Movement Notification in Mobile IPv6

Mark Borst, Bilhanan Silverajan

Abstract— **Mobile IPv6 is intended to hide the mobility of a node from the application. Some applications, however, are location aware and like to know when a node is moved. In this paper, we discuss the design and implementation of a Movement Notification library for Mobile IPv6. It supports applications running on a mobile node to ask for a notification when the node has moved. This can be done in either a blocking or a non-blocking way. This implementation is based on the MIPL Mobile IPv6 stack for Linux. It is implemented by a hook into the kernel, a userspace daemon and a userspace library. Testing shows that it works reliably.**

Keywords— **Mobile IPv6, movement notification**

I. Introduction

These days, nodes on the Internet are becoming more and more mobile. This gives rise to the need to support mobility at the IP level. A lot of research is done into the needs of moving nodes. One of the fundamental papers here is by Perkins and Johnson [1], which has finally resulted in an IETF draft standard called "Mobility Support in IPv6" [2], soon to become an RFC.

One of the achievements of Mobile IP is that the mobile node is always addressable by its home address, regardless of where the node actually is. The topologically correct addressing part is done by having a second address, the care-of address. This care-of address changes each time the node moves, but the home address remains the same. Effectively this shields the movement of the node to higher layers, which do not know anything about the movement at all.

While this is perfectly fine for most applications and protocols, for instance where it enables TCP to keep a connection open, it does not completely suit the needs of location-sensitive services. These services might want to be notified by the IP layer whenever a node moves to a new location.

One of the possible uses of application level knowledge of the fact that a node has moved, is in extending the Service Location Protocol (SLP) for mobile environments, of which an IPv6 implementation is well on its way [3]. Upon movement, SLP can be used to rediscover services in new environments that the node moves to. This requires the IP layer to have a mechanism that notifies the application when the mobile node has moved.

Another use could be an application that behaves differently when it is not in its home network. The network usage fee could for instance be higher in foreign networks, so an application could decide to defer downloading big files when it is in a foreign network, and wait until the node is in a lower-cost network again.

Dept. of Information Technology, Tampere University of Technology, P.O. Box 553, FIN-33101 Tampere, Finland MARK@MWBORST.COM, BILHANAN@CS.TUT.FI

This paper describes the design and implementation of a movement notification mechanism at the IP level to inform applications when the node moves. It is specifically aimed at supporting applications running on a mobile node. The name *mobapi* is used to refer to this implementation.

Section II describes the background of the work, and internet drafts available for this work. Section III describes the design, Section IV discusses the implementation, and Section V evaluates the work.

II. Background

Two Internet drafts exist that discuss passing information from the IP layer to applications. The first is "Extensions to Sockets API for Mobile IPv6" by Chakrabarti and Nordmark [4] and the second is "Mobile IP API" by Yegin et al. [5].

[4] extends the "Advanced Sockets API for IPv6" draft by Stevens et al. [6] to include Mobile IPv6. It provides a POSIX compliant way to read and modify the added options in an IP packet, which are the mobility protocol header, home address destination option and routing header. This is done by using raw sockets. It is mainly intended to facilitate debugging and diagnostic applications, but it could also be used by an application watching the mobility headers to distill movement from the change in options. This requires an analysis of all passing packets.

[5] specifies amongst others an API for movement notification. This draft is aimed at providing an API for enduser applications, to ask for movement notification. It does not specify how the movement has to be detected. As we are looking for an application level API to give notifications of movement, [5] is used as the basis for this work.

[5] also discusses why a Mobile IP API is important, and gives some examples on where it can be used. It has the following requirements for the API:

- provide location awareness
- provide movement awareness
- does not disturb other applications
- specific for Mobile IP, should not change protocols

The idea is that there is a Mobility Management Module, that keeps track of all mobile nodes that are known to the local computer. This is mainly from the perspective of a correspondent node, which could communicate with multiple mobile nodes.

The specification is written from the point of view of a correspondent node, but it also partially covers the needs of a mobile node. The most important aspect of the draft with regard to a mobile node is that it provides an API for an application to ask for a no-

tification when the node moves. This request goes by calling the function *mip_notify_movement*.

Asking for a notification can be done in either blocking mode, or non-blocking mode. In blocking mode, *mip_notify_movement* does not return until the node has moved or a specified timeout period has passed, so the calling application just waits until that time. In non-blocking mode, the application registers a callback function with *mip_notify_movement* which returns immediately. Subsequently, when the node has moved, the callback function is invoked. We anticipate that the non-blocking mode would be the way most protocols and event-based applications will want to use it.

Currently there are several implementations of Mobile IPv6 actively being developed [7]. The one that is used in this work, is MIPL (Mobile IPv6 for Linux) [8], developed as a patch against the Linux kernel by Helsinki University of Technology, Finland. The version used in this work is MIPL version 1.0, and the vanilla Linux kernel version 2.4.22.

When the term 'MIPL' is used in the rest of this work, it refers to the Mobile IPv6 stack implementation in the Linux kernel.

III. Design

As discussed earlier, the work of [5] is used as a basis for *mobapi*. A mobile node is identified in [5] with the following parameters:

- the home address of the mobile node
- the current care-of address of the mobile node
- the address of the home agent of the mobile node

These parameters are encoded in the following structure:

```
typedef struct mobile_node_t_ {
  struct in6_addr home_addr;
  struct in6_addr co_addr;
  struct in6_addr ha_addr;
} mobile_node_t;
```

The draft also specifies three functions:

- *mip_get_all_mobile_nodes*
- *mip_get_one_mobile_node*
- *mip_notify_movement*

The first two functions are meant to extract information from the mobility management module. The function *mip_get_all_mobile_nodes* returns a list of all nodes that are known. The function *mip_get_one_mobile_node* returns information about one specific mobile node.

The last function is meant to notify an application of movement of the node. As the intent of this project is to provide movement notification to applications running inside the mobile node, only the functionality of *mip_notify_movement* is considered.

An application can ask to be notified of the movement of a specific mobile node. This node is specified by its home address, while the any-address (0::0) specifies 'all nodes'. The need to specificy the node comes from the correspondent node operation, which can support multiple mobile nodes. In this implementation, the focus is on local operations. To conform to the specification, the home address has to be specified, but the most useful operation is with the any-address. A side effect is that the function *mip_notify_movement* always has to return the current addresses of the mobile node. If one specifies the any-address, and requires the function to return immediately (for instance by using a blocking call with a timeout of 0 ms), one can retrieve the home address of the node, and its care-of address.

The rest of this section explains specific design issues to implement *mobapi*.

A. Overall structure

The movement notification work (*mobapi*) is split into three parts: a kernel module addition, a userspace daemon, and a shared library. The ultimate purpose of *mobapi*, is to provide a library to all applications, so that the applications can call a function in the library to get notified of movements by the node. This library is called *libmobapi*.

Since the kernel is the only place where the movement information is known, this information has to be extracted and passed to the library. This kernel part is called *mobapi_kernel*. This kernel part is a hook into the MIPL Mobile IPv6 stack.

It should be possible that there are multiple applications wanting to get the movement information, and the kernel should not be burdened with managing communication channels to several applications. To relieve the kernel, a daemon is introduced that provides the glue between the kernel and the applications. This daemon is called *mobapid*.

This subdivision is shown in Figure 1.

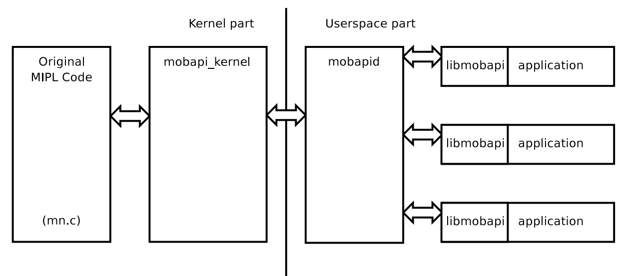

Fig. 1. The overall structure of *mobapi*

It is clear that there can be multiple applications running, but there is still only one access point to the kernel, which is managed by the daemon.

B. Communication between parts

The communication between the different parts (*mobapi_kernel*, *mobapid* and *libmobapi*) goes by passing messages via two different mechanisms. The kernel part implements a character device driver, that results in a file (*/dev/mobapi*) in the dev-fs file system. The daemon uses that file for communication with *mobapi_kernel* by means of read and write.

The daemon and the library communicate by means of FIFO's. A FIFO is a named pipe, via which unrelated processes can communicate. The daemon creates a well-known FIFO. The library can then open that FIFO, and send messages to the daemon. This way, all instances of the library can communicate with the daemon. To receive messages back, the library has to create its own private FIFO, via which the daemon can send messages to the library.

There are 4 messages defined in *mobapi*. They are:

- SUBSCRIBE
- UNSUBSCRIBE
- NODE_INFO
- NODE_MOVED

The first two messages are only sent from the library to the daemon. The last two are used as both kernel-to-daemon and daemon-to-library messages.

All messages consist of at least the following parameters containing the current status of the node:

- the home address of the mobile node
- the current care-of address of the mobile node
- the address of the home agent of the mobile node

These are also the values that are defined in the mobile_node_t datastructure in [5].

The first attempt of *mobapi* is to propagate the current status (contained in a NODE_INFO message) to the library as quick as possible. That is needed in order to let the library return the current status to the application also when no movement has occured. This means that when the daemon connects to the kernel, the kernel sends a NODE_INFO message to it. Also when an application connects to the daemon via the library, the daemon sends a NODE_INFO message to the library.

Establishing an association between the library and the daemon is done by sending a SUBSCRIBE message from the library to the daemon. As said before, the daemon responds to that by sending a NODE_INFO message. The association can be destroyed by sending a UNSUBSCRIBE message to the daemon. The daemon does not send a response to that message.

When the node has moved, *mobapi_kernel* sends a NODE_MOVED message to *mobapid*. The daemon relays that message to all applications it has an association with.

A possible flow of messages is shown in Figure 2. It shows a daemon handling two applications, one that wants to receive a couple of movement notifications, and another one that is only interested in one notification. This could be originating from a non-blocking and a blocking call, respectively. Clearly it can be seen here, that a SUBSCRIBE message triggers a NODE_INFO message.

C. Kernel

The kernel part, *mobapi_kernel*, consists of two parts: the interface to MIPL and the interface to the daemon, via the implementation of the character device */dev/mobapi*. The interface to MIPL provides the hooks into the kernel, and consists of three functions that can be called by MIPL, to provide startup, movement and exiting, and three functions of MIPL that are called by *mobapi_kernel*, to get hold of the addresses.

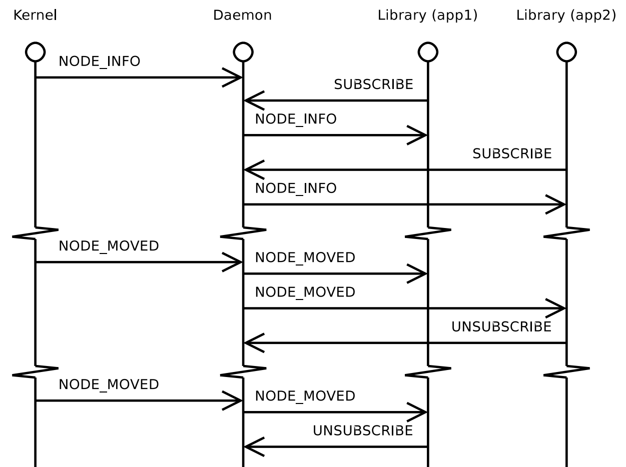

Fig. 2. A possible flow of messages

The implementation of the character device consists of five functions that implement standard operations on files.

D. Daemon

The daemon keeps track of the applications that want to receive notifications. It dispatches the notifications received from the kernel to all the applications that are interested.

The design of the daemon is quite straight-forward. It forks itself into the background, and then keeps itself busy with *poll*ing the two communication interfaces. When a message is received, it is processed, and then the daemon goes *poll*ing again.

E. Library

The library implements the functionality as specified in [5] that is needed for *mobapi*: the function *mip_notify_movement* and the macro *IS_AT_HOME*.

In blocking mode, the function returns when either the node has moved, or a timeout (specified in milliseconds) occurs.

In non-blocking mode, the application has to provide a callback function, that can be called when the node has moved. When that function is registered, *mip_notify_movement* returns. When the node moves (the library receives a NODE_MOVED message), the library calls the callback function. It passes the node's status (a filled 'mobile_node_t' structure), an event indicator (in case of movement, its value is MIP_MN_MOVED), and a parameter that is provided by the application. The callback function can be deregistered by calling *mip_notify_movement* again, with a NULL callback function. Then the callback function is called one last time with MIP_CB_DEREGISTER as event, and deregistered.

In order to let the application run and still get notifications from the daemon, POSIX threads are used. When a callback function is registered, a pthread is started, that starts listening on the FIFO. This way, when a message comes on the FIFO, the library notices it immediately, and can call the callback function then.

The overall algorithm of *mip_notify_movement* is this:

```
if(blocking)  . . .
else if(non-blocking)
  if(registration) . . .
  else if(deregistration) . . .
```

When the call to *mip_notify_movement* is a blocking call or a non-blocking callback registration, a SUBSCRIBE message is sent to the daemon. Upon reception of that message, the daemon sends a NODE_INFO message back. After that message is received, *libmobapi* starts waiting for the movement notification (a NODE_MOVED message). When that notification comes, *libmobapi* either returns (blocking mode), or does the callback (non-blocking mode). In both cases, the care-of and home addresses are returned.

To be able to give the address information back before a movement has occurred (that can be in blocking mode after a 0 ms timeout, or in non-blocking mode when a callback is deregistered before a movement occurs), the library must know the addresses as quick as possible. That is the reason why a SUBSCRIBE message is answered immediately with a NODE_INFO message.

IV. Implementation

A. Messages

The messages used for communication between the daemon and the library are:

```
struct mobapi_command_t {
  int type;
  struct in6_addr home_addr;
  struct in6_addr co_addr;
  struct in6_addr ha_addr;
  int fifo;
};
```

and for communication between the kernel and the daemon:

```
struct mobapi_message_t {
  int type;
  struct in6_addr home_addr;
  struct in6_addr co_addr;
  struct in6_addr ha_addr;
};
```

The *type* field can be either SUBSCRIBE, UNSUBSCRIBE, NODE_INFO or NODE_MOVED. These names were introduced before.

The *mobapi_command* message has an extra parameter *fifo*. This is set to the pid of the library, and indicates to what FIFO the daemon should send its response. This is discussed in section IV-C.

B. Kernel

The kernel part consists of the interface to MIPL, and the the character device */dev/mobapi* as interface to the daemon.

B.1 Interaction with MIPL

The interface to MIPL consists of three functions that can be called by MIPL: *mobapi_init*, *mobapi_exit* and *mobapi_mobile_node_moved*, and three functions of MIPL that are called by *mobapi_kernel*: *mipv6_get_care_of_address*, *mipv6_mn_is_home_addr* and *MIPV6_CALLPROC(mipv6_get_home_address)*.

When MIPL starts (when module mip6_mn is loaded by the linux kernel), the function *mobapi_init* is called. This function initializes the wait_queue needed for *mobapi_poll*, and registers the character device. This involves passing references to the functions needed for the interface to the character device. It also initializes the queue with messages to the daemon. Conversely, when module mip6_mn is unloaded, the function *mobapi_exit* is called. This function clears the message queue, and unregisters the character device.

When MIPL detects that the node has moved, it calls its own internal function *mipv6_mobile_node_moved*. This function takes care of sending all necessary binding updates to the home agent and correspondent nodes, and it also calls the function *mobapi_mobile_node_moved*. This function first gets the new care-of address, and then queues a NODE_MOVED message. Finally, it wakes up the wait_queue for *poll*.

The retrieval of addresses from MIPL is concentrated in one function, *mobapi_get_addresses*. This is to increase portability. Getting the care-of address is done by calling the function *mipv6_get_care_of_address*. To get the home address, the function *MIPV6_CALLPROC(mipv6_get_home_address)* is called, and to check the validity of the result, *mipv6_mn_is_home_addr* is called afterwards.

B.2 Interaction with daemon

Messages that have to be sent from the kernel to the daemon, are queued in a linked list, called *message_queue*, which has a head and a tail pointer. This gives the possibility to add messages to the end of the queue, and remove messages from the start. It actually implements a fifo queue. As there should be only one daemon running, this message queue is global, and need not be process specific.

Passing messages to the daemon is implemented by queuing them in this queue, and sending one each time *mobapi_read* is called while there are messages available. The function *mobapi_poll* returns 'readable' only when there is a message waiting in the queue. In this way the daemon can just call *poll*, and *read* a message when *poll* returns. The functions *mobapi_open* and *mobapi_release* increase and decrease the module use-count, respectively, and in *mobapi_open* also

a NODE_INFO message is queued, to tell the daemon immediately what the current status is.

One can create a character device from the kernel, essentially by providing the file subsystem with operations that implement all system calls. The relevant system calls for *mobapi_kernel* are *read(2)*, *write(2)*, *poll(2)*, *open(2)* and *close(2)*. All these calls are used by the daemon to access the device. The other possible calls, like *stat(2)*, *lseek(2)*, *fsync(2)* etc, are not used.

To implement all these calls, the respective functions

- *mobapi_read*
- *mobapi_write*
- *mobapi_poll*
- *mobapi_open*
- *mobapi_release*

are used. They are put into a *struct file_operations*, that is passed as parameter to *register_chrdev*, which registers a character device driver with the operating system and exports it interface.

The function *mobapi_poll* puts the process to sleep by calling *poll_wait*, using the wait-queue *notification_queue*. Then it builds a bitmask indicating whether the device is readable and/or writable. It is readable when *message_queue* is not empty.

In the *notification_queue* are all processes that wait on a certain event related to that queue. In *mobapi_kernel*, two functions make data available for read by queueing a message in *message_queue*: *mobapi_mobile_node_moved*, which queues a NODE_MOVED message, and *send_node_info*, which queues a NODE_INFO message. When they queue a message, they call *wake_up_interruptible* on the notification queue. This causes the process that was put to sleep by *poll_wait*, to wake up.

C. Daemon

The communication from library to daemon goes via a well-known FIFO. This FIFO is called */tmp/mobapi.in*. To communicate from the daemon back to the connected libraries, another technique is used. The library has to create its own FIFO, which is unique because it includes the process ID (pid) of the application in its name. The name then becomes */tmp/mobapi.$(PID)*. This pid is passed in a SUBSCRIBE message from the library to the daemon. The daemon then constructs the filename, opens that file for write, and can then send messages to the library.

When the daemon starts, the first thing it does is forking, and then let the parent exit. That means that the program moves to the background, detached from the terminal. It also has to catch some signals. This is to facilitate exiting the daemon. The way to exit the daemon is by sending it a SIGHUP, SIGKILL or SIGTERM signal. These signals are caught, and redirected to a signal handler (*die*), that closes the open files and removes the FIFO. A *write* to a FIFO that has no readers (i.e. there is no process that has it opened for reading), causes a SIGPIPE. If that signal is not caught, the daemon would abort. Therefore SIGPIPE is caught, but ignored.

After this setup process, the character device */dev/mobapi* is opened, and the well-known FIFO */tmp/mobapi.in* is created and opened. This FIFO is opened for read and write, to prevent POLLHUP's returning from *poll* when all clients close the file (then there are no writers). This is because the daemon also has to keep running when there are no clients.

When all is set up, the daemon enters an infinite loop, doing a *poll* on both communication gateways,*/dev/mobapi* and */tmp/mobapi.in*. When a message arrives on one of them, it is handled by *process_dev_entry* and *process_in_fifo*, respectively.

The function *process_dev_entry* reads a message from the character device, and depending on the type field, it either only stores the info (when it is NODE_INFO), or it relays a NODE_MOVED message to the clients that are interested in that specific node, when it is a NODE_MOVED message.

The function *process_in_fifo* first reads a message from the input FIFO. When that message is a SUBSCRIBE message, it checks whether the requested-for home address is valid (either the own home address, or the 0::0 address). If it is valid, then the client is added to the beginning of the list of clients. This list is a double linked list, defined as:

```
struct client_list_t {
  int fd;
  int fifo_nr;
  struct in6_addr home_addr;
  struct client_list_t *next;
  struct client_list_t *prev;
};
```

It includes 'fifo_nr', which is actually the pid of the library, and the file descriptor ('fd') associated with that open FIFO. Also the home address for which notifications are requested, is stored. After the client is added to the list, a NODE_INFO message is sent to the client.

When the received message is an UNSUBSCRIBE message, it is checked whether the unsubscription is for a valid home address. When it is, the FIFO is closed, and the client is removed from the client list.

D. Library

The library *libmobapi* implements the macro *IS_AT_HOME*, and the function *mip_notify_movement*. The macro *IS_AT_HOME* is not more than a comparison of the home address and the care-of address of the node. This is done by calling the system macro *IN6_ARE_ADDR_EQUAL* like this:

```
#define IS_AT_HOME(t) \
  IN6_ARE_ADDR_EQUAL( \
  &((t)->home_addr), &((t)->co_addr))
```

The main function that is implemented in *libmobapi* is this:

```
int mip_notify_movement(
```

```
  mobile_node_t *mobile_node,
  int non_blocking,
  unsigned int timeout_ms,
  long int cb_parameter,
  int (*callback)
    ( mobile_node_t mobile_node,
      int event,
      long int cb_parameter))
```

The parameter *cb_parameter* is only used to be passed to the callback function. The parameter *non_blocking* indicates whether the call is blocking or non-blocking. The parameter *timeout_ms* indicates the timeout for the blocking call. Setting it to -1 gives an infinite timeout. This timeout is implemented by using the timeout mechanism of the *poll* call. The parameter *event* of the callback function indicates what is the reason for the call to the callback. Its value can be MIP_MN_MOVED or MIP_CB_DEREGISTER. [5] also specifies MIP_BCE_DELETE, but that value is only useful in correspondent node operation, which is not implemented, so that value is not used.

The parameter *mobile_node* is used as both input and output parameter. The field *home_addr* of this structure is used to indicate the node of which the application is interested in the movement.

The other fields (*co_addr* and *ha_addr*) are used as output parameters, and are set to the current values when a blocking call returns. For a non-blocking call, the output values are unspecified.

The infrastructure is in place to move all the addresses mentioned in the structure *mobile_node_t* around. However, knowledge of the home agent address (*ha_addr*) is not implemented, so for that address the any-address is used. It also means that it should be quite easy to extend the functionality to include the correct *ha_addr*, by adapting *mobapi_get_addresses* in *mobapi_kernel*.

The return value of *mip_notify_movement* for blocking mode is zero when a timeout occurred, and a positive value (1) otherwise. For non-blocking mode, the return value is not specified, so zero is used.

The library maintains a double linked list of callbacks:

```
struct callback_t {
  struct in6_addr home_addr;
  int (*function)(mobile_node_t mn,
         int event, long int par);
  long int parameter;
  struct callback_t *next;
  struct callback_t *prev;
};
typedef struct callback_t callback_list;
```

This is because an application could register multiple callbacks each for another home address. This is valid in [5], but quite useless in this implementation, because only movement of the local node is reported. However, doing it is allowed, so one movement could cause multiple callbacks in one application.

V. Conclusions

We present a working implementation of the functions mentioned in this paper. To get a notification, an application can use a blocking call or a non-blocking call. Testing has shown that it works reliably.

The design is clean. The functionality included in the kernel part is small and straightforward. The interface to the other parts of the kernel is clearly outlined. This means that it should be fairly easy to port the work to other Mobile IPv6 stacks, like KAME [9] for *BSD, or Linux kernel 2.6 with a new MIPL version. The daemon and library should be trivial to port. The only major issue that could arise is the communication between the kernel and the daemon.

Although [5] is expired and no update has been made, the best way to build *mobapi* would be to combine the ideas of the two drafts [5] and [4], that is to build the movement notification API upon the sockets API to increase portability. By removing the kernel part, and building upon the Sockets API [4] to do the movement detection in userspace (by the daemon), *mobapi* could be trivially ported to any platform supporting the Sockets API.

However, no implementation exists of [4] for MIPL. Thus we chose to implement another way to get hold of movement information, by directly hooking into the kernel, that knows that information anyway. However, at the time of this writing, the KAME platform has announced support for [4], and this may form the future work for *mobapi*.

Proper mutual exclusion is required around the message queue, because the kernel can be multitasked, and *mobapi_mobile_node_moved*, *mobapi_read* and *mobapi_open* all modify the message queue. In addition to that queue, global variables are used to store the IP addresses. Access to these variables must therefore also be protected from simultaneous access. For mutual exclusion, the kernel provides spinlocks and semaphores [10]. In *mobapi* spinlocks are used, because there is only one resource and it is very simple to use. A semaphore would be too complex for the needs here.

One issue which came up when testing, is that the router with which we tested, advertised the same link local addresses on different interfaces. While this is valid behavior, it disturbed the movement detection, because that defines movement as: reception of a router advertisement that contains a different link local address and a different network prefix than the current router. The problem thus arises that the movement detection algorithm does not detect the movement. This problem is addressed in the Mobile IPv6 draft [2] (paragraph 11.5.1 Movement Detection discusses the problem, and paragraph 7.2 Modified Prefix Information Option Format specifies the solution), but the solution is not yet implemented in the router software. The temporary solution here is to let the router have different link local addresses on different interfaces.

As a side result, an easy and reliable way is also

provided to obtain an interface's IP address. This can be done by doing a blocking call with a timeout of 0. This immediately returns with the latest home address and care-of address.

References

[1] C. E. Perkins and D. B. Johnson, "Mobility support in IPv6," in *Proceedings of the Second Annual International Conference on Mobile Computing and Networking (MobiCom'96)*, 1996.

[2] D. Johnson, C. Perkins, and J. Arkko, "Mobility support in IPv6, draft-ietf-mobileip-ipv6-24.txt," June 30, 2003.

[3] B. Silverajan, J. Kalliosalo, and J. Harju, "A service discovery model for wireless and mobile terminals in IPv6," in *Proceedings of IFIP-TC6 8th International Conference on Personal Wireless Communications PWC 2003, Venice, Italy*, pp. 385 – 396, Springer-Verlag, September 23 - 25, 2003.

[4] S. Chakrabarti and E. Nordmark, "Extension to Sockets API for Mobile IPv6, draft-ietf-mip6-mipext-advapi-00.txt," February 2004.

[5] A. A. Yegin, M. M. Tariq, A. Yokote, G. Fu, C. Williams, and A. Takeshita, "Mobile IP API, draft-yokote-mobileip-api-02.txt," June 2003.

[6] W. Stevens, M. Thomas, E. Nordmark, and T. Jinmei, "Advanced Sockets API for IPv6, RFC 3542," May 2003.

[7] P. van den Bergen *et al.*, "Mobile-IP implementations." http://www.mip4.org/2004/implementations/.

[8] MIPL, "MIPL mobile IPv6 stack." http://www.mobile-ipv6.org.

[9] KAME, "KAME mobile IPv6 stack." http://www.kame.com.

[10] R. Russell, "Unreliable guide to locking." http://www.kernel.org/pub/linux/kernel/people/rusty/kernel-locking/index.html, 2003.