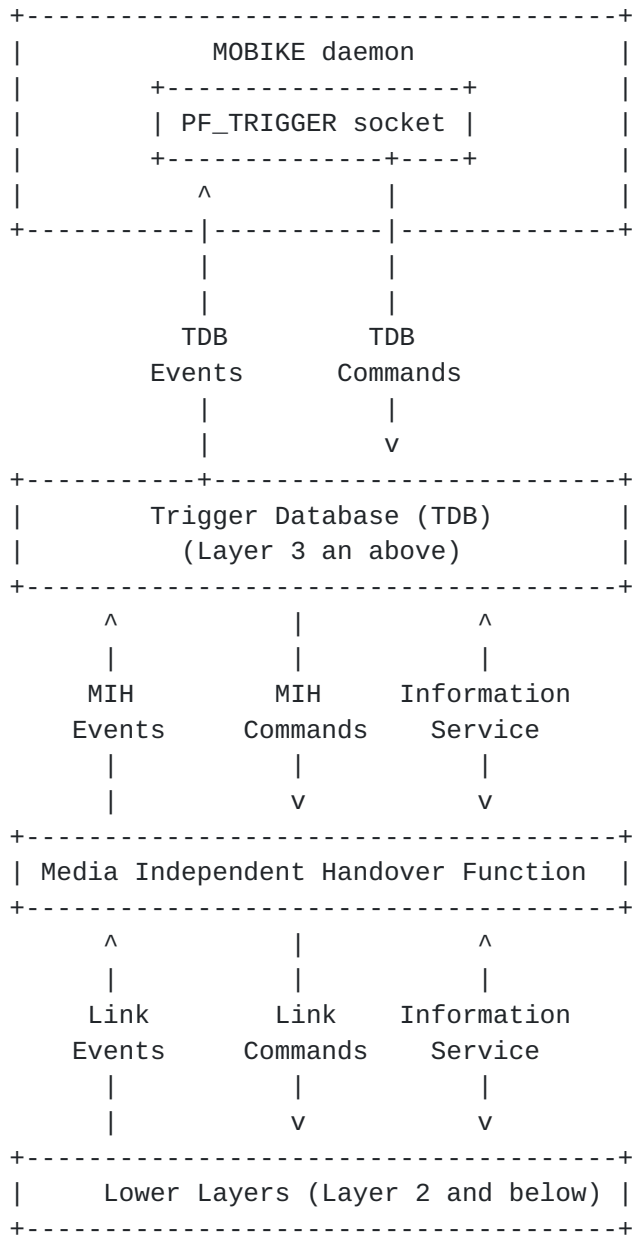
The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [3].

Additionally, the following terms are introduced:

- Trigger: Information which is relevant for MOBIKE about an address.
- Trigger Database (TDB): Collection of triggers which can be accessed via the API defined in this document.
- Media Independent Handover (MIH): Described in IEEE draft 802.21 [5] which is currently under development. The MIH Function provides abstracted services to higher layers about the status and performance of any link layer technology.

3. Trigger Database and MIH

The following section should give a brief overview for the interaction of Media Independent Handover (MIH) with Trigger Database (TDB) and MOBIKE daemon. The services of MIH are used e.g. for compiling relevant information about interface performance, which is then signalled through the Trigger Database to the MOBIKE daemon. Based on this data, the MOBIKE daemon may select a new preferred address or continues using the current one.



To receive event notifications from the MIH, the Trigger Database must perform two steps:

Capability discovery:

While the MIH provides many services, not all of them may be supported on a given platform. For learning, which of them are actually supported, the Trigger Database must query the MIH with a "MIH_Capability_Discover.request". The response, a "MIH_Capability_Discover.response" message will then indicate with a bit mask which services are supported. Alternatives and solutions for not supported but required services is done in a future version of this draft.

Service registration:

Now knowing the supported services, the Trigger Database must register to the services it is interested in with a "MIH_Event_Register.request" message. To confirm registration, the MIH answers with a "MIH_Event_Register.confirm" message and notifies the Trigger Database in case of status change of interfaces.

4. Trigger Classification

Many different events may cause a change in the address set used by MOBIKE (see [4]). These events can be issued by many different protocols running in kernel or user space. Since the reaction (if any) on a given event depends on the type of the event, a classification of these events is necessary.

As an example, we define the following triggers in this document:

Trigger type	Value	Description
TDB_TTYPE_IF_ADDED	1	New interface added
TDB_TTYPE_IF_REMOVED	2	Interface removed
TDB_TTYPE_IF_REMOVEDSOON	3	Interface is expected to be removed soon
TDB_TTYPE_IF_ADDRADDED	4	New address added to interface
TDB_TTYPE_IF_ADDRREMOVED	5	Address removed from interface
TDB_TTYPE_IF_ADDRCHANGED	6	Interface has changed one of its addresses (e.g. new DHCP lease)
TDB_TTYPE_TUNNEL_ADDED	7	IPSec tunnel was established
TDB_TTYPE_TUNNEL_CHANGED	8	IPSec tunnel conf. changed
TDB_TTYPE_TUNNEL_REMOVED	9	IPSec tunnel was removed
TDB_TTYPE_CONN_ESTABLISHED	10	e.g. dial-in network has connected
TDB_TTYPE_CONN_LOST	11	connection to network lost
TDB_TTYPE_DEST_UNREACHABLE	12	e.g. ICMP packet received
TDB_TTYPE_MAX	13	Maximum value for trigger types

The types TDB_TTYPE_TUNNEL_ADDED, TDB_TTYPE_TUNNEL_CHANGED and TDB_TTYPE_TUNNEL_REMOVED are inspired by [8]. Any listed trigger types will be signalled using the "tdb_trigger" message structure described in Section 7

Details about the supported message types and their formats can be found below:

TDB_TTYPE_IF_ADDED:

This message is signalled if a new interface comes up and directly refers to the "Link_Up.indication" event notification of the MIH Function.

TDB_TTYPE_IF_REMOVED:

This message is signalled if an interface is going down and directly refers to the "Link_Down.indication" event notification

of the MIH Function.

TDB_TTYPE_IF_REMOVEDSOON:

This message is signalled if an interface is expected (predicted) to go down within a certain time interval and directly refers to the "Link_Going_Down.indication" event notification of the MIH Function.

A future version of this document will add more triggers and a more detailed description of them.

5. API for the Trigger Database

To access the trigger database, an API is defined. For that purpose the new network protocol family ID PF_TRIGGER has to be defined. The operation of the API is analogue to the PF_KEY interface (see [2]).

To access the API, a socket of the family PF_TRIGGER has to be created. To communicate with the Trigger Database, messages are sent and received through the socket with the send() and recv() functions. Any other functions like bind(), connect(), etc. are not supported and MUST NOT have any effects on a socket of the PF_TRIGGER family.

The following exhibits a sample socket creation:

```
int s = socket(PF_TRIGGER, SOCK_RAW, PF_TRIGGER);
```

The format of the messages is the following: Each message starts with a fixed header. Appended to this header, there are some payloads depending on the type of the message. The available message types are described in Section 6.

Each time when a message is sent to the Trigger Database, it will respond with a message of the same type. This response contains the same payloads as transmitted to the Trigger Database, only some additional information MAY be included (e.g., the Trigger Database assigns an id to each trigger).

The normal operation works in the following way: A MOBIKE implementation, which wants to be informed about changes in the Trigger Database, registers itself to the Trigger Database by sending a TDB_REGISTER message.

If a protocol daemon wants to add, delete or modify an existing trigger it sends a TDB_ADD, TDB_DELETE or TDB_MODIFY message respectively to the Trigger Database including information that is important to add, delete or modify the trigger.

The Trigger Database acknowledges this message with a TDB_ADD, TDB_DELETE or TDB_MODIFY response to the network protocol and with a TDB_NOTIFY message to the registered MOBIKE implementation. This notify message contains information about the newly added, deleted or modified trigger including its ID. All available information about a trigger can be requested with a TDB_GET message.

If a MOBIKE implementation no longer wants to receive notifications for changes to the Trigger Database, it sends a TDB_DEREGISTER message.

In a future version of this document, we will try to add some information about scenarios to better illustrate the interaction.

6. Supported Message Types

Several different message types can be sent to the Trigger Database using a PF_TRIGGER socket. The message type is indicated by the tdb_header_msgtype field that is part of the generic message header (see Section 7) and can be one of the following values:

Message type	Value	Description
TDB_ADD	1	Add a trigger to the Trigger Database
TDB_GET	2	Get information about an existing trigger.
TDB_DELETE	3	Delete a trigger from the Trigger Database
TDB_REGISTER	4	Registers an application to receive messages for each new trigger added.
TDB_DEREGISTER	5	Deregisters an application from receiving messages for each new trigger added.
TDB_NOTIFY	6	A new trigger has been added, deleted or updated.
TDB_MODIFY	7	Modify a trigger in the Trigger Database
TDB_DUMP	8	Dump all Trigger Database entries
TDB_FLUSH	9	Delete all Trigger Database entries
TDB_MAX	10	Generic maximum for message types

Each message type requires different payloads to be appended. Each payload starts with a generic payload header followed by payload specific data. The generic header has the following structure:

```
struct tdb_payload {
        uint16_t        tdb_payload_len;
        uint16_t        tdb_payload_type;
} __attribute__( ( packed ) );
/* sizeof( struct tdb_payload ) == 4 */
```

The tdb_payload_len field contains the length of the payload divided by 8. The type of the payload is determined by the tdb_payload_type field, which contains one of the following values:

Payload type	Value	Description
TDB_PT_INTERFACE	1	Information about an interface
TDB_PT_ADDRESS	2	The IP address of an interface
TDB_PT_TRIGGER	3	Trigger id, type, etc.

Details about the supported message types and their formats can be found below:

TDB_ADD:

If an application or network protocol wants to add a new trigger, it sends a TDB_ADD message to the Trigger Database. The new trigger is stored in the Trigger Database and a corresponding TDB_NOTIFY message that indicates that a new trigger has been added is sent to all registered applications.

The format of the message is:

<HEADER, TRIGGER, INTERFACE, [ADDRESS]>

The TRIGGER payload indicates the type of the trigger and also includes some trigger specific data. The INTERFACE payload is needed to select the appropriate physical interface, the new trigger is related to. For many triggers, an additional address payload is required. It contains, for example, the new address for a TDB_TTYPE_IF_ADDRCHANGED trigger.

The response from the Trigger Database contains the same information as the request:

<HEADER, TRIGGER, INTERFACE, [ADDRESS]>

TDB_DELETE:

A trigger, which is stored inside the Trigger Database, can be deleted using the TDB_DELETE payload. In the request the only information, which has to be specified is the id of the trigger, which is stored in 'TRIGGER(*)'. A corresponding TDB_NOTIFY message that indicates that a trigger has been deleted is sent to all registered applications.

The format of the message is:

<HEADER, TRIGGER(*)>

The Trigger Database responds with a message with the following format:

<HEADER, TRIGGER>

In the response, the TRIGGER payload has all fields filled with the correct values.

TDB_GET:

The TDB_GET message is used to request all available information of a specified trigger. In the request the only information, which has to be specified is the id of the trigger, which is stored in 'TRIGGER(*)'.

The format of the message is:

<HEADER, TRIGGER(*)>

The Trigger Database responds with a message of the following format:

<HEADER, TRIGGER, INTERFACE, [ADDRESS]>

In the response a fully initialized TRIGGER payload is present. Additionally, INTERFACE payload is present as well as and an optional an ADDRESS payload, if an address is available for the specified trigger.

TDB_REGISTER:

An application, which is interested in each new trigger, can register itself to the Trigger Database. After the application has registered, it receives a message each time a new trigger has been added to the database. The format of the message is:

<HEADER>

No additional payload has to be added. The Trigger Database responds with a message of the same type and with the same content, i.e. its format is:

<HEADER>

TDB_DEREGISTER:

An application, which is no longer interested in receiving notifications about trigger changes, can de-register itself from the Trigger Database. The format of the message is:

<HEADER>

No additional payload has to be added. The Trigger Database responds with a message of the same type and with the same content, i.e. its format is:

<HEADER>

TDB_NOTIFY

An application that has registered itself to get informed about the new triggers or updates to these triggers, receives a TDB_NOTIFY message. The format of the message is the same as for a TDB_ADD message. The only difference is that some field are filled by the Trigger Database before sending the TDB_NOTIFY message.

The format of the message is:

<HEADER, TRIGGER, [INTERFACE], [ADDRESS]>

Since this message is sent by the Trigger Database itself, a registered application MUST NOT respond to it.

TDB_MODIFY:

If an application or a network protocol wants to modify a trigger (because its status has changed), it sends a TDB_MODIFY message to the Trigger Database. The new trigger is stored and a corresponding TDB_NOTIFY message that indicates that an existing trigger has been modified is sent to all registered applications.

The format of the message is:

<HEADER, TRIGGER, [INTERFACE], [ADDRESS]>

The TRIGGER payload indicates the type of the trigger and also includes some trigger specific data.

The response from the Trigger Database contains the same information as the request:

<HEADER, TRIGGER, [INTERFACE], [ADDRESS]>

TDB_DUMP:

An application, that wants to learn all currently available triggers should send a TDB_DUMP message. Since a TDB_GET message requires a specific trigger id for retrieval, applications which to not know all trigger IDs depend on this message class for learning all unknown triggers. The format of the message is:

<HEADER>

The Trigger Database will respond with all currently available triggers entries by serially sending the following message:

<HEADER, TRIGGER, INTERFACE, [ADDRESS]>

TDB_FLUSH:

For deleting all entries in a Trigger Database, the TDB_FLUSH message is used. Since the TDB_GET message requires a specific trigger id for deletion, reliable cleaning of a Trigger Database can be done with this message. The format of the message is:

<HEADER>

The Trigger Database will respond with the following message:

<HEADER>

7. Payload Format

HEADER:

Each message starts with the fixed header. It contains general information about the message and determines, which payloads have to be included in it. It has the following format:

```
struct tdb_header {
        uint8_t         tdb_header_version;
        uint8_t         tdb_header_msgtype;
        uint8_t         tdb_header_errno;
        uint8_t         tdb_header_reserved1;
        uint16_t        tdb_header_msglen;
        uint16_t        tdb_header_reserved2;
        uint32_t        tdb_header_seq;
        uint32_t        tdb_header_pid;
} __attribute__( ( packed ) );
/* sizeof( struct tdb_header ) == 16 */
```

The fields of this structure contain the following values:

tdb_header_version: The version of the used PF_TRIGGER interface. This document specifies this API in version 1.

tdb_header_msgtype: This field contains the type of the message. All possible values are listed in the table in Section 6.

tdb_header_errno: If an error occurred while processing a request, the response will only include the message header without any payloads. The type of the error is indicated by the value in this field. The values are taken from the error number specification of the operating system (e.g. the errno.h file).

tdb_header_msglen: The length of the message divided by 8 is stored into this field.

tdb_header_seq: This field contains the number of the last message sent incremented by 1.

tdb_header_pid: The process id of the program sending the message. If the message is generated inside the kernel, this value is set to zero.

INTERFACE:

The INTERFACE payload is used to provide all needed information about an active network interface.

The format of the INTERFACE payload is the following:

```
struct tdb_interface {
        uint16_t        tdb_interface_len;
        uint16_t        tdb_interface_pltype;
        uint32_t        tdb_interface_selector;
        uint32_t        tdb_interface_type;
        uint32_t        tdb_interface_bandwidth;
        uint32_t        tdb_interface_quality;
        uint32_t        tdb_interface_reserved;
} __attribute__( ( packed ) );
/* sizeof( struct tdb_interface ) == 16 */
```

This fields contain the following values:

tdb_interface_len: This field contains the length of the payload divided by 8.

tdb_interface_pltype: This field contains the value TDB_PT_INTERFACE.

tdb_interface_selector: The tdb_interface_selector field stores interface enumeration information for unique identification (IF #0, #1, #2, ...). When a new interface comes up, this value should be set by the kernel.

tdb_interface_type: Classification of an interface, for instance fixed or wireless network link. The MIH Function provides this information by issuing a "MIH_Poll.request" from the Trigger Database, before creating any event notification destined for the MOBIKE daemon.

tdb_interface_bandwidth: Information about the maximum bandwidth of an interface. The MIH Function provides this information by issuing a "MIH_Poll.request" from the Trigger Database, before creating any event notification destined for the MOBIKE daemon.

tdb_interface_quality: Information about current connection quality of an interface. The MIH Function provides this information by issuing a "MIH_Poll.request" from the Trigger Database, before creating any event notification destined for the MOBIKE daemon.

tdb_interface_reserved: This field is reserved for future use and MUST be set to zero.

Further information about an interface might be necessary. Especially asymmetric link connectivity/availability in case of wireless connections might be relevant. This is left for future investigation.

ADDRESS:

The ADDRESS payload is used to provide the IP address of an interface to the Trigger Database or registered application. This information is important for most triggers. But it might be possible that there are trigger types which do not need an ADDRESS payload.

The format of the ADDRESS payload is:

```
struct tdb_address {
        uint16_t      tdb_address_len;
        uint16_t      tdb_address_pltype;
        uint8_t       tdb_address_proto;
        uint8_t       tdb_address_prefixlen;
        uint16_t      tdb_address_reserved;
} __attribute__( ( packed ) );
/* sizeof( struct tdb_address ) == 8 */

/* followed by some form of struct sockaddr */
```

Information about IP address and probably ports is provided by a sockaddr structure which is attached to the tdb_address structure. A sockaddr structure is capable of storing both a IPv4 and IPv6

address. The fields of the tdb_address structure contains the following values:

tdb_address_len: This field contains the length of the payload including the sockaddr structure divided by 8.

tdb_address_pltype: The tdb_address_pltype field contains the value TDB_PT_ADDRESS.

tdb_address_proto: The tdb_address_proto field is normally set to zero. However, if is are set in the attached sockaddr needed, then the field SHOULD be set to the protocol number of the upper layer protocol. (e.g. TCP or UDP). This functionality may become relevant for signalling IPSec related information (e.g. tunnel changes)

tdb_address_prefixlen: This field contains the prefix length of the address, which might be useful to key management applications, which employ it in access control decisions. If the tdb_address_prefixlen is non-zero the address has a prefix.

tdb_address_reserved: The tdb_address_reserved field is reserved for future use and MUST be set to zero.

TRIGGER:

The TRIGGER payload is used to provide all needed information about a trigger itself, e.g. the trigger type, an id, etc. The notation TRIGGER(*) indicates that only the id field is used to identify the trigger and all other fields SHOULD be set to zero.

The format of the TRIGGER payload is the following:

```
struct tdb_trigger {
        uint16_t        tdb_trigger_len;
        uint16_t        tdb_trigger_pltype;
        uint16_t        tdb_trigger_type;
        uint16_t        tdb_trigger_reserved1;
        uint32_t        tdb_trigger_id;
        uint32_t        tdb_trigger_reserved2;
} __attribute__( ( packed ) );
/* sizeof( struct tdb_trigger ) == 16 */
```

This fields contain the following values:

tdb_address_len: This field contains the length of the payload divided by 8.

tdb_address_pltype: This field contains the value TDB_PT_TRIGGER.

tdb_address_type: The type of the trigger is stored into this field. All possible values are listed in the table in section Section 4.

tdb_address_id: The id of a trigger is assigned by the Trigger Database itself. In the message sent by userspace programs, which do not know this value (e.g. for TDB_ADD messages), this value MUST be set to zero.

Further information about a trigger might be necessary. This is left for future investigation.

8. Applicability

Even though this document is intended to give a solution for MOBIKE, the API is generic enough to make information available for other protocols as well.

The Next Step In Signaling (NSIS) protocol suite, for example, requires access to up-to-date information about IP addresses, interfaces and interactions with mobility protocols. In order to react on mobility events some sort of interaction between the kernel, various signalling protocols (including Mobile IP, IKE/IPsec, etc.) and the NSIS daemon is required (see [9]). Hence, an NSIS daemon supporting mobility could benefit from a generic interface to meet it's requirements for fast and accurate detection of mobility events, address and interface changes. GIMPS, for example, demands immediate reaction in case of a mobility event (e.g., handover). Monitoring procedures of mobility management protocols like Mobile IP are required to react to these mobility events in an appropriate way.

The trigger database and it's API could provide necessary information for detecting such a movement (new interface/IP address available, expiring Mobile IP timers).

9. IANA Considerations

This document defines an IANA registry for the protocol family PF_TRIGGER.

An IANA registry might be needed for the different trigger types (for which examples are provided in Section 4).

10. Security Considerations

This document describes an API which allows information about IP addresses to be obtained at a local host. A malicious application or protocol daemon could disseminate wrong information. This would make other protocols, such as MOBIKE, believe that the status of a particular address has changed. This will likely lead to unexpected protocol behaviour, such as switching between addresses back-and-forth. Hence, a certain trust has to be placed into the applications and protocol daemons that are allowed to access the database to insert, modify or delete triggers. Access control mechanisms might enforce certain rights to use the API or parts of it.

11. Acknowledgments

The authors would like to thank Murugaraj Shanmugam, Yu Xinwen, Wolfgang Groeting and Stefan Berg for their comments. Furthermore, the authors would like to thank Emanuel Corthay for pointing them to the IEEE 802.21 draft.

12. References

12.1. Normative References

[1] Eronen, P., "IKEv2 Mobility and Multihoming Protocol (MOBIKE)", draft-ietf-mobike-protocol-08 (work in progress), February 2006.

[2] McDonald, D., Metz, C., and B. Phan, "PF_KEY Key Management API, Version 2", RFC 2367, July 1998.

[3] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", March 1997.

12.2. Informative References

[4] Kivinen, T. and H. Tschofenig, "Design of the MOBIKE Protocol", draft-ietf-mobike-design-08 (work in progress), March 2006.

[5] Rajkumar, Ajay., Williams, Michael., Liu, Xiaoyu., and Vivek. Gupta, "Media Independent Handover Services", IEEE-Draft Draft IEEE Standard for Local and Metropolitan Area Networks / IEEE P802.21/D00.01, July 2005.

[6] Sklower, K., "Tree-based Packet Routing Table for Berkeley UNIXProceedings of the Winter 1991 USENIX Conference, Dallas, TX, USENIX Association. 1991. pp. 93-103.", 1991.

[7] Salim, J., Khosravi, H., Kleen, A., and A. Kuznetsov, "Linux Netlink as an IP Services Protocol", RFC 3549, July 2003.

[8] Sugimoto, S. and F. Dupont, "PF_KEY Extension as an Interface between Mobile IPv6 and IPsec/IKE", draft-sugimoto-mip6-pfkey-migrate-01 (work in progress), August 2005.

[9] Lee, S., Jeong, S., Tschofenig, H., Fu, X., and J. Manner, "Applicability Statement of NSIS Protocols in Mobile Environments", draft-ietf-nsis-applicability-mobility-signalling-02 (work in progress), July 2005.

Authors' Addresses

Udo Schilcher
Universitaet Klagenfurt
Klagenfurt, Carinthia 9020
Austria

Email: udo.schilcher@edu.uni-klu.ac.at

Hannes Tschofenig
Siemens
Otto-Hahn-Ring 6
Munich, Bayern 81739
Germany

Email: Hannes.Tschofenig@siemens.com

Franz Muenz
Siemens AG
Otto-Hahn-Ring 6
Munich, Bayern 81739
Germany

Email: Franz.Muenz@thirdwave.de

Shinta Sugimoto
Ericsson Research Japan
Nippon Ericsson K.K.
Koraku Mori Building
1-4-14, Koraku, Bunkyo-ku
Tokyo 112-0004
Japan

Phone: +81 3 3830 2241
Email: shinta.sugimoto@ericsson.com

Intellectual Property Statement

Disclaimer of Validity

Copyright Statement

Acknowledgment

Funding for the RFC Editor function is currently provided by the Internet Society.