



TAMPERE UNIVERSITY OF TECHNOLOGY



**University of Twente**  
*The Netherlands*

Mark Borst

## **A Movement Notification Library for Mobile IPv6**

Master of Science Thesis

Supervisors: DI Bilhanan Silverajan, TUT  
Dr.ir. Aiko Pras, UT  
Prof. Jarmo Harju, TUT  
Ir. Remco van de Meent, UT

# Foreword

This thesis work is done to obtain the degree of Master of Science for the study Telematics at the Department of Computer Science of the University of Twente, The Netherlands. The work itself is done at the Networks and Protocols Group, Institute of Communications Engineering, Tampere University of Technology, Finland.

The result of this work is also described in short on my webpage: <http://mwborst.com/mobapi>, where the source code of the programs can be downloaded. There one will also find the paper that is to be published in Proceedings of the Eunice Summer School 2004, “Advances in fixed and mobile networks” as result of this work.

I would like to thank my company in the lab, Bil, Joonas and the others, for their support, friendship and good laughs. I’d also like to thank the international student community at the university, that provided a good base of friends. Eriikka (.fi), Katrina (.us) and Takeshi (.jp) fit very well in that row. And I especially want to thank my one and only Dutch friend here at the university, who came for the same reason as me: Jan. Too bad he left this beautiful country earlier than me, so that I had to miss the countless cups of coffee we had together.

I’ve learnt a lot here, also on the practical side of networks, which was a worthy addition to the curriculum offered by the UT. Looking back, I can say that I have had a great time and gained a valuable experience.

Mark Borst

# Abstract

This thesis describes the design and implementation of a Movement Notification library for Mobile IPv6. This library is needed because Mobile IP hides the movement of a mobile node from applications. However, some applications are interested in the movement of a node, so they need an interface to obtain that information. To that end, the “Mobile IP API” draft has been specified in the IETF, and the part of this draft that is relevant for movement notification is implemented in this thesis.

The library provides a way for an application running on a mobile node to ask for notifications when that node moves. This can be done in blocking and non-blocking modes. The implementation is based on the MIPL Mobile IPv6 stack for Linux. Its design is threefold: a kernel-hook provides the movement information to a userspace daemon; that daemon then dispatches the information to userspace library instances that are linked to an application. This library ultimately implements the “Mobile IP API”.

A working implementation of the library is delivered. Testing has shown that this library works reliably; that applications are indeed notified when the node moves. Its design is simple and straightforward, and its interfaces are clearly defined. As a side effect, this library also provides a standard way for applications to retrieve the current home address and care-of address of the local node. Further work is suggested to disband the kernel-hook, and implement the movement detection part in the daemon, making the system more platform-independent and thus more portable. This bases the movement detection part on the “Advanced Sockets API”, which provides a POSIX compliant way for applications to obtain IP layer information.

# Table of Contents

<b>Foreword</b> . . . . .	i
<b>Abstract</b> . . . . .	ii
<b>Table of Contents</b> . . . . .	iii
<b>List of Acronyms</b> . . . . .	v
<b>1 Introduction</b> . . . . .	1
<b>2 Background and theory</b> . . . . .	4
2.1 Mobile IP . . . . .	5
2.2 Mobile IPv6 . . . . .	6
2.3 Movement in MIPv6 . . . . .	6
2.4 API drafts . . . . .	7
2.4.1 Extension to Sockets API for Mobile IPv6 . . . . .	7
2.4.2 Mobile IP API . . . . .	8
<b>3 Analysis</b> . . . . .	10
3.1 The MIPL Mobile IPv6 stack . . . . .	10
3.2 Analysis of the Mobile IP API . . . . .	11
3.3 Linux environment . . . . .	13
3.3.1 Mutual exclusion . . . . .	13
3.3.2 Parallel execution . . . . .	14
3.3.3 Communication between kernelspace and userspace . . . . .	14
3.3.4 Interaction with files . . . . .	15
3.3.5 Inter-process communication . . . . .	16
3.4 Scope of the work . . . . .	17
<b>4 Design</b> . . . . .	18
4.1 Overall structure . . . . .	18
4.2 Communication messages . . . . .	19
4.3 Communication channels . . . . .	20
4.4 Kernel part . . . . .	22
4.4.1 Interaction with MIPL . . . . .	22
4.4.2 Interaction with daemon . . . . .	23
4.5 Daemon part . . . . .	23
4.6 Userspace library part . . . . .	24
<b>5 Implementation</b> . . . . .	26
5.1 Communication . . . . .	26
5.2 Kernel part . . . . .	27
5.2.1 Interaction with MIPL . . . . .	28
5.2.2 Interaction with daemon . . . . .	29
5.3 Daemon part . . . . .	30
5.4 Userspace library part . . . . .	33

<b>6</b>	<b>Testing</b> . . . . .	37
6.1	Test environment . . . . .	37
6.2	Testing and results . . . . .	39
<b>7</b>	<b>Conclusions and recommendations</b> . . . . .	41
	<b>References</b> . . . . .	43
<b>Appendices</b>		
<b>A</b>	<b>Mobile IP API Internet draft</b> . . . . .	44

# List of Acronyms

<b>API</b>	Application Programming Interface
<b>IETF</b>	Internet Engineering Task Force
<b>IP</b>	Internet Protocol
<b>IPv4</b>	Internet Protocol version 4
<b>IPv6</b>	Internet Protocol version 6
<b>IPC</b>	Inter Process Communication
<b>FIFO</b>	First In First Out
<b>MIPv6</b>	Mobile IPv6
<b>RFC</b>	Request For Comment (Internet standard)
<b>SLP</b>	Service Location Protocol
<b>TCP</b>	Transmission Control Protocol

# 1 Introduction

This report describes the design and implementation of a movement notification mechanism for Mobile IPv6 to inform applications when the node moves. The term “node” is used to denote any machine on the Internet, including routers, PC’s, laptops and cell phones. A “mobile node” is a machine that can move, e.g. a laptop, PDA or cell phone. The reader is supposed to be familiar with IPv6 and the basics of the UNIX system.

These days, nodes on the Internet are becoming more and more mobile. This gives rise to the need to support this mobility at the IP network 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.

Routing on the Internet is done based on the network part of the IP address of the destination. The address of a node must always be location-correct (within the correct IP subnet) for the routing to work. That means that the network part of the IP address of the node corresponds to the network number of the network the node is connected to. As nodes can move to networks where their IP address is no longer topologically correct, that would break the routing, and the node would be unreachable. In order to continue communication, the node should change its IP address to an address that is topologically correct, each time it moves to another IP subnet. However, this conflicts with the needs of higher layer protocols as TCP, that rely on a constant IP address.

One of the achievements of Mobile IP is that the mobile node is always addressable by its

home address, regardless of what network the node actually is connected to. The topologically correct addressing is done by giving the mobile node 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 shielding is perfectly fine for most applications and protocols, 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 (connects to another network).

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. While TCP typically does not use any lower-layer information, it could use movement knowledge to avoid congestion and adapt to a new window-size quickly when the congestion (or *bandwidth · delay* product) on the new path to a node has changed significantly from the congestion (or *bandwidth · delay* product) on the old path.

Another usage could be an application that uses multicast, for instance for video streaming. When the home agent does not tunnel multicast packets to the mobile node, the mobile node has to subscribe to multicast groups in the foreign network. When the node moves to another network, it should re-subscribe to the multicast groups in order to provide smooth transitions. This can be done if the application using the multicast session gets a notification when the node has moved.

The goal of this work is to get a working implementation of a Movement Notification library for Mobile IPv6. It is specifically aimed at supporting applications running on a mobile node. The name *mobapi* is used to refer to this implementation. This report describes the

design and implementation of this movement notification library.

Section 2 describes the background of the work, including Mobile IPv6 and Internet drafts that can be used for this work. Section 3 describes MIPL, analyzes the Mobile IP API draft that is used, and describes the facilities in the Linux environment that are relevant for *mobapi*. Section 4 describes the design, containing the communication structure, and the division in kernel-part, daemon and library. Section 5 discusses the implementation of these parts, and describes in detail the functions that are used and their interactions. Section 6 describes the testing that is done and the results of it. Section 7 evaluates the work and gives recommendations for future work.

## 2 Background and theory

The main protocol suite that keeps the Internet running is TCP/IP (Transmission Control Protocol / Internet Protocol). IP is the network layer and takes care of delivering packets to their destination. It is a best-effort service that delivers packets based on their destination address.

Every host in an IP network has a permanent IP address, which is used to uniquely identify the host. In IPv4, this address is 32 bits long. This IP address is split into a network part (*network prefix*) and a host part. The network prefix (*subnet*) identifies the network the host is connected to, and the host part identifies the host within that network.

Routing is done based on the network prefix of the address. Packets are forwarded to the router that has the best route to that network. When the packet arrives in the destination network, it is delivered to the correct host.

Higher layer protocols, as TCP, rely on the IP address to maintain connections. To support a connection, a protocol must know the IP address of the node it communicates with. When that IP address is no longer reachable, the connection is dropped.

Because of the growing amount of hosts connected to the Internet, the address space of IPv4 is getting filled quickly. Several attempts have been made to counter this threat, the most notable being Classless Internet Domain Routing (CIDR) [4]. Nowadays however, the IETF has specified a new version of the IP standard, IPv6 [5]. Amongst others, IPv6 extends an address to 128 bits, providing a sufficient amount of addresses for everyone.

## 2.1 Mobile IP

Mobile IP aims at retaining IP connectivity even when a node is attached to another network than its own network. The own network is called the *home subnet*. A node has a permanent IP address in its home network, its *home address*. Normal IP routing will deliver packets that are addressed to the home address only to the node's home subnet. The node with which a mobile node communicates, is called a *correspondent node*.

When a node moves to another network (a *foreign network*), it obtains a new IP address (via a local address assigning technique, e.g. DHCP), which is topologically correct in the foreign network (the network prefix is in the subnet of the foreign network). This new address is called the *care-of address*.

The association between a home address and a care-of address is called a *binding*. The node then tells that binding to its *home agent*. This home agent is a router that sits in the node's home subnet. The home agent intercepts packets destined for the mobile node, and relays them to the node. This relaying is done by tunneling them to the care-of address.

When not careful, one gets *triangle routing*. That is, when the correspondent node always uses the home address to send data to the mobile node, it is always routed first to the home agent, and then from the home agent to the mobile node. The mobile node itself sends data directly to the correspondent node, which completes the triangle. Therefore, the mobile node can send a *binding update* to the correspondent node, which contains its care-of address, so a correspondent node can then address packets immediately to the care-of address, avoiding the communication via the home agent.

The higher-layer protocols can still communicate using the home address of the mobile node, for the replacement with the care-of address is done transparently in the IP layer. The higher-layer protocols will not notice at all that the node has moved (except for a short service interruption during the movement probably).

## 2.2 Mobile IPv6

Mobile IPv6 (MIPv6) specifies the following two extra IPv6 Destination options: Binding Update and Binding Acknowledgement.

Mobile IPv6 requires the correspondent node to maintain a cache of bindings that allows them to avoid triangle routing. Therefore a mobile node can send binding updates containing the latest care-of address to the correspondent node.

The transition from IPv4 to IPv6 gives the unique possibility to introduce new concepts into the protocol. While Mobile IPv4 was more or less a hack to allow mobility, and is not supported by all hosts on the Internet, Mobile IPv6 is an integral part of the IPv6 standard, and as such has to be supported by every host. This can happen because there is no installed base of hosts where it has to be compatible with, and now still IPv6 is mostly used on an experimental basis, so changes are allowed to happen. Thus a (correctly implemented) IPv6 network has native support for nodes to move.

## 2.3 Movement in MIPv6

Section 11.5 of the Mobile IPv6 draft standard [2] discusses movement. Basically, a node may use all mechanisms available to detect movement.

The section of most importance here is the Movement Detection section 11.5.1. This details the procedure for movement detection. The procedure mentioned here is to detect L3 (OSI layer 3, the network layer) movement, that is movement at the IP layer. L2 (OSI layer 2, the data link layer) movement is not detailed, as that is network-technology dependent.

In short, the node has moved when its old default router is no longer bi-directionally reachable. This is detected with the Neighbor Unreachability Detection procedure. When this movement is detected, a node should:

- perform Duplicate Address Detection (DAD) of the link-local address,
- select a new default router
- perform Prefix Discovery

- form a new care-of address
- register the new care-of address with its home agent
- send binding updates to its correspondent nodes.

When this procedure is ready, the node can continue to communicate with its correspondent nodes, and the higher-layer connectivity is retained.

## 2.4 API drafts

Two Internet drafts exist that are meant to supply information from the IP stack to applications. They are “Extension to Sockets API for Mobile IPv6” by Chakrabarti and Nordmark [6], and “Mobile IP API” by Yegin et al. [7]. The first one is aimed at Mobile IPv6, while the second one is meant for Mobile IP in general. They are discussed here.

### 2.4.1 Extension to Sockets API for Mobile IPv6

The “Extension to Sockets API for Mobile IPv6” draft [6], which is on track to become an RFC now, provides a mechanism for API access to retrieve and set information for the fields that are added in Mobile IPv6 with respect to IPv6. The “Advanced Sockets API for IPv6” RFC by Stevens et al. [8] specifies API access to the IPv6 fields, and this draft extends it to incorporate Mobile IPv6 fields. These fields are the Mobility Header, the Home Address Destination option and the Type 2 Routing Header. It aims towards POSIX compliance.

The target applications for this API are believed to be debugging and diagnostic applications as well as policy applications that would like to monitor the protocol information at the application layer.

The draft defines the constants and data structures for C programs that specify the data fields in the Mobile IPv6 packets. It also defines the API to access the added fields. This is done by using raw sockets.

## 2.4.2 Mobile IP API

The “Mobile IP API” draft [7] specifies an interface between the IP layer and the application layer. It provides mobility information about the host that applications are running on (mobile node functionality) or other hosts they are communicating with (correspondent node functionality). It introduces the concept of a Mobility Management Module to keep track of all mobile nodes that are known to the local node. Amongst others it knows the binding updates and the movement information of hosts.

The draft outlines some usage scenarios where knowledge of the movement of a mobile node can be advantageous. Most of these scenarios are mentioned in Section 1. It then specifies the requirements for the Mobile IP API design:

- provide location awareness
- provide movement awareness
- not disturb other applications (read-only functionality)
- not generate extra signaling or change protocols.

A mobile node is identified in [7] by these parameters:

- its home address
- its care-of address
- the address of its home agent.

It specifies a structure in which these parameters are captured.

It then outlines three functions, to get information of mobile nodes and to get movement notification. The function `mip_get_all_mobile_nodes()` returns a list of all nodes that are known by the mobility management module. The function `mip_get_one_mobile_node()` returns information about one mobile node specified by its home address. The function `mip_notify_movement()` provides a notification when a specified mobile node moves. The macro `IS_AT_HOME()` is used to determine whether a given mobile node is in its home network.

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.

## 3 Analysis

In this chapter, platform specific issues will be discussed. The MIPL Mobile IPv6 stack will be discussed, the Mobile IP API will be analyzed, and the facilities that the Linux environment provides are discussed. Finally the scope of the work will be more precisely defined.

### 3.1 The MIPL Mobile IPv6 stack

Several Mobile IPv6 implementations exist [9], but for this work, the MIPL implementation will be used, mainly because it was already in use in the project environment. Mobile IPv6 for Linux (MIPL) [10] is an implementation of a Mobile IPv6 stack for the Linux platform. The version discussed here is version 1.0, which is a patch against Linux kernel version 2.4.22. It is developed by Helsinki University of Technology.

It provides three main modules:

- `mip6_cn`, correspondent node functionality
- `mip6_ha`, home agent functionality
- `mip6_mn`, mobile node functionality.

The correspondent node functionality is needed on every IPv6 node that is communicating with a mobile node. The home agent functionality is needed only in a node that functions as a home agent. The mobile node functionality is needed in all mobile nodes.

## 3.2 Analysis of the Mobile IP API

The Mobile IP API draft describes a mobile node's status with this structure:

```
struct mobile_node_t {
    struct in6_addr home_addr; /* home address of MN */
    struct in6_addr co_addr;   /* care-of address of MN */
    struct in6_addr ha_addr;   /* MN's home agent */
};
```

The node is identified by its home address. Its status is determined by the combination of home address and care-of address. The home agent address is provided as extra information.

The main functionality specified in the draft is the function `mip_notify_movement()`. This function provides a notification to the application that invoked it when the specified node has moved. Its specification is:

```
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 node for which notification of movement is asked is specified in the field *mobile\_node*. When it is set to the unspecified address (0::0), it is treated as a wildcard to get notifications for all mobile nodes.

The function `mip_notify_movement()` specifies two modes: *blocking* and *non-blocking*. In blocking mode, the function waits until the asked-for mobile node has moved or *timeout\_ms* milliseconds have passed (whichever occurs first), and then returns. This effectively stalls the application until the node has moved. When it returns, the parameter

*mobile\_node* is filled with the latest status of the node.

In non-blocking mode, the application has to provide a callback function, that is called when the node has moved. The function `mip_notify_movement()` returns immediately, allowing the application to continue running. The content of *mobile\_node* is then unspecified. When the node has moved, the callback function provided by the application is called. It is passed an up-to-date *mobile\_node* structure. When `mip_notify_movement()` is called in non-blocking mode with the *callback* parameter set to NULL, it is interpreted as a callback deregistration.

The parameter *cb\_parameter* can be passed to `mip_notify_movement()`, that is passed on to the callback function. This parameter is not used in `mip_notify_movement()`. The parameter *event* indicates the reason for the callback. Its value can be MIP\_MN\_MOVED, MIP\_CB\_DEREGISTER or MIP\_BCE\_DEREGISTER. According to [7], MIP\_MN\_MOVED signals that a binding update is received, that indicates that the mobile node has obtained a new care-of address. This is an incomplete definition that only applies to correspondent node operation. On a mobile node, the movement detection is done by other means, that actually trigger the sending of the binding updates to correspondent nodes. This local movement detection also results in a MIP\_MN\_MOVED event. MIP\_CB\_DEREGISTER is used when the callback is deregistered, because when that happens, the callback function is called one last time. It can also indicate that the home address for which the callback had been registered is deleted. MIP\_BCE\_DEREGISTER indicates that the binding update has been deleted, for instance because of expiration of the lifetime. This is again only useful in the case where binding updates are involved, i.e. in the correspondent node operation.

An application should be able to register multiple callbacks, distinguished by different home addresses, different callback functions, different values of *cb\_parameter* or any combination of these. One use-case could be that the application wants to register interest in several different mobile nodes. Or it could be that the application is composed of several classes or threads, that each individually want to register a callback function. Another case could be that the application registers one central callback function, that dispatches movement notifications based on the value of *cb\_parameter* for different instances. Therefore all callbacks registered for a certain mobile node have to be called.

This analysis boils down to the following requirements. There must be a library that can be linked with the application wanting to receive the notification. That means this library must be in userspace. This library must know the movement information. The library must also know the current status of the nodes. As the draft specifies that a blocking call with a 0ms timeout must return the current status, the library must also know this status before the node has moved. This also holds for a callback registration that is immediately followed by a callback deregistration. Also, in this case the current status must be passed to the application.

### 3.3 Linux environment

The Linux environment provides several facilities and techniques that are used in this work. They will be shortly outlined in this section. Most of the information provided here is extracted from [11] and [12] or from resources on the Internet.

#### 3.3.1 Mutual exclusion

Proper mutual exclusion is required for access to global variables in the kernel, because the kernel can be multitasked. Mutual exclusion means that two or more concurrent processes cannot access a variable at the same time. A basic example illustrating this problem is when two simultaneous processes need one shared variable. Process 1 reads the variable, and right after that, process 2 gets to run. Process 2 reads and modifies the shared variable, and then the first process gets to run again. Process 1 then takes action upon an outdated value of the variable. With mutual exclusion, process 2 is not allowed to access the variable as long as process 1 holds the lock to that variable. For this purpose, the kernel provides *spinlocks* and *semaphores* [13].

A spinlock is a very simple single-holder lock: if you can't get the spinlock, you keep trying (spinning in a busy loop) until you can. Spinlocks are very small and fast, and can be used anywhere. They also do not sleep, which is required for interrupt context.

Semaphores are more complex, as there can be more than one holder at a time. A semaphore

is basically a counter indicating the number of resources available. If this number is 1 (the most common case), the semaphore is often called a *mutex*. A spinlock is therefore also called a mutex. If a process cannot get the semaphore, it will be put in the wait queue, and thus sleep.

### 3.3.2 Parallel execution

Linux provides two ways of parallel execution: parallel processes and parallel threads (multithreading) within a process. A parallel process can be created by calling `fork(2)`. Then the parent process is cloned, and a child process is created. These processes are identical, and only the result of the call to `fork()` differs.

A thread is used for parallel execution within one process. Linux provides the *linuxthreads* library to support multithreading. This is a POSIX compliant threading library.

A daemon is a process that runs in the background and is not attached to a console. It performs tasks that do not require much processing. They usually run for a long time, as they are started when the system boots, and only terminated when the system shuts down.

### 3.3.3 Communication between kernelspace and userspace

The UNIX system provides a strict separation between kernelspace and userspace, also called kernelland and userland. The kernelspace runs with superuser (root) privileges, and has thus complete control over the system. The userspace has restricted privileges. This separation is mainly for security reasons.

The most straightforward way to communicate between kernelspace and userspace would be to use a function call. However, this is not possible because of the strict separation between kernelspace and userspace. It is not possible to do function calls from userspace to kernelspace and vice versa.

The other option is to use some communication channel [12]. Possibilities are:

- a character device driver (/dev entry)

- a `proc-fs` /`proc` entry
- a `sysfs` entry.

Both a `procfs` and a `sysfs` entry export internal values from a kernel driver to userspace. `Procfs` is deprecated as of kernel version 2.6. `Sysfs` can only export ASCII data. Both map to local variables inside the kernel. That means that both are not really suitable to pass messages.

A character device driver [12, chapters 3 and 5] exports the device that it manages via a file-entry in the `/dev` file system. Applications can use common operating system calls on it, and the device driver has to implement these system calls. This gives the driver complete flexibility and control over what it allows to happen. The device driver exports the functions it supports to the operating system when it registers the device. A wide variety of functions can be supported.

### 3.3.4 Interaction with files

When a file-interface is used, the following options are available to interact with that file [11, chapter 12.5]:

- `poll(2)` waits for an event on a number of file handlers
- `select(2)` waits for an event on a number of file handlers
- `ioctl(2)` invokes special operations on a file.

The functions `select()` and `poll()` provide more or less the same functionality, but the latter is the newer interface. Internally in the kernel, `select()` is implemented as a wrapper around the internals of `poll()`. The function `ioctl()` is not such a nice way to use, for it has a quite unclear interface.

When using `poll()`, one specifies an array of descriptors, where each descriptor consists of the file descriptor to check (input), the events of interest (input), and the events that occurred (output). As long as the events of interest do not happen, `poll()` puts the process to sleep, so that it does not use any processor resources. When an event of interest happens on one of the specified input file descriptors, it wakes up the application and returns, indicating which file descriptor has changed and what event has happened. This information can then

be used to perform the desired action on the file descriptor.

For instance, one could use it to poll whether a file becomes readable (then there is data available in the file, so probably another process has written data to it). When `poll()` indicates that the file is readable, the application can read from the file. Thus the application does not need to check continuously itself, but can delegate the checking to `poll()`.

### 3.3.5 Inter-process communication

The UNIX environment provides several ways for inter-process communication (IPC) [11, chapter 14]. They are amongst others:

- pipe
- FIFO
- message queue
- semaphore
- shared memory
- stream pipe
- named stream pipe
- socket
- stream.

Only the first two are interesting for this work, and are discussed here.

A pipe is the simplest and oldest form of IPC. It provides half-duplex communication between processes with a common ancestor. As such, its use is limited, but widely used for instance for piping inside the shell. There the shell creates the pipes, and then forks to execute the specified commands, attaching those pipes to `stdin` and `stdout`. It can be used by having the parent create two pipes, a read and a write pipe. Then this parent forks, thereby cloning the pipes to its child. The child can then use these pipes to communicate with its parent.

A FIFO is a named pipe. A normal pipe is not represented in the file system, but a named pipe gets a file name, and can thus be found in the file system. Therefore also the rules and operations for files apply on FIFOs. A normal pipe can only provide a communica-

tion channel between related processes (parent and child), but with a FIFO also unrelated processes can exchange information.

There are two problems with using FIFOs that one has to be aware of. One is that if an application writes to a FIFO that is not opened by any process to read from it, a SIGPIPE signal is generated, which can abort the application if that signal is not handled. The second is that when the last writer for a FIFO closes the FIFO, an end-of-file is generated for the reader of the FIFO, and `poll` returns with POLLHUP, which can disturb the normal functioning of the program. The solution for the first problem is to catch the SIGPIPE signal, and handle it (probably by doing nothing). The solution for the second problem is to open the FIFO for read-write instead of read-only, meaning that the reading application is also a writer. This way no end-of-file is generated, for there is always a writer left.

### 3.4 Scope of the work

The “Mobile IP API” provides an API to applications in order to provide notifications when a mobile node has moved. As this is the intent of this work, this API will be used as interface to the application. The “Advanced Sockets API” provides a good base for the work, but there is no implementation of it yet for MIPL. The duration of this project is too short to also implement this API, so the movement information will be extracted from the kernel via other means.

This work does not implement the whole Mobile IP API specification. The goal is to provide movement notifications to the application. It is done by implementing the movement notification part of the Mobile IP API draft, in the function `mip_notify_movement()`. It will provide blocking and non-blocking functionality to the application.

## 4 Design

In this chapter, the design of the work will be discussed. It first covers the overall structure and the messages that are exchanged, and then discusses the design of the parts.

### 4.1 Overall structure

The purpose of this work is to provide an API for movement notification to applications. This means that there must be a library that can be used by applications, which thus has to be in userland. This library is called *libmobapi*. It should be possible for multiple running applications to all receive notifications. Thus there should be multiple instances of the library running at the same time.

This library has to know movement information. The only place this information is known, is in the kernel. The Mobile IPv6 stack detects when a node has moved. As this information is not exported in any way to userspace, some way has to be found to do that. This is done by building a hook into the kernel's Mobile IPv6 stack. The kernel hook is called *mobapi\_kernel*.

Then, because it should be possible to have multiple instances of the library running at the same moment, some program has to keep track of that. The kernel should not do that, because to implement that, one needs multiple communications channels to all library instances. The solution is to use a daemon for that. This daemon keeps track of the applica-

tions that are interested in receiving movement notifications, and dispatches the movement notifications to them. The daemon is called *mobapid*.

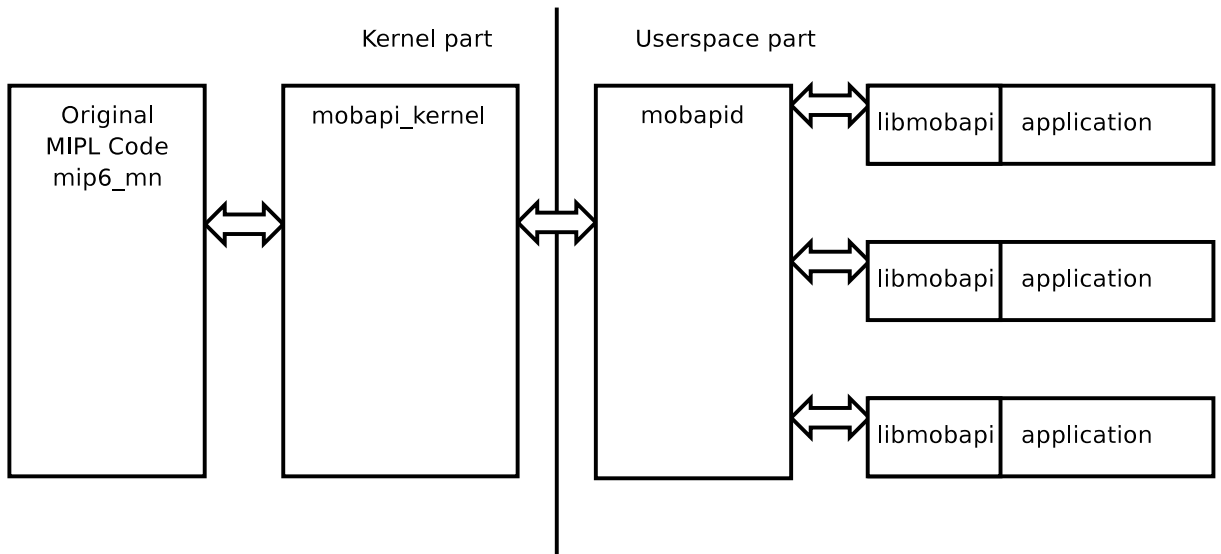


Figure 4.1: The overall structure of *mobapi*

This subdivision is shown in Figure 4.1. It is clear here that there can be multiple applications running, each being linked to *libmobapi*. The daemon concentrates all connections from the libraries and forms the communication channel to the kernel part. The means of communication are discussed in the next part.

## 4.2 Communication messages

For communication among the various parts, messages are passed. These messages are:

- NODE\_MOVED
- NODE\_INFO
- SUBSCRIBE
- UNSUBSCRIBE

All messages contain the parameters of a mobile node as specified before (home address, care-of address and home agent's address).

The first message, *NODE\_MOVED*, is used as an indication that the node mentioned in the message has moved. It is passed from the kernel to the daemon, and from the daemon to

the library. It is only sent when the node has moved.

The second message, `NODE_INFO`, is needed because, as discussed before, the API can be used to obtain the current parameters of a mobile node without any movement occurring. This message is also passed from the kernel to the daemon and further to the library. It is sent from the kernel to the daemon immediately when the daemon opens communication with the kernel. The daemon then sends it further to the library immediately when the library opens communication with the daemon.

The last two messages, `SUBSCRIBE` and `UNSUBSCRIBE`, are only sent from the library to the daemon. The library sets up an association with the daemon by sending a `SUBSCRIBE` message, that also includes information about the used communication channel. Then the daemon knows how to reach the library to send messages to the library. The library can terminate an association by sending an `UNSUBSCRIBE` message.

An example of the flow of messages is shown in Figure 4.2. It shows a daemon handling two applications: one that wants to receive several movement notifications, and the other 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.

## 4.3 Communication channels

Two different mechanisms are used as communication channel. The communication between kernel and daemon goes via a character device driver. This device driver is implemented in the kernel, and manifests itself by a file-entry in the `dev-fs` file structure. This file can be opened by the daemon, that then can send and receive messages via that channel.

The communication between daemon and library goes via a FIFO. The daemon creates a FIFO with a well-known filename, and the library can open that and write messages to it. This way, all instances of the library have a way to send messages to the daemon. To receive messages, the library has to create its own private FIFO. Other applications are not allowed to access this private FIFO. Only the daemon can write to that FIFO because the daemon is running as root, so it is a safe way to send messages from the daemon to the library.

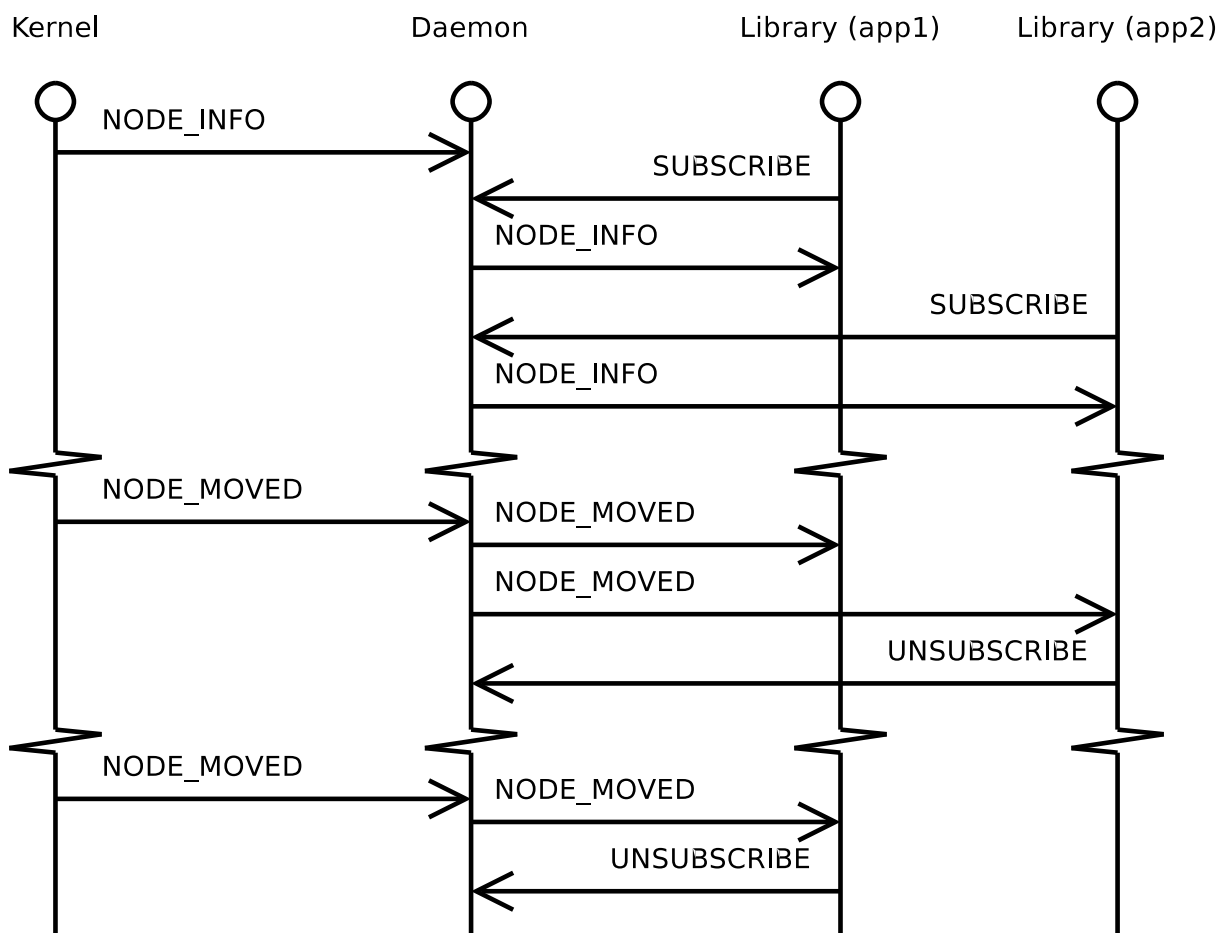


Figure 4.2: A possible flow of messages

This design is chosen because the library has no other way to discover a running daemon and set up a communication channel with it, than with previous knowledge on how to reach the daemon. This knowledge involves the filename of a FIFO in this case. Via this FIFO, the daemon can be reached, and when that FIFO cannot be opened, there is no daemon running.

The library uses a private FIFO to receive messages back from the daemon, because if the same well-known FIFO would be used, all library instances would receive all messages that the daemon sends to a library instance. This is not desirable for security concerns, so a private FIFO is used that only the daemon can write to and the library can read from.

This mechanism is based on [11, chapter 14.5 FIFOs].

## 4.4 Kernel part

The kernel part, *mobapi\_kernel*, consists of two parts: the interface to MIPL and the interface to the daemon *mobapid*. The interface to MIPL implements the kernel hook to obtain information from the IP layer, that is movement information and addresses. The interface to the daemon contains an interface for passing messages to the daemon.

The kernel part has one important datastructure, that is a queue containing messages that have to be sent to the daemon. It also has the home address, care-of address and home agent's address as global variables. This queue and the addresses are global and thus have to be protected from simultaneous access.

### 4.4.1 Interaction with MIPL

The interface to MIPL consists of three functions that are called by MIPL (startup and exit functions and a function that is called when the node has moved), and a function that calls MIPL code to obtain the current status as IPv6 addresses.

The startup function `mobapi_init()` is called when the MIPL module *mip6\_mn* is loaded. It sets up the message queue and registers the character device driver interface with the operating system.

The exit function `mobapi_exit()` is called when *mip6\_mn* is unloaded. It cleans up the message queue, and unregisters the character device.

The function `mobapi_mobile_node_moved()` is called by MIPL when it detects that the node has moved. It gets the current node status from MIPL by calling `mobapi_get_addresses()`, and then queues a `NODE_MOVED` message for sending to the daemon.

The function `mobapi_get_addresses()` is used to obtain the current status of the mobile node from MIPL.

## 4.4.2 Interaction with daemon

The interface with the daemon consists of five functions that implement standard operations on files. These functions are `mobapi_open()`, `mobapi_release()`, `mobapi_read()`, `mobapi_write()` and `mobapi_poll()`. They are called when an application calls standard UNIX functions that operate on files, respectively `open(2)`, `close(2)`, `read(2)`, `write(2)` and `poll(2)`. Not all possible file operations that UNIX supports are implemented, because they are not all needed. The only functions needed here are those necessary to transfer messages to the daemon, and possibly to receive messages from the daemon, though the design does not use that last feature. References to these functions are put in a structure that is passed to the operating system as part of the registration of the character device driver.

When the function `mobapi_open()` is called, that is when an application calls `open()` on the character device, a `NODE_INFO` message is queued for transmission. It also increases the usage count of the kernel module. The function `mobapi_release()` only decreases the usage count. The function `mobapi_read()` passes one message from the message queue to the application calling it. It then removes the sent message from the message queue. The function `mobapi_write()` is only provided for testing purposes. By writing to the device, movement can be simulated (i.e. `mobapi_mobile_node_moved()` is called). The function `mobapi_poll()` implements the `poll()` system call that is discussed before. It returns 'readable' when there is a message waiting in the message queue.

## 4.5 Daemon part

The daemon *mobapid* acts as a relay between the kernel part *mobapi\_kernel* and instances of the library *libmobapi*. Messages from the kernel are relayed to the connected libraries. Most of the time it waits for input from the communication channels to the kernel (character device) and the library (well-known FIFO).

The daemon keeps track of the connected libraries in a list of clients. The home address of the node for which the library wants to receive notifications is stored in that list, and information about the communication channel to the library.

The daemon listens on the character device driver interface to the kernel to receive `NODE_INFO` and `NODE_MOVED` message from *mobapi\_kernel*, and it listens on its well-known FIFO interface from *libmobapi* to receive `SUBSCRIBE` and `UNSUBSCRIBE` messages.

When an application registers interest in movement (via the library) by sending a `SUBSCRIBE` message, the daemon adds that application to the client list. Conversely, when the application deregisters itself by sending an `UNSUBSCRIBE` message, the registration is removed from the client list. When a client is registered, the daemon opens its communication channel to the library and sends a `NODE_INFO` message. Then, when the daemon receives a `NODE_MOVED` message from the kernel, it resends that message to all clients that are interested in movement of that node.

## 4.6 Userspace library part

The library *libmobapi* implements the part of the work that is visible to applications: the function `mip_notify_movement()` and the macro `IS_AT_HOME()`. The macro is trivial, but the function `mip_notify_movement()` involves a bit more thinking. As discussed in Section 3.2, `mip_notify_movement()` supports two modes of operation: blocking and non-blocking.

When `mip_notify_movement()` is called in blocking mode, *libmobapi* sends a `SUBSCRIBE` message to the daemon, waits until it receives a `NODE_MOVED` message or a timeout occurs, sends an `UNSUBSCRIBE` message to the daemon, and returns with the latest node status. After it has sent the `SUBSCRIBE` message, it will immediately receive a `NODE_INFO` message. This message is used in order to return the node status when the timeout occurs while the node has not yet moved.

A non-blocking call to `mip_notify_movement()` can be either a callback registration or deregistration. When it is a registration, the callback is added to the list of callbacks that the library maintains and a `SUBSCRIBE` message is sent to the daemon. To be able to provide non-blocking functionality, POSIX threads are used. After sending the `SUBSCRIBE` message, a thread is started to wait for messages (the poll-thread). Only one poll-thread is used per library instance, so if the poll-thread already exists, it will not be created. In the

meantime `mip_notify_movement()` returns, letting the application continue its work. When the poll-thread receives a `NODE_MOVED` message, it calls the callback function. Then it continues waiting for messages from the daemon.

When the call to `mip_notify_movement()` indicates a callback deregistration, an `UNSUBSCRIBE` message is sent to the daemon, and the callback function is called one last time. When there are no callbacks registered any more, the poll-thread is stopped.

## 5 Implementation

This section discusses the implementation aspects of *mobapi*. It first discusses how the communication is implemented, and then discusses the implementation of the three subparts of *mobapi*: *mobapi\_kernel*, *mobapid* and *libmobapi*.

### 5.1 Communication

Two communication channels mechanisms are used: the character device driver and the FIFO mechanism. The character device driver created by the kernel is mapped to the file */dev/mobapi* and requires root privileges to be accessed. The well-known FIFO created by the daemon is */tmp/mobapi.in* and is writable by all applications. The private FIFO created by the library is implemented by appending the process ID (pid) of the application to the base filename to make the filename unique, resulting in the name */tmp/mobapi.\$PID*. This FIFO is only readable by the application itself, and not writable at all. However, programs running with root privilege can write to it, so the daemon can write to it.

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 types were introduced before. The main parameters are the addresses specified as the status of a mobile node, the home address *home\_addr*, care-of address *co\_addr* and home agent's address *ha\_addr*.

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.

## 5.2 Kernel part

The kernel part maintains a global linked list of messages that are to be sent to the daemon. This list has a head and a tail pointer to facilitate removing from the front of the list and adding to the end. These pointers are protected with a spinlock.

It also maintains the three addresses (home address, care-of address and home agent's address) parameters as global variables. These addresses are also protected with a spinlock.

Finally it has a global *wait\_queue*, which is needed for the *mobapi\_poll()* function, which needs not be protected.

It has a couple of internal helper functions. They are *send\_node\_info*, *add\_message* and *remove\_message*. The first function retrieves the information needed for a

NODE\_INFO message by calling `mobapi_get_addresses()`, and then queues that message in the message queue by calling `add_message`. The function `add_message` adds the given message to the end of the message queue. The function `remove_message` removes the first message from the message queue. During a call to either of these functions, one must hold the spinlock for the message queue.

### 5.2.1 Interaction with MIPL

The interface from *mobapi\_kernel* to MIPL consists of four functions: `mobapi_init()`, `mobapi_exit()`, `mobapi_mobile_node_moved()` and `mobapi_get_addresses()`.

The function `mobapi_init()` is called from the MIPL function `__init mip6_init()`, which is run when the module *mip6\_mn* is loaded. It initializes the `wait_queue` and the message queue, and registers the character device.

The function `mobapi_exit()` is called from the MIPL function `__exit mip6_exit()`, that is executed when the module *mip6\_mn* is unloaded. It empties the message queue and unregisters the character device.

The function `mobapi_mobile_node_moved()` is called from the MIPL function `mip_mobile_node_moved()`, that is called when the mobile node has moved and during the initialization of *mip6\_mn*. The function `mip_mobile_node_moved()` takes care of sending binding updates to the home agent and correspondent nodes. At the end it calls `mobapi_mobile_node_moved()`. This function first retrieves the latest addresses by calling `mobapi_get_addresses()`, and then queues a NODE\_MOVED message in the message queue by calling `add_message()`. Finally it wakes up the `wait_queue` for `mobapi_poll()`. When the function is called during initialization, no message is queued; it just returns.

The function `mobapi_get_addresses()` is not called from MIPL, but from within *mobapi\_kernel*. It calls the MIPL functions `MIPV6_CALLPROC( mipv6_get_home_address )` to get the home address and `mipv6_mn_is_home_address()` to check the result. It then calls

`mip_get_care_of_address()` to get the care-of address. The obtained addresses are stored in the global address variables.

### 5.2.2 Interaction with daemon

The interaction with the daemon goes via the five functions `mobapi_open()`, `mobapi_release()`, `mobapi_read()`, `mobapi_write()` and `mobapi_poll()`. They are bundled in the *struct file\_operations* structure that is passed to the operating system when the character device is registered.

The function `mobapi_open()` is called when an application calls the operating system call `open()` on the character device entry: `open("/dev/mobapi")`. The module usage count is increased, and a `NODE_INFO` message is queued in the message queue by calling `send_node_info()`.

The function `mobapi_release()` only decreases the module usage count.

The function `mobapi_read()` copies one message from the queue to the buffer supplied by the application. It first copies the first message in the queue to a local variable and removes it from the queue by calling `remove_message`. Then it copies this message to the buffer that the application provided and returns.

The function `mobapi_write()` only calls `mobapi_mobile_node_moved()`, which is used for testing purposes.

The function `mobapi_poll()` is called when an application calls `poll()` on a file descriptor of */dev/mobapi*. It adds the current process (the application that invoked the function) to the `wait_queue` associated with that file descriptor, and then determines a bit-mask representing the state of the file. In this case, it is readable only when there are messages waiting in the message queue. This bit-mask is returned.

## 5.3 Daemon part

The daemon consists of a lot of functions. The most important functions are `main()`, `process_dev_entry()` and `process_in_fifo()`. Then there are a lot of helper functions: `open_fifo()`, `send_node_info()`, `send_moved()`, `read_message()`, `read_command()`, `remove_fd_from_client_list()` and `die()`. The names of most of these functions speak for themselves, so these functions are not discussed separately.

The daemon maintains a double linked list of *clients*, where clients are library instances that have subscribed to movement notifications. They are 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;
};
```

This structure contains the file descriptor and FIFO number of the FIFO that is the communication channel to a connected library instance. Furthermore, it contains the home address of the mobile node that the application is interested in. This is a global list, but because the daemon is single-threaded, it need not be protected for mutual exclusion. Also the three addresses of the mobile node are maintained as global variables.

The daemon starts by running `main()`, of which a condensed version (some comments and debug code is stripped) is shown below. It first attaches the function `die()` (that closes the open files and removes the well-known FIFO) to the signals `SIGTERM`, `SIGINT` and `SIGQUIT`, so stopping the daemon can be done by `kill()`ing it. Then it daemonizes itself by calling `daemon_init()`. The function `daemon_init()` forks itself into the background, and lets the parent exit. It is then detached from the terminal, so it cannot receive input from the terminal any more. Also the signal `SIGPIPE` is caught with the dummy function `SIG_IGN()`. Then the character device `/dev/mobapi` is opened and the

well-known FIFO */tmp/mobapi.in* to the library is created. This FIFO is opened for read and write, to prevent `poll()` to return `POLLHUP` when all clients close it. After that, an infinite loop starts, `poll()`ing the two communication channels. When input occurs on the character device, the function `process_dev_entry()` is called to handle that, and when input occurs on the well-known FIFO, the function `process_in_fifo()` is called to handle that.

```
int main(int argc, char *argv[] )
{
    struct pollfd fdarray[2];
    clients = NULL;

    signal(SIGTERM, die);
    signal(SIGINT, die);
    signal(SIGQUIT, die);
    signal(SIGPIPE, SIG_IGN );

    daemon_init();
    dev_fd = open( DEVFILE, O_RDWR );
    mkfifo( INFIFO, S_IWOTH);
    in_fd = open( INFIFO, O_RDWR | O_NONBLOCK );

    while( 1 ){
        /* listen to fifo and /dev entry */
        fdarray[0].fd = dev_fd;
        fdarray[0].events = POLLIN | POLLRDNORM;
        fdarray[0].revents = 0;
        fdarray[1].fd = in_fd;
        fdarray[1].events = POLLIN | POLLRDNORM;
        fdarray[1].revents = 0;

        poll( fdarray, 2, -1);
    }
}
```

```

if( (fdarray[0].revents & (POLLIN | POLLRDNORM)) ==
        (POLLIN | POLLRDNORM) ){
    /* dev-entry readable */
    process_dev_entry( dev_fd );
}
if( (fdarray[1].revents & (POLLIN | POLLRDNORM)) ==
        (POLLIN | POLLRDNORM) ||
        (fdarray[1].revents & POLLHUP) == POLLHUP ){
    /* in-fifo readable */
    process_in_fifo( in_fd );
}
} /* while(1) */

/* this code is never reached */
return 0;
}

```

The function `process_dev_entry()` reads a message from the dev-entry by calling `read_message()`. Then, depending on the *type* of the message, it either stores the information (for a `NODE_INFO` message), or relays the movement information to the connected clients (for a `NODE_MOVED` message). This also stores the information and then sends the message to all clients that have subscribed to movement information of that node. This sending is done by calling `send_moved()`, and if that returns an error (e.g. because the outgoing FIFO is closed as result of a crash of the client), the function `remove_fd_from_client_list()` is called to remove the registration from the list of clients.

The function `process_in_fifo()` retrieves the incoming message from the well-known FIFO by calling `read_command()`. The type of the message can be either `SUBSCRIBE` or `UNSUBSCRIBE`. When it is a `SUBSCRIBE` message, the asked-for home address is checked to see whether that home address is supported (this implementation only supports the home address of the local node, or the unspecified address `0::0`). If that home address is valid, the FIFO to the client is opened by calling `open_fifo()`, the client is added to the client list, and `send_node_info()` is called to send the node's status to

the client. When it is a UNSUBSCRIBE message, the client belonging to that home address and FIFO is looked up in the client list, the FIFO is closed, and the client registration is removed from the client list.

## 5.4 Userspace library part

The library provides the interface to the application. This interface consists of the function `mip_notify_movement()` and the macro `IS_AT_HOME()`. It omits the functions `mip_get_all_mobile_nodes()` and `mip_get_one_mobile_node()` that are also specified in [7].

The macro `IS_AT_HOME()` is not more than a wrapper around the system macro `IN6_ARE_ADDR_EQUAL()` on the home address and care-of address of a mobile node. When the care-of address is equal to the home address, the node is at home. It is implemented as:

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

The parameter *t* is here a *mobile\_node\_t* structure.

The function `mip_notify_movement()` is the main function of interest. The declaration of the function is:

```
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 input field *home\_addr* of parameter *mobile\_node* denotes the home address of the node for which the application wants to receive a movement notification. When that field is set to the unspecified address (0::0), it indicates that the application is interested in notification for all nodes that are known to the library. When the function returns from a blocking call, the fields of *mobile\_node* are set to the current values of the addresses. If the *home\_addr* was 0::0, then it is set to the home address of the node that has moved.

The parameter *non\_blocking* indicates whether the application wants to use the blocking or non-blocking mechanism. The parameter *timeout\_ms* indicates the timeout in milliseconds after which a blocking call has to return. When it is set to -1, it is interpreted as infinity, which disables the timeout mechanism.

The parameter *cb\_parameter* is stored for passing to the callback function. It is not interpreted in the library.

The callback function is defined as:

```
int callback( mobile_node_t mobile_node,
             int event, long int cb_parameter);
```

The application has to provide a function with this interface, and pass the pointer to that function to `mip_notify_movement()` when it requests non-blocking movement notifications. Note that [7] specifies that the *mobile\_node* parameter is passed by value, not by reference. The parameter *cb\_parameter* gets the same value as the parameter that was passed to `mip_notify_movement()`. The possible values of *event* are discussed in section 3.2.

The library maintains a double linked list of callback functions where the home-address of the node, the pointer to the callback function, and the callback parameter are stored. It is defined as:

```
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;
};

```

When the pointer to the callback function is NULL during a non-blocking call to `mip_notify_movement()`, this denotes a callback deregistration. The registered callback function for the specified home-address is called one last time, with *event* set to `MIP_CB_DEREGISTER` and *mobile\_node* set to the latest binding, and then the callback is deregistered.

The library also contains a couple of helper functions, `add_callback()`, `remove_callback()`, `send_subscribe()`, `send_unsubscribe()`, `open_fifos()`, `do_callback()` and `do_poll()`. Their names are descriptive, so they are not discussed in detail here. The first two functions add and remove a callback from the callback list. The second pair of functions send a message to the daemon. The function `open_fifos()` creates and opens the FIFO to the daemon, and opens the well-known FIFO from the daemon. The function `do_callback()` does the callbacks to the application. It searches the callbacks in the list of callbacks, and then calls the callback functions.

The function `do_poll()` is an important function. It is the function that runs in the poll-thread when non-blocking mode is used. This function just `poll()`s on the FIFO from the daemon, to receive `NODE_MOVED` messages. When such a message is received, the function `do_callback()` is called to perform the callback. It ends when the list of callbacks is empty, effectively also ending the poll-thread.

The main outline of the function `mip_notify_movement()` is this:

```

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

```

During the registration of a callback, blocking calls are not allowed.

In blocking mode, first the FIFOs are opened by calling `open_fifos()`. Then a `SUBSCRIBE` message is sent by calling `send_subscribe()`. Then a loop with `poll()` is

entered, waiting for messages from the daemon. When a timeout occurs (if that is specified), an UNSUBSCRIBE message is sent by calling `send_unsubscribe()`, and the function returns with return-value 0, and has *mobile\_node* set to the latest binding. But even with a timeout of 0 ms, first a NODE\_INFO message has to be received, to be able to pass the latest binding back. When finally a NODE\_MOVED message is received, an UNSUBSCRIBE message is sent to the daemon by calling `send_unsubscribe()`, and the function returns with a non-zero return value (1).

In non-blocking callback registration, first `open_fifos()` is called when there is no thread running yet. Then the callback is added to the callback list by calling `add_callback()`. Then a SUBSCRIBE message is sent, and if the poll-thread is not existing yet, this thread is created, which starts `do_poll()`. Then `mip_notify_movement()` returns.

In a non-blocking callback deregistration, first a callback is done by calling `do_callback()`, with *event* set to MIP\_CB\_DEREGISTER. Then an UNSUBSCRIBE message is sent via `send_unsubscribe()`, and `remove_callback()` is called to remove the callback from the callback list. When `do_poll()` detects that there are no callbacks left, it stops, effectively also stopping the poll-thread. When that thread is stopped, the FIFOs are closed, and `mip_notify_movement()` returns.

## 6 Testing

This section describes the testing that is done to verify the implementation. The first part discusses the setup of the test-environment, describes the test network and discusses problems that arose while setting up the test-environment. The second part discusses the tests that were performed, and their results.

### 6.1 Test environment

As Mobile IPv6 is still in an early stage, setting up the environment is not that obvious. Mobile IPv6 requires three machines:

- mobile node
- home agent
- correspondent node.

The desired testing environment is one in which it is possible that a mobile node roams (moves) to several networks.

The mobile node *mn-2* is a laptop running Debian GNU/Linux (the 'unstable' version) with Linux 2.4.22, MIPL 1.0 and the *mobapi* software. For MIPL, one has to specify the home address and the address of the home agent in a configuration file, as MIPL does not support dynamic discovery at run-time. This node can be connected to three different networks by moving the ethernet cable connected to the node to another outlet. The test network

environment is shown in Figure 6.1.

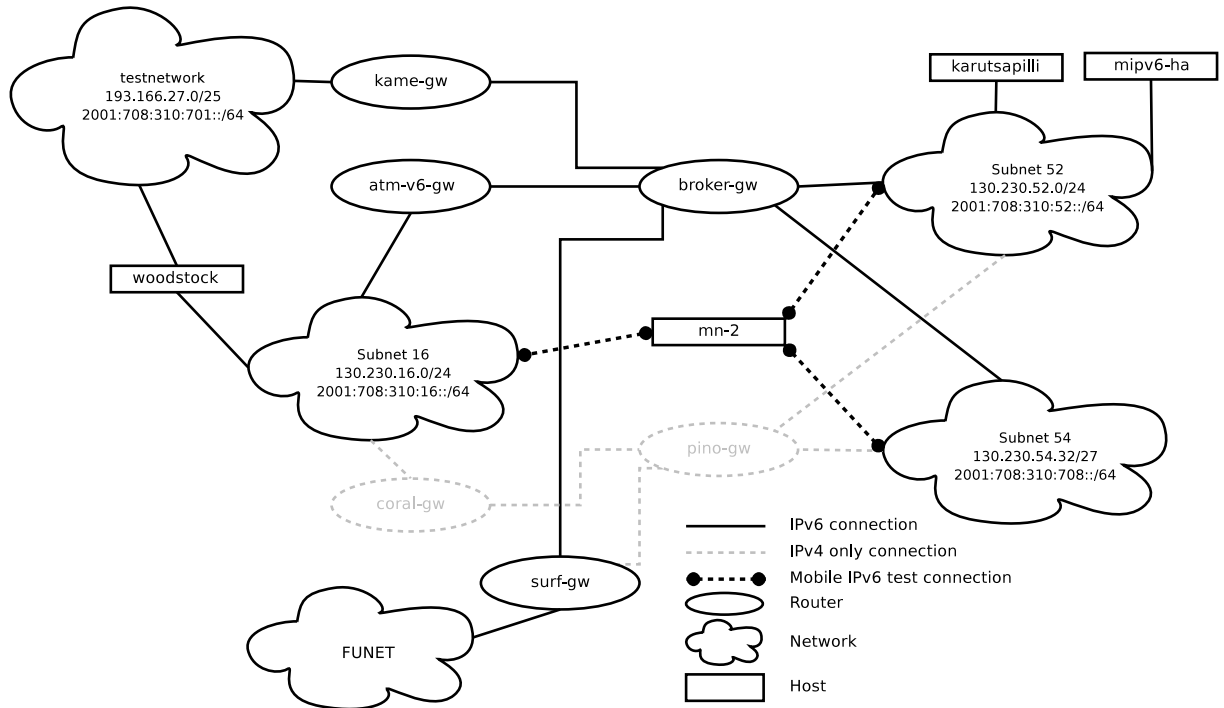


Figure 6.1: Overview of the test network

As can be seen here, all IPv6 connections from the 52 and 54 networks go via the router *broker-gw*.

The home agent *mipv6-ha* is a machine also running MIPL 1.0 with Linux 2.4.22 but without *mobapi* software. It is connected to the 52 network.

As shown in Figure 6.1, the mobile node can roam through several networks. Its home network (and home address) is the 52 network, but it roams to the 16 and the 54 network. For the correspondent node several machines are used, but mostly *karutsapilli* in the 52 network and *woodstock* that is dual-homed in the testnetwork and the 16 network. *Karutsapilli* has the MIPL stack, and can thus function as a fullblown correspondent node (this means that it understands binding updates and thus avoids triangle routing), but *woodstock* does not, so *woodstock* still performs triangle routing.

One issue that came up when testing movement was that *broker-gw* advertised the same link-local address for its interfaces to the 52 and 54 network. This is valid behavior, but it causes the movement detection algorithm to fail. The movement detection algorithm in MIPL defines movement as “reception of a router advertisement that contains a different

link local address and a different network prefix than the current router”. This definition thus leads to the situation that MIPL does not detect movement when the link local addresses are the same. This problem is recognized in the paragraphs 11.5.1 Movement Detection and a solution is specified in 7.2 Modified Prefix Information Option Format of [2]. The router should set the added Router Address bit in its router advertisements. This bit specifies that the prefix field actually contains a global address of the router, and the prefix of that address is the announced prefix in the router advertisement. This way the node sees a different global address in advertisements that come from another subnet, and thus can conclude that the node has moved. However, this solution was not yet implemented in the routing software, so the workaround solution is to make the link-local addresses on different interfaces differ from each other.

Some strange things happened when the home agent was running the router advertisement daemon (radvd) and thus sent router advertisements. This caused routes on other nodes to appear and disappear, so the connection was not stable. This was fixed by stopping radvd on the home agent. Also the source address field of the IPv6 packet did not always contain the correct value. This was fixed by upgrading from MIPL 0.9 with Linux kernel 2.4.20 to MIPL v1.0 with Linux kernel 2.4.22.

## 6.2 Testing and results

To test Mobile IPv6 functionality, ssh sessions were started from *mn-2* to *karutsapilli* and *woodstock*. On these sessions a console-based graphical screensaver (*cmatrix*) was run, and then *mn-2* was moved to another network. It takes some time to pickup the connection (sometimes up to 20 seconds), but then the program could continue smoothly.

To test *mobapi*, a test program is used that can perform blocking and non-blocking calls. The procedure to run *mobapi* is this:

- start module *mip6\_mn* by insmod-ing it (as root)
- start the daemon *mobapid* (as root)
- start the test-program several times (as normal user).

Then *mn-2* roamed to several networks. As several test programs were run simultaneously,

they all received the notification when mn-2 moved. This could be seen clearly in the output. During non-blocking tests, the test-program kept printing dots to the screen while the poll-thread was waiting for movement notifications.

## 7 Conclusions and recommendations

In this report a clean design and a working implementation of a movement notification library for Mobile IPv6 is presented. An application can register interest in movement of a mobile node in blocking and non-blocking modes. The library handles those registrations and retrieves its movement information from the daemon, the daemon dispatches the movement notification that it gets from the kernel, and the kernel part provides the movement information to the daemon. Testing has shown that it works reliably.

The design of *mobapi* is clean and straightforward. The interfaces from part to part are clearly defined. The kernel part is fairly small, and its interfaces to other parts of the kernel are clear. That means that it should be fairly easy to port the work to other Mobile IPv6 stacks, as KAME [14] for the \*BSD platform, or the Linux 2.6 kernel with a new version of MIPL. The daemon and library should be platform independent. They only use standard UNIX functionality, except for the character device driver interface between kernel and daemon, which might be Linux specific. Thus it should be trivial to port them to another platform.

Though the “Mobile IP API” draft [7] is expired, it provides a good interface to applications for receiving movement notifications. However, the best design would be to build the “Mobile IP API” on top of the “Advanced Sockets API”. This last API provides a POSIX compliant way to extract IPv6 information from the IP stack. Thus together these two drafts provide an interface from the kernel, and an interface to the application. In between those two interfaces some connecting glue is needed, so there could be the daemon again, which

retrieves information via the Advanced Sockets API, distills movement information from it, and passes that to the library which provides the Mobile IP API. This solution actually means that the kernel part becomes obsolete. The daemon has to be adapted to be able to retrieve the movement information using the Advanced Sockets API, and the library need not be changed. This work could then be trivially ported to any platform that supports the Advanced Sockets API.

Because there was no implementation of the Advanced Sockets API for MIPL available and time was too short to implement it and it is a sidetrack for the intended work, the decision was made to use the kernel hook as described in this report. However, now KAME has announced support for this API, which means that the work can be ported to the above described system.

As a side result of the movement notification library besides just movement notification, *mobapi* also provides a standardized way to retrieve the home address and care-of address of a mobile node. There is no standard way to do that yet, but by using a blocking call with a short timeout (e.g. 0 ms), these addresses are returned immediately.

# 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] V. Fuller, T. Li, J. Yu, and K. Varadhan, "Classless Inter-Domain Routing (CIDR): an Address Assignment and Aggregation Strategy, RFC 1519," Sep 1993.
- [5] S. Deering and R. Hinden, "Internet Protocol, Version 6 (IPv6), Specification, RFC 2460," Dec 1998.
- [6] S. Chakrabarti and E. Nordmark, "Extension to Sockets API for Mobile IPv6, draft-ietf-mip6-mipext-advapi-00.txt," February 2004.
- [7] 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.
- [8] W. Stevens, M. Thomas, E. Nordmark, and T. Jinmei, "Advanced Sockets API for IPv6, RFC 3542," May 2003.
- [9] P. van den Bergen *et al.*, "Mobile-IP implementations." <http://www.mip4.org/2004/implementations/>.
- [10] MIPL, "MIPL Mobile IPv6 for Linux." <http://www.mobile-ipv6.org>.
- [11] W. R. Stevens, *Advanced Programming in the UNIX Environment*. Addison-Wesley Publishing Company, 1992.
- [12] A. Rubini, *Linux Device Drivers*. O'Reilly & Associates, Inc., 1998.
- [13] R. Russell, "Unreliable guide to locking." <http://www.kernel.org/pub/linux/kernel/people/rustyl/kernel-locking/index.html>, 2003.
- [14] KAME, "KAME Project IPv6/IPsec stack for BSD variants." <http://www.kame.net>.

# Appendix A: Mobile IP API Internet draft

Internet Draft  
Document: draft-yokote-mobileip-api-02.txt  
Expires: December 2003

Alper E. Yegin  
Muhammad M. Tariq  
Aki Yokote  
Guangrui Fu  
Carl Williams  
Atsushi Takeshita

June 2003

## Mobile IP API

### Status of this Memo

This document is an Internet-Draft and is in full conformance with all provisions of Section 10 of RFC2026.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF), its areas, and its working groups. Note that other groups may also distribute working documents as Internet-Drafts.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

The list of current Internet-Drafts can be accessed at  
<http://www.ietf.org/ietf/lid-abstracts.txt>

The list of Internet-Draft Shadow Directories can be accessed at  
<http://www.ietf.org/shadow.html>.

### Abstract

The Mobile IP API provides an interface between the mobility management module and the application layer. Using Mobile IP API, applications can extract the mobility information that is already maintained by the mobility management module. This API provides mobility awareness for applications.

This document describes application scenarios followed by requirements and definition of Mobile IP API. Application scenarios are example applications that can take advantage of awareness of the mobility information.

## Table of Contents

1. Introduction.....	3
2. Key Words.....	4
3. Usage Scenarios.....	4
4. Requirements.....	5
4.1 Location awareness.....	5
4.2 Movement awareness.....	5
4.3 Use of API.....	5
4.4 Scope of Mobile IP API.....	5
5. Data Structures and Constants.....	5
6. Functions.....	6
6.1 Retrieving Information on All Mobile Nodes.....	6
6.2 Retrieving Information on a Given Mobile Node.....	6
6.3 Movement Notification.....	7
6.3.1 Blocking Mode.....	7
6.3.2 Non-blocking Mode.....	7
6.3.3 Registering Callback Function.....	7
6.3.4 Deregistering Callback.....	8
6.4 Verifying location.....	9
7. Security Considerations.....	9
8. References.....	9
9. Author's Addresses.....	9
10. Full Copyright Statement.....	10

1. Introduction

The Mobile IP API is an interface between mobility management and application layer. The basic functionality of the Mobile IP API is to enable applications to get mobility information about the host they are running on or other hosts they are communicating with. It is assumed that the mobility management (module) already has the mobility information (via Mobile IP signaling). There is no need to generate any extra signaling between the Mobile Node (MN) and the Correspondent Node (CN), or change the functionality of current Mobile IP [3][5] functions for the host to obtain mobility information.

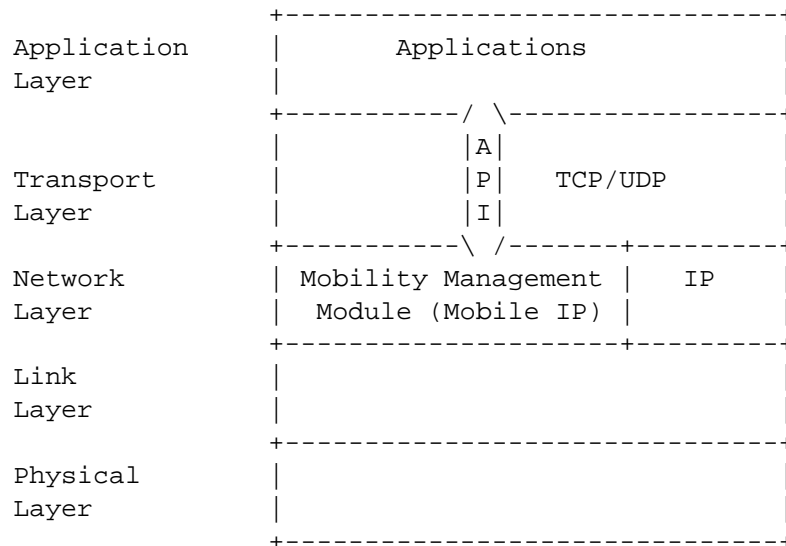


Figure-1.0 Network stack and Mobile IP API

Mobile IP has been designed to provide network layer mobility in a transparent manner. Host mobility can be managed without having to involve applications in any way. There is no need to let the application layer know about underlying mobility of mobile terminals. However, it does not mean that the applications should not know about mobility of mobile terminal. It can be useful to pass mobility information from network layer to the applications. A mobility aware application can exploit such information in many different ways. Some of these are discussed in usage scenarios section.

Depending on the access technology and network architecture (including access points deployment plan), learning an IP address within a topology can also imply learning geographical location with some level of accuracy. While this API provides location information in terms of topologically correct IP addresses (let's call this

routing-based location), mapping that to geographic location is outside the scope of this API. External mechanisms, such as subnet-location database lookups, can be used for this purpose.

An application running on the MN can learn the location of the host within IP topology by using this API. Similarly, an application running on the CN can learn the routing-based location of MNs it's communicating with. Applications can also detect movement by observing change in IP addresses.

Mobile IP API does not generate any extra signaling between the MN and CN. The API only relies on receipt of Binding Update (BU) on the CN to provide information to the applications running on the CN.

## 2. Key Words

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 RFC-2119 [1].

## 3. Usage Scenarios

This section describes the example application scenarios for the Mobile IP API. The following scenarios apply to one of four classes of usage.

(a) MN queries its own routing-based location and verifies whether it is home or in a visiting network. Applications that take different actions based on the location of the local host can use this functionality. For example, in case network access fees are higher in visited networks, an e-mail client might choose not to download attachments when it detects the local host is away from home network.

(b) MN gets notification of its own movement, and learns the new routing-based location. Applications that take actions upon any movement can use this functionality. For example, SLP [2] discovery can be kick started as soon as movement is detected. This would increase the performance of the service by proactively discovering servers upon movement.

(c) CN queries the MN's location. Location-based service providers can utilize this functionality. For example, a web server can deliver location-based content by determining the location of the clients (MNs).

(d) CN gets notification of MN's movement, and learns the new routing-based location.

## 4. Requirements

This section describes the requirements for the Mobile IP API design. Current version of the Mobile IP API is designed for Mobile IPv6 [3]. It can easily be extended to support Mobile IPv4 [5].

### 4.1 Location awareness

Mobile IP API must enable applications to learn MN's routing-based location (i.e., MN's care-of address). These applications might be running on the MN itself, or on a CN that this MN is corresponding with.

### 4.2 Movement awareness

Mobile IP API must enable applications to get notified when MN changes location. These applications might be running on the MN itself, or on a CN that this MN is corresponding with.

### 4.3 Use of API

Mobile IP API should have read-only functions and must not effect the operation of any other application or the mobility module on the host.

### 4.4 Scope of Mobile IP API

Mobile IP API must be designed specific to Mobile IP, and should not generate new signaling or change functionality of the Mobile IP protocols [3].

## 5. Data Structures and Constants

The `mobile_node_t` can hold home address and care-of address of a MN.

```
struct    mobile_node_t {
    struct in6_addr    home_addr;    /* home address of MN */
    struct in6_addr    co_addr;      /* care-of address of MN */
    struct in6_addr    ha_addr;      /* MN's home agent */
}
```

Apart from above mentioned data structures, following are also defined.

```
#define MIP_MN_MOVED      1
#define MIP_BCE_DELETE   2
#define MIP_CB_DEREGISTER 3
```

These definitions are used by movement notification API (see section 6.3)

## 6. Functions

### 6.1 Retrieving Information on All Mobile Nodes

Applications can call the `mip_get_all_mobile_nodes()` function to get information about all mobile nodes maintained by the mobility management module.

```
int mip_get_all_mobile_nodes(mobile_node_t **mn_list,
                             int local)
```

Since the number of mobile nodes may not be known in advance, the function itself allocates appropriate amount of memory to hold all the mobile nodes and assigns that to `*mn_list`.

The local argument determines whether this function should list all the mobile nodes running on the same host as the caller, or all the mobile nodes this host is communicating with. Latter simply dumps the content of the binding cache of a correspondent node. The value of this field should be one for local, and zero for non-local.

If there was no mobile node existed in memory, the function shall return zero. If there was an error, the function shall return -1.

### 6.2 Retrieving Information on a Given Mobile Node

Applications can call the `mip_get_one_mobile_node()` function to get information about a specific mobile node.

```
int mip_get_one_mobile_node(mobile_node_t *mobile_node)
```

Desired mobile node's home address should be stored in the `home_addr` field of the structure.

This function shall return positive value for success, return zero if the mobile node is unknown, and return -1 if there is any error.

### 6.3 Movement Notification

Applications can call `mip_notify_movement()` function to get a notification upon movement of mobile nodes. Applications can do so either in non-blocking mode by registering a callback function for notification, or in blocking mode function call blocks until the event of interest is detected.

```
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))
```

If the home address is set to unspecified IPv6 address (i.e. 0::0) then it is treated as a wildcard and the application will get notification for all mobile nodes.

#### 6.3.1 Blocking Mode

If the value of `non_blocking` (flag) argument is set to zero, then the function will not return until a new binding update for mobile node whose home address is specified in `mobile_node` is received or `timeout_ms` milliseconds are passed (whichever comes first).

Upon return, the `co_addr` field of the `mobile_node` is initialized to the latest care-of address of the mobile node. The function shall return zero, if timeout occurred; otherwise a return positive value.

A possible error is Unknown Home Address, and others may be defined later.

#### 6.3.2 Non-blocking Mode

If the value of `non_blocking` argument is set to non-zero, then this function call will register or deregister a callback function.

#### 6.3.3 Registering Callback Function

When the `non_blocking` (flag) argument is set to a non-zero value, a (pointer to) callback function must be provided as well. This callback function is called when specified mobile node(s) moves.

During callback, the callback function is passed with an initialized `mobile_node` structure. The `cb_parameter` is passed to the callback function as is, and the event is set to an appropriate value such as `MIP_MN_MOVED`. Other events include `MIP_CB_DEREGISTER`, `MIP_BCE_DELETE` (binding cache entry deleted). The description of these events is as follows:

(a) `MIP_MN_MOVED`

The mobility management module has received a binding update indicating that the mobile node has acquired a new care-of-address.

(b) `MIP_CB_DEREGISTER`

The call back function for the specified address has been deregistered. This may occur due to application calling `mip_notify_movement()` function with `non_blocking` flag set to non-zero and `callback` set to `NULL`. This may also occur due to deletion of (all of) home-address(es) that the callback was registered for.

(c) `MIP_BCE_DELETE`

The binding cache entry has been deleted for a reason such as expiration of lifetime.

The value of `timeout_ms` will be ignored when callback registration occurred.

A possible error is Unknown Home Address, and others may be defined later.

#### 6.3.4 Deregistering Callback

If the value of `callback` argument is set to `NULL`, then this call will de-register any callbacks registered for the specific `mobile_node(s)` (wildcard address is allowed here as well).

The callback function is called one final time with `mobile_node` (containing latest binding), `cb_parameter`, and `MIP_CB_DEREGISTER` as the event.

The possible errors are due to no callback registered by the calling process for that home address, and unknown home address.

#### 6.4 Verifying location

This macro can be used for determining whether a given mobile node is at home or away from home.

```
int IS_AT_HOME(struct mobile_node_t *)
```

This macro returns non-zero if the mobile node is at home (i.e. the care of addresses is same as the home address of the mobile node); otherwise it returns zero.

#### 7. Security Considerations

As previously stated, Mobile IP API does not generate extra signaling between the MN and the CN. For providing information to the applications running on the CN, the API only relies on receipt of Binding Updates on the CN. If the MN does not send a Binding Update for any reason including location privacy, this API will not provide any information to the applications running on the CN. As such, this API does not reveal any information other than what MN is willing to provide.

#### 8. References

- [1] S. Bradner, "Key words for use in RFCs to Indicate Requirement Levels", RFC 2119, March 1997.
- [2] E. Guttman, C. Perkins, J. Veizades, M. Day. "Service Location Protocol, Version 2". RFC 2608, June 1999.
- [3] D. Johnson, C. Perkins, J. Arkko. Mobility Support in IPv6. draft-ietf-mobileip-ipv6-22.txt, May 2003.
- [4] J. Myers, M. Rose. "Post Office Protocol - Version 3". RFC 1939, May 1996.
- [5] C. Perkins, editor. IP Mobility Support. RFC 3344, August 2002.

#### 9. Author's Addresses

Alper Yegin  
DoCoMo Communications Laboratories USA, Inc.  
181 Metro Drive, Suite 300  
San Jose, CA 95110  
Phone: +1-408-573-1050 (main)  
Fax: +1-408-573-1090  
E-mail: alper@docomolabs-usa.com

Muhammad Mukarram Bin Tariq  
DoCoMo Communications Laboratories USA, Inc.  
181 Metro Drive, Suite 300  
San Jose, CA 95110  
Phone: +1-408-573-1050 (main)  
Fax: +1-408-573-1090  
E-mail: tariq@docomolabs-usa.com

Guangrui Fu  
DoCoMo Communications Laboratories USA, Inc.  
181 Metro Drive, Suite 300  
San Jose, CA 95110  
Phone: +1-408-573-1050 (main)  
Fax: +1-408-573-1090  
E-mail: fu@docomolabs-usa.com

Carl Williams  
MCSR Labs  
3790 El Camino Real, #154  
Palo Alto, CA 94306  
Phone: +1-650-279-5903  
E-mail: carlw@mcsr-labs.org

Atsushi Takeshita  
NTT DoCoMo, Inc.  
3-5m Hikarinooka, Yokosuka,  
Kanagawa, 239-8536 Japan  
E-mail: takeshita@mml.yrp.nttdocomo.co.jp

#### 10. Full Copyright Statement

Copyright (C) The Internet Society (2001). All Rights Reserved.

This document and translations of it may be copied and furnished to others, and derivative works that comment on or otherwise explain it or assist in its implementation may be prepared, copied, published and distributed, in whole or in part, without restriction of any kind, provided that the above copyright notice and this paragraph are included on all such copies and derivative works. However, this document itself may not be modified in any way, such as by removing the copyright notice or references to the Internet Society or other Internet organizations, except as needed for the purpose of developing Internet standards in which case the procedures for copyrights defined in the Internet Standards process must be followed, or as required to translate it into languages other than English.

The limited permissions granted above are perpetual and will not be revoked by the Internet Society or its successors or assigns. This document and the information contained herein is provided on an "AS IS" basis and THE INTERNET SOCIETY AND THE INTERNET ENGINEERING TASK FORCE DISCLAIMS ALL WARRANTIES, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY WARRANTY THAT THE USE OF THE INFORMATION HEREIN WILL NOT INFRINGE ANY RIGHTS OR ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.