IKEv2 Mobility and Multihoming (mobike)
Internet-Draft
Expires: August 18, 2005

U. Schilcher
H. Tschofenig
Siemens

February 14, 2005

Application Programming Interface to a Trigger Database for MOBIKE
draft-schilcher-mobike-trigger-api-00.txt

Status of this Memo

Copyright Notice

Abstract

One of the functionality of MOBIKE is to create and to maintain a set of available addresses and to provide them to the communication partner. A MOBIKE peer should have some information about the status of each address in order to execute the respective actions (e.g., switching from one preferred address to another). This information, which will be referred as trigger, is distributed over a number of

protocols daemons at an end host. To make this information available to the 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 implementation is started first it has to build a set of all available addresses (or a subset of them for policy reasons; see [3]) before communicating with another peer. From these addresses, it has to select one of the addresses as 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 could be disconnected 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.

For MOBIKE to work, it is really important to know about the status of the available addresses in order to make reasonable decisions. A number of other protocols running on the end host might have various information necessary to derive a decision whether to switch from one preferred address to another or whether it is necessary to modify the peer address set.

In this document, we therefore suggest to define an API that allows protocol daemons to insert information (triggers) into a "database" that can later be made available to the MOBIKE daemon. The API is based on the BSD routing socket API in a similar fashion as PF_KEY [1] extends the same API for generic key management usage. This document therefore heavily focuses on the functionality offered by the PF_KEY specification.

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 [2].

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.

3. Trigger Classification

Many different events may cause a change in the address set used by MOBIKE (see [3]). These events can be notified 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_ADDRCHANGED	3	Interface has changed its address (e.g. new DHCP lease)
TDB_TTYPE_CONN_ESTABLISHED	4	e.g. dial-in network has connected
TDB_TTYPE_CONN_LOST	5	connection to network lost
TDB_TTYPE_DEST_UNREACHABLE	6	e.g. ICMP packet received

A future version of this document will add more triggers and a more

detailed description of them.

4. 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 [1]).

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 commands. Any other commands like bind, connect, etc. are not supported and MUST NOT have any effects on a socket of the PF_TRIGGER family.

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 5.

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 every new trigger, registers itself to the Trigger Database by sending a TDB_REGISTER message. If a protocol daemon wants to add a new trigger, it sends a TDB_ADD message to the Trigger Database including information that is important for this new trigger.

The Trigger Database acknowledges this message with a TDB_ADD response to the network protocol and with a TDB_NOTIFY message to the registered MOBIKE implementation. This notify message contains some information about the new trigger including its id. All information available about the new trigger can be requested with a TDB_GET message.

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

5. 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 6) 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	Register an application to receive a messages for each new trigger added.
TDB_NOTIFY	5	A new trigger has been added, deleted or updated.
TDB_MODIFY	6	Modify a trigger in the Trigger Database

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_ADDRESS	1	The IP address of an IF
TDB_PT_TRIGGER	2	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, [ADDRESS]>

The TRIGGER payload indicates the type of the trigger and also includes some trigger specific data. 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, [ADDRESS]>

TDB_DELETE:

A trigger, which is stored inside the Trigger Database, can be deleted using the TDB_DELETE payload.

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.

The format of the message is: <HEADER, TRIGGER(*)>

The Trigger Database responds with a message of the following format: <HEADER, TRIGGER, [ADDRESS]>

In the response a fully initialized TRIGGER payload is present. Additionally, an ADDRESS payload is present, 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_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, [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 new 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, [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, [ADDRESS]>

6. 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 5.

tdb_header_errno: If an error occured 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.

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 trigger types that 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_prefixlen;
        uint8_t         tdb_address_reserved1;
        uint16_t        tdb_address_reserved2;
} __attribute__( ( packed ) );
/* sizeof( struct tdb_address ) == 8 */
```

Appended to the tdb_address structure is always a sockaddr structure that includes the actual IP address. It is possible to add an IPv4 or an 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_prefixlen: This field contains the prefix length of

the address.

TBD: Clarification about the prefix len needs to be provided in a future document version.

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 3.

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.

7. 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 3).

8. 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 behavior, 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.

9. Acknowledgments

The authors would like to thank Murugaraj Shanmugam for his comments.

10. References

10.1 Normative References

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

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

10.2 Informative References

[3] Kivinen, T. and H. Tschofenig, "Design of the MOBIKE protocol", Internet-Draft draft-ietf-mobike-design-01, January 2005.

Authors' Addresses

Udo Schilcher
Siemens
Otto-Hahn-Ring 6
Munich, Bayern 81739
Germany

Email: USchilcher@siemens.com

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

Email: Hannes.Tschofenig@siemens.com

Intellectual Property Statement

Disclaimer of Validity

Copyright Statement

Acknowledgment

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